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.

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.

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

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.

- 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.

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.

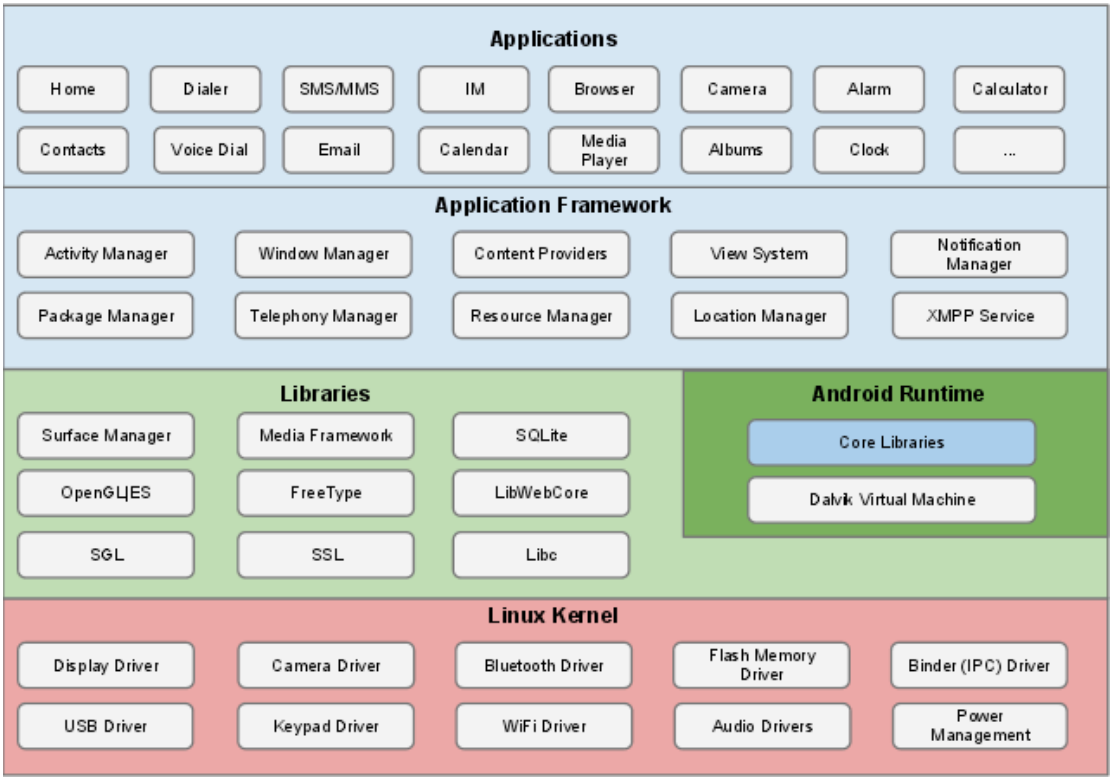

Figura 1: Pilha de software do Android. Fonte: [Android Open Source Project 2012a].

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.

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.

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.

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

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

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

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*

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

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

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

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

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;

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,

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;

- 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:

  - *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.

- 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;

- 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:

- **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*.

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.

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

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

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.

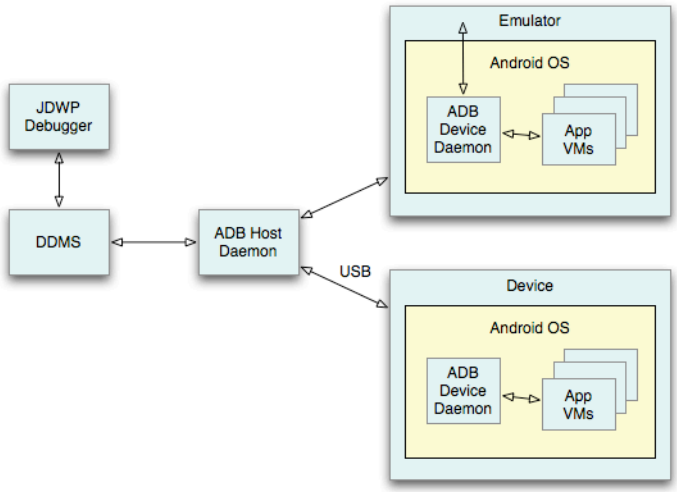

Figura 3: Ferramentas para monitoramento e depuração no Android SDK (*host*) e SO Android (alvo). (Fonte: [Android Developers Project 2012a[])

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

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

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,

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.

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

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.

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

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.

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.facebook.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].

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

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,

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.

- **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

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.

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.

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.

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.

