Capítulo 5

Introdução à segurança de aplicações para a TV digital interativa brasileira

Alexandre Melo Braga[1], Gilmara Santos Restani[1]

[1]Fundação CPqD – Centro de Pesquisa e Desenvolvimento em Telecomunicações. Rod. Campinas-Mogi-Mirim (SP-340) km 118,5 – 13086-902 – Campinas – SP – Brazil.

ambraga@cpqd.com.br, grestani@cpqd.com.br

Abstract

This chapter contains an overview of the security aspects of the Brazilian interactive digital TV. The text is structured around technical security assessments of set top boxes and shows preliminary results obtained by the contributing authors on security assessments of interactive applications for these devices. The text deals with the aspects of secure programming, secure execution environment, and security of receivers, when connected to an IP network. Further, several vulnerabilities found in the software technologies associates to the TVD receivers are shown. The vulnerabilities identified and documented in this text have parallel in known vulnerabilities found in other programming languages and traditional vulnerabilities catalogs.

Resumo

Este capítulo descreve resultados preliminares obtidos pelo CPqD na avaliação de segurança de aplicações interativas para a TV digital brasileira. Foram avaliados os aspectos de programação segura, ambiente de execução segura dos aplicativos e segurança de receptores conectados a uma rede IP. Foi iniciado um trabalho de identificação e documentação de vulnerabilidades de programação insegura, de modo que vulnerabilidades em Lua encontrem paralelo em vulnerabilidades já conhecidas em outras linguagens de programação.

5.1. Introdução

No Brasil, os receptores de televisão digital interativa (TVDi) tendem a se tornar instrumentos de inclusão digital para grandes parcelas da população. Nos próximos anos, diversos serviços serão oferecidos via televisão digital. Muitos dos quais somente serão possíveis com o atendimento a requisitos de segurança fortes. Para atender a demanda prometida, os receptores de televisão digital, enquanto equipamentos de computação e comunicação possuirão capacidade computacional equivalente a dos computadores pessoais mais simples; ou ainda, a dos telefones celulares mais sofisticados.

O desafio em relação à plataforma de software de TVDi brasileira está em oferecer pelo menos o mesmo grau de confiança de que gozam as plataformas mantidas pelos gigantes da indústria de software. Por outro lado, as implementações das linguagens de programação da plataforma de TVDi ainda não sofreram o mesmo grau de escrutínio da comunidade científica de segurança que as implementações de linguagens utilizadas pelos grandes produtores mundiais de software. Além disso, por ainda não serem tão universais quanto os sistemas de mercado, elas também não sofreram o mesmo grau de exposição às ameaças comuns aos sistemas de Internet, de computação doméstica e corporativa.

Este trabalho foi motivado pela ausência, até onde vai o conhecimento dos autores, tanto de documentação técnica especializada, quanto de ferramentas de mercado, seja para programação segura, seja para proteção de aplicações interativas, na TVDi brasileira, em particular para as tecnologias da plataforma Ginga. A lista bibliográfica reflete a ausência de textos técnicos e científicos nesta área. Vale ressaltar que não foram encontradas, até o momento de submissão deste minicurso, referências bibliográficas sobre desenvolvimento seguro e nem sobre programação segura em Ginga e TVDi.

O texto a seguir está organizado da seguinte forma. A seção 5.2 descreve de forma resumida a plataforma da televisão digital brasileira, descreve as aplicações interativas e as camadas do *middleware* Ginga embarcado em um receptor de tv digital. A seção 5.3 aborda a segurança do receptor de TVDi de acordo com a segurança de sistemas embarcados, a segurança nos dispositivos portáteis, e trata os receptores de TVDi como plataforma de computação confiável para transações comerciais. E de forma resumida mostra a iniciativa de normatização dos mecanismos de segurança de aplicativos no sistema brasileiro de televisão digital. Na seção 5.4 é apresentado um catalogo resumido das vulnerabilidades de programação insegura identificadas nas linguagens de programação utilizadas pelo Ginga, em particular, o NCLua. A seção 5.5 mostra, com ferramenta construída para tal, mas em ambiente simulado, como realizar avaliações de segurança em receptores de TVDi com Ginga. A seção 5.6 traz uma abordagem sistêmica para a segurança da informação e dos sistemas embarcados nos receptores de TVDi onde serão tratados aspectos de desenvolvimento seguro de aplicações, em particular aspectos de projeto seguro de sistemas, programação segura, robustecimento de sistemas operacionais, segurança em redes e implantação e operação seguras. Todos estes aspectos de segurança serão contextualizados na utilização do receptor de TVDi como dispositivo de computação e comunicação. E por fim a seção 5.7 apresenta as considerações finais e trabalhos futuros.

5.2. Visão Geral da TV Digital interativa brasileira e suas aplicações

Esta seção contém uma visão geral bastante resumida da plataforma de televisão digital brasileira. A maior parte do conteúdo será voltada para a descrição da arquitetura de software de apoio às aplicações interativas, as camadas de software tipicamente embarcadas em um receptor de TVDi (*set-top box*), o *midlleware* ginga, e a estrutura das aplicações interativas, Ginga-J, Ginga-NCL, Lua e a ponte NCLua.

Um pouco da história da experiência do CPqD - Centro de Pesquisa e Desenvolvimento com a TV Brasileira. Desde da década de 90, o CPqD vem assessorando o Governo Federal e os organismos competentes (ANATEL e Ministério das Comunicações) de forma geral, na definição do modelo de TV digital brasileiro. A vasta experiência e elaborações de documentos no cenário da TV digital fazem com que seus colaboradores se preocupem cada vez mais com a questão de adequação de padrões e regulamentações nos componentes deste cenário. Desta maneira, uma breve descrição do cenário de TV digital se faz necessária.

O sistema atual de televisão digital na grande maioria do país ainda é híbrido, ou seja, analógico-digital. Mas, este cenário já começou a ser alterado e algumas operadoras de TV aberta já estão transmitindo o sinal digital. Para a recepção da sinalização digital é utilizado um receptor, aparelho responsável pela conversão destes sinais. Em algumas cidades, a produção, a edição e o armazenamento de conteúdo são quase inteiramente digitais.

Desta forma, o usuário final é beneficiado com: uma qualidade superior de áudio e vídeo; o transporte digital dos sinais, os quais permitem uma recepção mais limpa e menos suscetível às interferências, inclusive em receptores móveis e portáteis. E ainda, a diversidade de programação é maximizada com a utilização de padrões do grupo MPEG- *Moving Picture Experts Group*. Soma-se a todos estes benefícios a característica potencialmente mais interessante, a distribuição de diversos serviços interativos agregados à programação, tais como: eventos esportivos com câmeras, placar em tempo real, legendas e áudio em vários idiomas, informativos de previsão de tempo e indicativos do mercado financeiro entre outras possibilidades. A facilidade de um canal de retorno entre o usuário e a emissora permite outros serviços mais completos como transações bancárias, por exemplo.

A tecnologia de TV digital cria um novo ambiente, onde a experiência de assistir televisão é transformada, ou seja, é possível uma interatividade do usuário final, o qual pode ter um papel ativo perante o remetente dos dados via canal de retorno. Este novo ambiente permite um cenário com diversas possibilidades, oportunidades e desafios e entender este processo e a tecnologia envolvida na disponibilização desses serviços interativos é fundamental para o entendimento do foco deste trabalho.

A arquitetura genérica comum aos sistemas de TV digital basicamente é uma plataforma multimídia capaz de transmitir sinais de áudio e vídeo de alta qualidade, e também dados, utilizando o sinal de radiodifusão em freqüências de VHF – *Very High Frequency*/UHF – *Ultra High Frequency*. A capacidade de transmissão de dados, podendo ou não estar vinculados à programação, permite o desenvolvimento de novos serviços e aplicações.

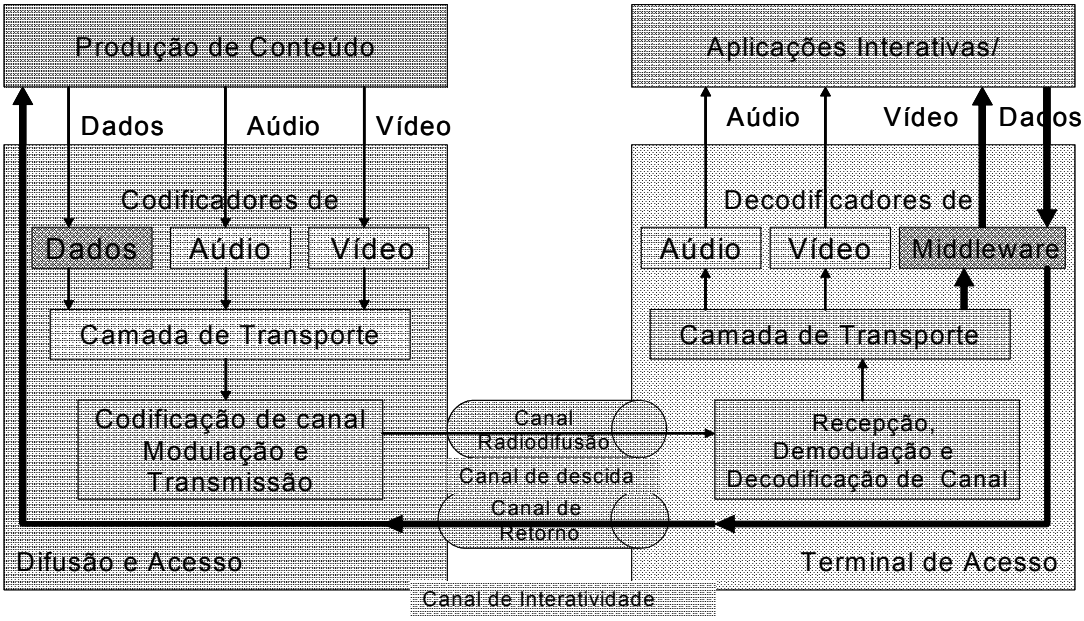

Figura 1 - Diagrama de Fluxo de Informação.

A Figura 1 mostra de forma sintetizada a arquitetura sistêmica do modelo de TV digital Aberta através da representação do fluxo de informação. A parte de Difusão e Acesso, do lado esquerdo da figura, é composta pelos módulos de codificação e empacotamento das informações a serem transmitidas para os receptores digitais. O lado direito da figura é à parte do Terminal de Acesso, ou seja, o receptor de TVDi , foco deste trabalho, e é composta por módulos necessários para efetuar o processamento reverso da parte de Difusão e Acesso, reconstituindo as informações originais de áudio, vídeo e dados. O sinal é recebido por meio de antenas receptoras, e passa pelo processo de demodulação de canal, de onde resulta o sinal de transporte que será enviado à etapa de demultiplexação, no módulo da Camada de Transporte. Nesta camada os sinais são codificados em áudio e vídeo e são submetidos aos respectivos decodificadores, e os dados são submetidos ao *middleware*. Os primeiros reconstituem os sinais originais. No caso do *middleware*, além do trato das instruções, funciona também como uma plataforma de execução de software e será descrito mais à diante. E como resultados finais têm-se as Aplicações Interativas, utilizadas pelo usuário final.

Além disto, o sistema de TV digital Aberta possui um Canal de Interatividade, composto de um Canal de Descida e um Canal de Retorno, o qual possibilita a interação do usuário final com a Produção de Conteúdo, permitindo-lhe receber ou enviar solicitações e informações. Neste trabalho, o foco é o terminal de acesso e a seguir especificamente serão mais detalhados o Canal de Interatividade e o *Middleware*.

O Canal de Interatividade é o subsistema de TV digital responsável pela infraestrutura para a comunicação das Aplicações Interativas, do Terminal de Acesso com os servidores de aplicação do Provedor de Conteúdo. Este canal é conceituado como o meio que o usuário pode interagir encaminhando ou recebendo dados/programações das emissoras de TV. Este canal é formado por dois canais de comunicação: Canal de Descida – o qual estabelece a comunicação via radiodifusão das emissoras para o usuário final; e o Canal de Retorno – compreendido por qualquer tecnologia de redes de acesso que estabeleça a comunicação IP (*Internet Protocol*) no sentido do usuário para

as emissoras. No lado do receptor é possível perceber as diferentes camadas que compõe esta arquitetura conforme mostra a Figura 2.

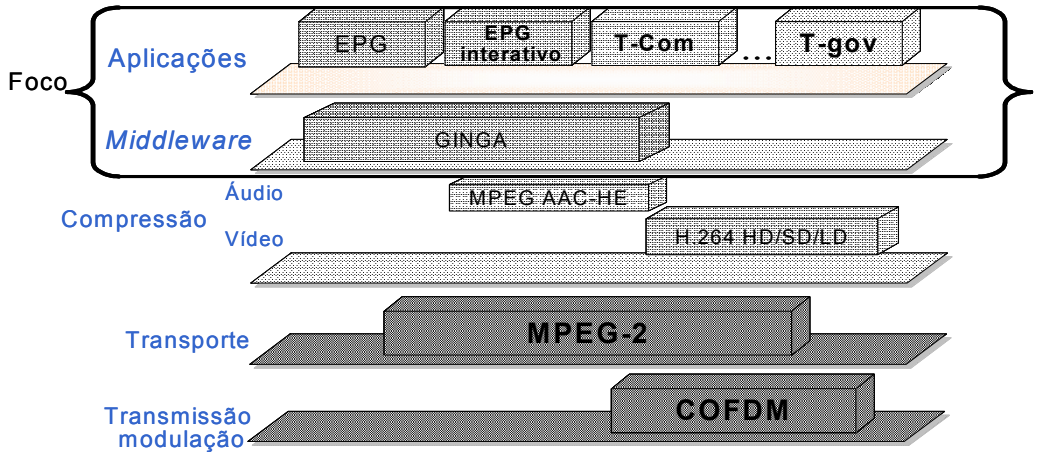

Figura 2- Canal de Interatividade

Na Figura 3 a seguir é possível verificar que o receptor de TVDi recebe o canal de radiodifusão de diversas emissoras/programas, que contém dados de interatividade do Canal de Descida no feixe de transporte (TS). Para a interface entre o receptor de TV digital e as Redes de Comunicações é necessário um modem adequado à solução de rede alocada pelo usuário final.

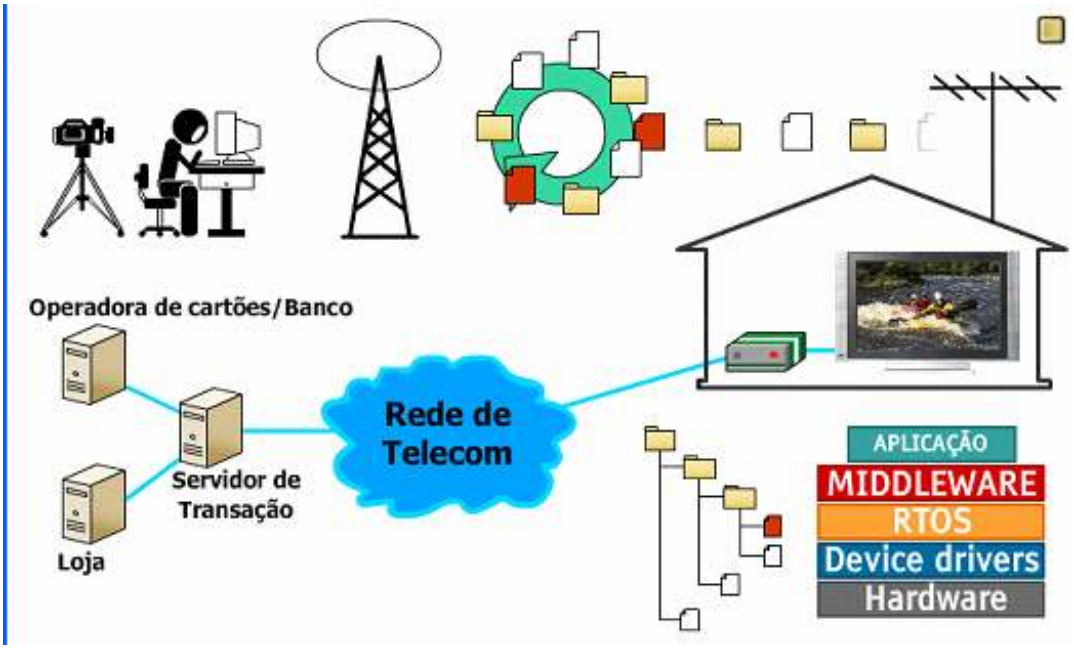

Figura 3 - TV Digital e seus componentes.

Na Figura 3 também é possível verificar a presença de aplicações no desenho menor representando o carrossel. O desenho demonstra o meio de distribuição do conjunto de dados no sistema de radiodifusão denominado *Object Carousel*. Os dados são

transmitidos como um conjunto de objetos de arquivos e diretórios (File e *Directory objects*). O *File Object* é o próprio conteúdo do arquivo que se deseja transmitir. Ele é composto de referências (via ponteiros) a outros arquivos ou diretórios e a representação dos dados ocorre de forma hierárquica. Cada objeto presente no *Object Carousel* é transmitido em uma única messagem chamada BIOP (*Broadcast Inter ORB Protocol*). Basicamente os princípios do DSM-CC (*Digital Storage Media – Command & Control*) são: *Data Carousel*: este é utilizado na transmissão de blocos de dados, não havendo nenhuma indicação do significado do dado transmitido, ficando a interpretação por conta do receptor. E o princípio do *Object Carousel* é construído no topo do carrossel sendo similar a um sistema de arquivos.

A seguir será detalhado o *Middleware* SBTVD presente na solução de TV digital aberta.

Segundo Soares o *middleware* SBTVD é uma camada de software posicionada entre os códigos das aplicações e a infraestrutura de execução (plataforma de hardware e sistema operacional) como ilustrado na Figura 4. O *middleware* no padrão brasileiro consiste de máquinas de execução das linguagens oferecidas, e bibliotecas de funções, que permitem o desenvolvimento rápido e fácil de aplicações.

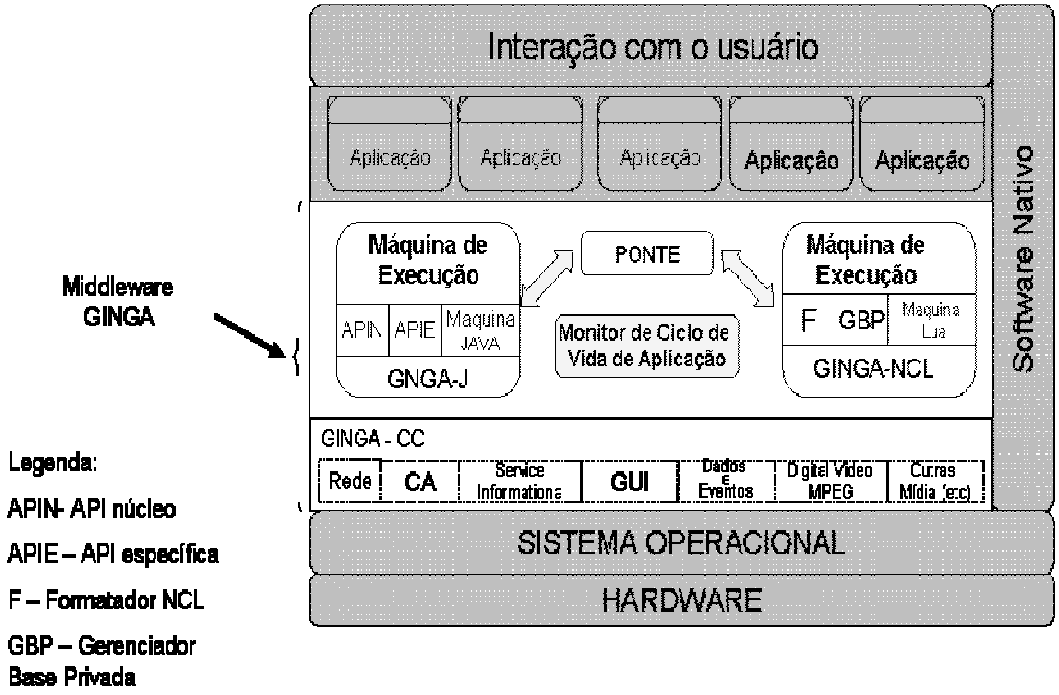

Figura 4 - Arquitetura *Middleware* SBTVD – GINGA.

O universo das aplicações pode ser dividido em três módulos principais: Ginga-CC, Ginga-NCL e Ginga-J, sendo que os dois últimos compõem a camada de serviços específicos e estão separados pelo tipo de aplicações que são responsáveis. O GINGA-NCL é responsável pelas Aplicações Declarativas e o GINGA –J pelas Aplicações Interativas.

A máquina de apresentação declarativa é baseada na linguagem NCL –*Nested Context Language*, uma linguagem declarativa para autoria de documentos multimídia e trata de interpretação semântica do modelo NCM – *Nested Context Model*. Este último é um modelo conceitual e descreve as estruturas de dados, os eventos e o relacionamento entre estes dados, e também definem as regras de estruturação e operações sobre os dados para a manipulação e atualização das estruturas. O Ginga-J é responsável pela maquina da apresentação imperativa e utiliza a linguagem Java. O Ginga-J está dividido em três módulos: a máquina virtual Java; o núcleo e suas APIs(verde); e o módulo responsável pelo suporte às APIs específicas (amarela e vermelha) do Ginga-J.

O *middleware* brasileiro deve apresentar as duas máquinas de aplicações declarativas e imperativas nos receptores fixos e móveis, sendo somente exigida a maquina declarativa em receptores portáteis.

5.3. Cenário de ameaças à TVDi e às aplicações interativas

Esta seção aborda a segurança do receptor de TVDi de acordo com três linhas de atuação. A primeira é a segurança de sistemas embarcados. A segunda é a segurança de dispositivos portáteis, mas com grande poder de computação e comunicação (similares aos *smartphones*). A terceira trata os receptores de TVDi como plataforma de computação confiável para transações comerciais. Além disso, será tratada a iniciativa de normatização dos mecanismos de segurança de aplicativos no sistema brasileiro de televisão digital. As ameaças serão avaliadas no contexto de aplicações com requisitos de segurança fortes, como por exemplo, o oferecimento de serviços bancários pela TVDi (*t-banking*) e de comércio eletrônico pela TVDi (*t-commerce*) et al.

Esta seção contém a análise das questões de segurança relacionadas ao receptor de TVDi e a linguagem Lua. A Figura 5 é uma extrapolação de figura análoga de *Ravi*, et al (2004) e ilustra as necessidades de segurança de cada participante da cadeia de valor da TVDi, desde a produção de hardware, passando pela geração de conteúdo televisivo, até o oferecimento de serviços interativos, vinculado à programação, para o usuário final. Na Figura 5, no lado esquerdo (borda quadrada) estão as ameaças mais comuns associadas a cada nível da pilha de valor. No lado esquerdo, o requisito de segurança correspondente à ameaça em questão.

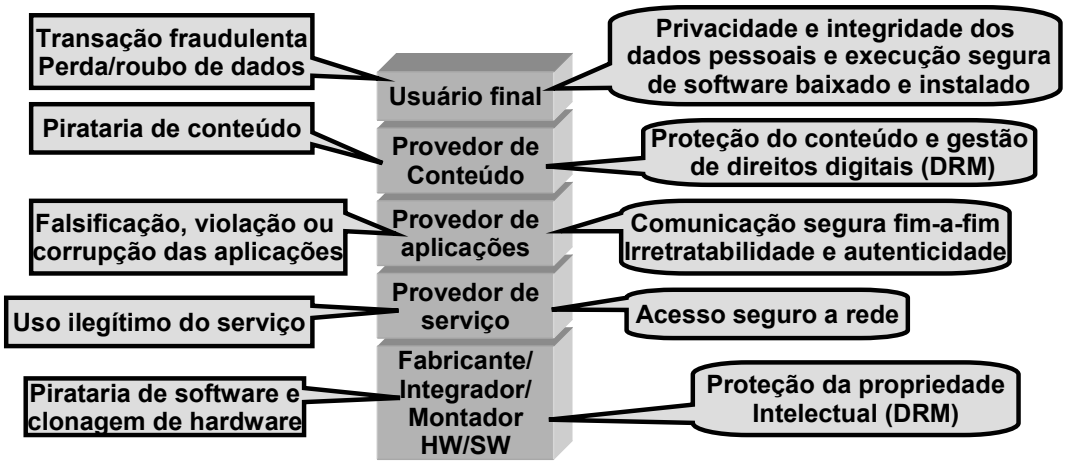

Figura 5 - Necessidades de segurança em cada nível da pilha (cadeia) de valor agregado do sistema de TVDi.

O trabalho apresentado neste texto é principalmente voltado para proteção das aplicações (camada 3 da pilha de valor) e da privacidade e integridade dos dados de usuários finais (camada 5).

5.3.1. Semelhanças de segurança de *smartphones* e de receptores de TVDi

Quando confrontadas por problemas semelhantes, as indústrias de receptores de TVDi e de *smartphones* responderam com soluções semelhantes. Esta seção traça paralelos entre as técnicas de ataque aos *smartphones* e aos receptores TVDi. Assim como também faz analogias entre as estratégias de seguranças adotadas por estas indústrias.

5.3.1.1 Evolução das técnicas de ataque aos *smartphones* e aos receptores TVDi

A tendência mundial de evolução das técnicas de ataque aos *smartphones* e às redes subjacentes de telecomunicações pode ser comparada ao estado das técnicas de ataque sobre PCs conectados a Internet e, mais recentemente, nos aparelhos de telefones celulares inteligentes (*smartphones*), e os receptores de televisão digital interativa. Conforme ilustrado pela Figura 6.

No caso dos PCs, foi na década de 80 que a evolução das técnicas de ataque, em particular, os softwares maliciosos, começou a afetar o público em geral. Naquela época, os espécimes de softwares maliciosos típicos eram os vírus de computador, os quais contaminavam o setor de boot dos discos rígidos e disquetes. Atualmente existem redes inteiras comprometidas e controladas pelo crime organizado para realização dos mais diversos tipos de ataques, incluindo os ataques maciços coordenados pelas *Botnets*. Este texto não detalha a história dos vírus de computador, por se tratar de informação pública e amplamente disponível.

De acordo com Hypponen (2007), o primeiro vírus de *smartphone* surgiu em 2004, desde então há relatos de centenas de softwares maliciosos para estes aparelhos, conforme Bickford, et al (2010). Os ataques mais sofisticados às redes via *smartphones*, com o controle remoto de diversos aparelhos para a realização de ataques coordenados, já são considerados, em textos acadêmicos, uma possibilidade técnica e seriam possíveis em teoria, conforme Traynor, et al (2009). Segundo *Oberheide and Jahanian* (2010),

Oberheide, et al (2008) e Cai, *Machiraju and Chen* (2009), os *smartphones* serão uma fonte de vazamento de informações privadas em redes públicas e ambientes de computação em nuvem, serão ainda vetores de ataques maciços às redes de computadores e de telecomunicações, assim como representarão a próxima fronteira de proliferação dos softwares maliciosos.

De forma análoga ao que tem ocorrido com os PCs e ao que ocorre com os *smartphones*, espera-se ainda nesta década um aumento significativo da quantidade de incidentes de segurança envolvendo os receptores de TVDi.

Ainda, os estudos, experimentos e trabalhos correlatos realizados pelos autores deste texto levam a crer que a semelhança entre os aspectos de segurança de *smartphones* e de receptores de TVDi vão além do paralelismo das técnicas de ataque, abrangendo também, com grande similaridade, as técnicas de segurança e proteção. Conforme descrito na próxima seção.

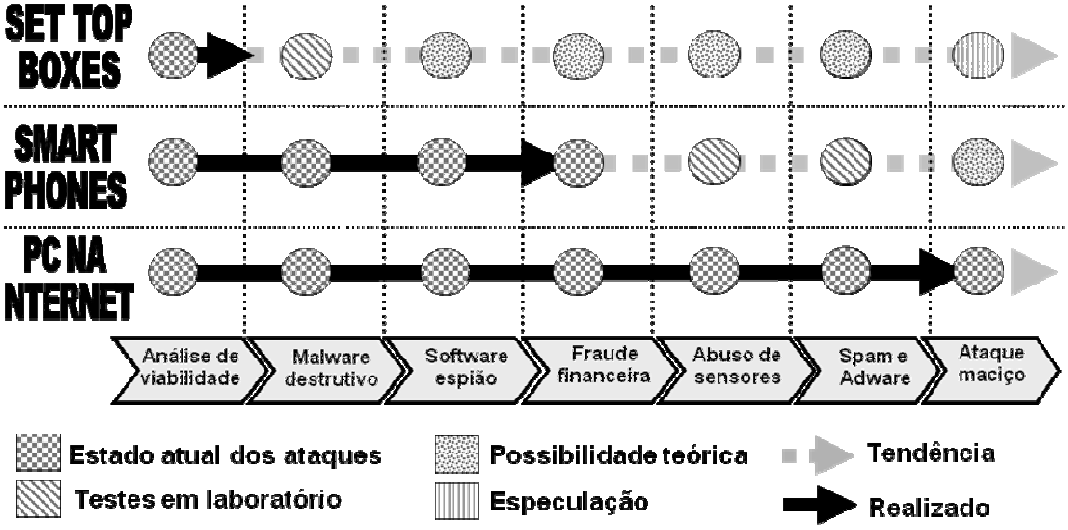

Figura 6 - Paralelo de evolução das técnicas de segurança em três tipos de tecnologias, computadores pessoais conectados a Internet, smartphones com grande poder computacional e receptores de TV digital interativa.

5.3.1.2 Analogias na implementação de mecanismos de segurança

A aderência a normas e padrões é uma parte importante da garantia de integridade de qualquer peça de tecnologia. O receptor de TVDi, em relação aos aspectos de segurança, está sujeito a norma ABNT NBR 15605. Esta norma especifica os mecanismos do sistema de segurança para o sistema brasileiro de televisão digital terrestre. A primeira parte desta norma, ABNT NBR 15605-1 (2008), trata de controle de conteúdo; isto é, DRM (*Digital Rights Management*).

Na segunda parte desta norma, ABNT NBR 15605-2 são definidos os mecanismos de autenticação dos receptores, dos dispositivos externos e de usuários, além das questões de segurança e autenticação de aplicativos interativos, assim como do canal de interatividade.

Para *Oberheide* e *Jahanian* (2010), as plataformas de software modernas para *smartphones* implementam uma arquitetura de segurança baseada em três pilares. Primeiro, a entrega segura de aplicações, a qual está relacionada à habilidade de uma plataforma móvel em verificar a integridade e a autenticidade de origem de uma aplicação a ser instalada no dispositivo móvel. Segundo níveis de confiança, que determinam graus de segurança e privilégios e são implementados por mecanismos de controle de acesso. Terceiro, o isolamento de aplicações e do Sistema Operacional, que se refere à habilidade de uma plataforma móvel em isolar ou conter uma aplicação em particular, como uma estratégia de prevenção contra o comprometimento de outras aplicações ou o próprio S.O.

O modelo de segurança adotado pela norma ABNT NBR 15605-2 não é fundamentalmente diferente daquele adotado pelos fabricantes de dispositivos móveis, os três pilares citados anteriormente. Não por acaso, a norma brasileira de segurança de aplicações interativas para TV digital oferece uma arquitetura de segurança semelhante em diversos aspectos à arquitetura de segurança das plataformas móveis modernas.

De modo análogo ao oferecido pelas plataformas de aplicativos para dispositivos móveis, um receptor interativo, com *middleware* Ginga, aderente a norma brasileira, poderá possuir no mínimo as seguintes características de segurança: autenticação de usuário, segurança no canal de interatividade com SSL/TLS e suporte à autenticação de aplicativos com o mecanismo de controle de acesso correspondente.

Em relação à segurança na distribuição das aplicações. A norma ABNT NBR 15605-2 estabelece que as aplicações disponíveis pelo carrossel sejam autenticadas. A autenticação de aplicações será feita com assinaturas digitais, onde a entidade responsável pela assinatura da aplicação possuiria um certificado de identidade ICP-Brasil e o próprio processo de assinatura seria semelhante à assinatura de código móvel da plataforma Java.

O mecanismo de controle de acesso defendido pela norma tem semelhanças com o modelo mandatório. A autonomia de uma aplicação interativa, assinada ou não, sempre será limitada às restrições de acesso do receptor de TV digital. Uma aplicação não assinada ou não autenticada nunca poderá solicitar a modificação das restrições impostas pelo receptor para ampliar sua autonomia. Já uma aplicação assinada e autenticada poderia solicitar permissões extras, mas ainda assim estaria limitada a no máximo o conteúdo de um arquivo de requisição de permissões (certificado de atributos),emitido e assinado por entidade reconhecida.

A implementação do mecanismo de isolamento de aplicações seria semelhante ao *sand-box* utilizado em outras tecnologias de código móvel. Esta implementação pode ser bastante flexível. Por exemplo, o *sand-box* poderia até dispensar o uso de autenticação para aplicativos considerados inofensivos, atribuindo a estes uma autonomia mínima, de acordo com a política de segurança defendida pela norma ABNT NBR 15605-2.

E quanto ao sistema embarcado de hardware restrito, o receptor de TVDi apresenta uma série de desafios para segurança da informações. A Figura 7 ilustra, em um diagrama de blocos, as camadas de software do receptor de TVDI e suas necessidades de segurança de aplicações e de dados manipulados por elas, nos ambientes de multi-aplicações e de multi-serviços: (1) Isolamento de aplicações, proporcionado pelo *middleware*; (2)

comunicação segura de informações em trânsito; (3) armazenamento seguro de dados e programas; (4) biblioteca criptográfica plenamente funcional; (5) placa aceleradora criptográfica, independente do hardware restrito dedicado às funcionalidades principais do receptor.

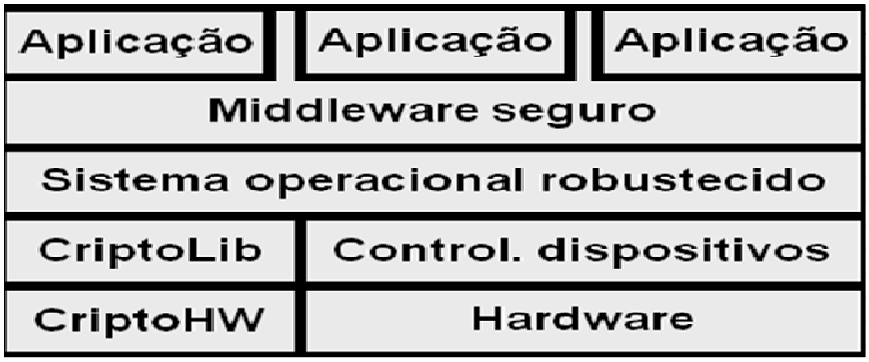

Figura 7 - Camadas de software e hardware para Set-Top Box multi-serviços seguro.

5.4. Vulnerabilidades de programação insegura

Esta seção apresenta o embrião de um trabalho contínuo de catalogação de vulnerabilidades de programação insegura identificadas nas linguagens de programação utilizadas pelo Ginga, em particular, o NCLua. Cada vulnerabilidade catalogada será documentada com uma descrição geral, programas exemplo de exploração e de simulação da vulnerabilidade e estratégias de mitigação.

Foram investigadas bibliotecas, APIs e rotinas perigosas. E foi elaborado um catálogo de vulnerabilidades de programação inseguras identificadas e documentadas em Lua, refletindo um trabalho em progresso.

Foram estudadas diversas fontes de informação sobre Lua. As seguintes podem ser citadas como mais relevantes: Lerusalimschy (2003), Lerusalimschy, Figueiredo e Celes (2006), Barbosa e Soares (2008), Lerusalimschy (2009), NCLua Tutorial, Comunidade Lua, a norma ABNT NBR 15605, partes 1 e 2, e os pacotes de software LuaCrypto e LuaSec.

Os programas exemplo e vulnerabilidades encontradas foram testados na distribuição Lua for Windows, no simulador Ginga-NCL Virtual STB e em receptores TVDi de mercado com Ginga-NCL e Lua embarcados. Cada vulnerabilidade identificada e documentada é acompanhada de uma estratégia de mitigação. Recomendações mais amplas sobre segurança de sistemas são apresentadas adiante no texto.

5.4.1. Rotinas, bibliotecas e APIs perigosas

A norma ABNT NBR 15606-2 (2007), na seção 10 - Objetos procedurais Lua em apresentações NCL, Anexo B, estabelece o uso de pacotes Ginga-J em NCLua e define as rotinas perigosas a seguir como opcionais. O *package loadlib*. Todas as funções do pacote IO. No pacote OS, as rotinas *clock, execute, exit, getenv, remove,*

rename, tmpname e *setlocale*. Todas as funções do pacote debug. Elas podem não estar disponíveis em NCLua, pois são dependentes do sistema operacional.

As investigações realizadas durante este trabalho identificaram diversas APIs opcionais disponíveis de forma perigosa. Esta lista apresentada na Tabela 1 não é exaustiva. De fato, trata-se de um trabalho em progresso.

Tabela 1: funções com potencial para exposição de vulnerabilidades.

Categoria	Rotina	Ameaça
Pacote OS	os.remove() e os.rename()	Estas rotinas proporcionam acesso e modificação do sistema de arquivos.
	os.exit() os.getenv()	Controle e acesso a informações de programas. Por exemplo, terminação da execução de um programa ou obtenção de ambiente.
	os.date()	Defeito documentado. Na versão 5.1, falha e termina a execução do programa, em algumas plataformas.
	os.execute()	Esta rotina pode ser usada na injeção de comandos arbitrários, repassados ao sistema operacional.
Carga /execução de código	loadfile() dofile()	Estas rotinas podem ser usadas na injeção de código malicioso via arquivos.
	loadstring	Esta rotina pode ser usada na injeção de códigos maliciosos via strings.
Pacote IO	io.write() io.flush() io.read()	Por se tratar de API para acesso a arquivos, é comumente associada a ataques de negação de serviço. Pode ser usada de modo negligente e consumir todo o disco/armazenamento. Também é associada a condições de competição (race conditions). Ainda, a rotina de leitura de caracteres pelo teclado pode ser usada sem verificação dos dados de entrada.

No receptor de TVD, o pacote IO não seria usado para entrada de dados pelo console (teclado + display), pois naquele ambiente vale a integração Lua e NCL (NCLua). Mas ainda assim seria usado para manipulação de arquivos.

5.4.2. Vulnerabilidades de programação Lua

As vulnerabilidades identificadas nesta seção são descritas de acordo com o seguinte formato: uma descrição geral, como explorar a vulnerabilidade, programas exemplo em Lua e estratégias de mitigação da vulnerabilidade. Além disso, é feita uma classificação da vulnerabilidade Lua, com base na vulnerabilidade análoga em dois catálogos conhecidos: SANS/CWE Top 25 (2010) e OWASP Top 10 (2010), quando aplicável.

5.4.2.1. Injeção de comandos

Esta vulnerabilidade pode se manifestar quando o software constrói totalmente ou parcialmente um comando do sistema operacional usando dados de entrada repassados diretamente a outro componente, o qual executa o comando. A rotina de entrada de dados não saneia ou saneia incorretamente caracteres especiais, capazes de modificar o significado dos dados quando enviados a outro componente.

Nesta implementação da vulnerabilidade, dois pontos devem ser observados. Primeiro, a aplicação aceita uma entrada de dados, cujo uso esperado é desviado para injetar o nome de um programa a executar. A aplicação simplesmente redireciona o comando inteiro para o sistema operacional. Segundo é suposto que o programador não teve a intenção de expor a execução de comandos a qualquer pessoa não confiável, mas o programador provavelmente não identificou as formas alternativas que os atacantes mal intencionados manipulam a aplicação.

A vulnerabilidade de injeção de comandos em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Vulnerabilidade A1, *Injection*, de OWASP Top 10 (2010), e
- Erro de programação número 9 (CWE-78), *Failure to Preserve OS Command Structure – OS Command Injection*, de SANS/CWE Top 25 (2010).

5.4.2.1.1. Exploração da vulnerabilidade

Se um atacante pode executar códigos arbitrários no receptor de TVDi, então isto pode significar que o equipamento está seriamente comprometido. O atacante pode ser capaz de assumir o controle sobre o programa em execução ou potencialmente romper os limites do processo em execução para abrir um novo shell no computador.

A vulnerabilidade é exposta quando um programa usa a rotina loadstring() para executar um trecho de código Lua construído a partir de material digitado pelo usuário. O código exemplo, mostrado no Programa 1, é o de uma calculadora simples que lê uma expressão aritmética e a resolve, apresentando o resultado. O código usa a facilidade Lua de resolver expressões aritméticas e executar seqüências de caracteres como comandos da linguagem.

```
01   i = 0;
02   f = "i= "..io.read();
03   g = assert(loadstring(f));
04   g();
05   print(i);
```

Programa 1: Calculadora de expressões aritméticas simples.

A calculadora pode ser abusada do seguinte modo. Quando o programa solicita a entrada de dados, digita-se a seqüência maliciosa de caracteres "`1; << código injetado>>;`". Neste trecho de código, o "1;" fecha a string da expressão aritmética.

O sinal de ponto e vírgula ao final do código injetado encerra o comando e evita concatenações defeituosas. A seguir, três explorações da calculadora.

Exploração 1. Injetando um código simples. Basta digitar a seqüência de caracteres a seguir quando o programa estiver esperando a expressão aritmética.

```
"1 ; print('oi');"
```

O resultado é a impressão de "oi" antes da impressão do resultado da expressão aritmética, "1".

Exploração 2. Injetando comando de sistema operacional. Basta digitar a seqüência de caracteres a seguir quando o programa estiver esperando a expressão aritmética.

```
"1 ; os.execute('dir');"
```

O resultado, em Windows, é a impressão da listagem dos arquivos, contidos na pasta corrente, antes da impressão do resultado da expressão aritmética, "1".

Exploração 3. Injeção de código Lua. Basta digitar a seqüência de caracteres a seguir quando o programa estiver esperando a expressão aritmética

```
"1; dofile('do.lua');
```

O resultado é a execução do arquivo do.lua, mostrado no Programa 2, onde é impressa a string "Código injetado", antes da impressão do resultado da expressão aritmética.

```
01   Print("Código injetado")
```

Programa 2: Código injetado a partir de arquivo.

5.4.2.1.2. Estratégia de mitigação

A estratégia de mitigação mais simples é evitar o uso das funções consideradas perigosas: loadstring(), dofile(), os.execute(). Se isto não for possível, o saneamento de dados de entrada, para evitar a injeção de comandos, oferece uma solução parcial. Algumas recomendações preventivas genéricas são as seguintes:

- Executar a aplicação com o mínimo de privilégios necessários;
- Evitar passar a entrada do usuário diretamente para os comandos que avaliam código (em Lua, loadstring, dofile, os.execute). Em vez disso, usar a entrada do usuário para selecionar uma ação a partir de conjunto de pré-definido de opções de comandos.
- Aplicar os controles que limitem a ação de código em linguagens consideradas inseguras. No caso deste exemplo, conter ou evitar os scripts do sistema operacional. O sand-box de código também se aplica a esta situação.

5.4.2.2. Condição de competição

Uma *Race Condition* (traduzida aqui como Condição de Competição) é a situação na quais programas em execução concorrem pelo uso simultâneo de um recurso de computação. A Condição de Competição pode ocorrer quando uma seqüência de operações, dependente do tempo, é realizada simultaneamente em várias *threads* (linha de execução) ou ambiente de multiprocessamento. Neste contexto, as instruções de uma

thread podem ser suspensas para dar vez à outra thread. Enquanto uma thread está suspensa, o ambiente do qual a thread depende pode ser alterado de forma inesperada. A causa desta vulnerabilidade é a suposição implícita e incorreta de que a seqüência de operações em questão é atômica. Uma solução possível é a de isolar o código mais relevante em um ponto crítico que pode ser tratado como logicamente atômico. As condições de competição que têm implicações de segurança são aquelas relacionadas com acessos ao sistema de arquivo. Tipicamente, a verificação do direito de acesso antes do acesso propriamente. Do ingles *Time-of-Check Time-Of-Use*, ou ToCToU.

A vulnerabilidade de condição de competição em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Não se aplica ao OWASP Top 10 (2010), e
- Erro de programação número 25 (CWE-362), *Race Condition*, de SANS/CWE Top 25 (2010).

5.4.2.2.1. Exploração da Vulnerabilidade

Corrotinas de Lua, similares a *threads* em outras linguagens de programação, podem ser construídas de modo a competirem por recursos, causando uma situação de *deadlock*. Um exemplo simples do conceito não precisa usar *threads*, ou corrotinas.

Esta seção mostra um exemplo de como fazer uma condição de competição em Lua. O exemplo explora ToCToU sobre arquivos. Desdobramento interessante do conceito bastante simples de condição de competição em outras linguagens de programação (C/C++/Java). O cenário é o seguinte: enquanto o programa executa, o testador muda a permissão do arquivo acessado pelo programa.

Nesta exploração, duas corrotinas diferentes de um mesmo programa estão competindo pelo arquivo. Cada corrotina abre e fecha o arquivo. Mas neste caso, quando a corrotina toctou2 é executada no meio da corrotina toctou1, a corrotina toctou1 perde a referência ao arquivo.

```
01   function toctou1 ()
02   -- Time of Check (ToC)
03      f,m = assert(io.open(".\\teste.txt","w"));
04      print(f,m);
05      print(io.type(f))
06      coroutine.yield()
07
08   -- Time of Use (ToU)
09      m = assert(f:write("\n Corrotina 1 "..os.time()))
10      print(m);
11      f:close();
12   end;
13
14   function toctou2 ()
15   -- Time of Check (ToC)
16      f,m = assert(io.open(".\\teste.txt","w"));
17      print(f,m);
18      print(io.type(f))
19
20   -- Time of Use (ToU)
21      m = assert(f:write("\n Corrotina 2 "..os.time()))
22      print(m);
23      f:close();
24   end;
25
26   co1 = coroutine.create(toctou1)
27   co2 = coroutine.create(toctou2)
28   coroutine.resume(co1)
29   coroutine.resume(co2)
30   coroutine.resume(co1)
```

Programa 3: toctou1 abre o arquivo e sede a vez para toctou2 abrir e gravar dados no mesmo arquivo, antes de toctou1.

5.4.2.2.2. Estratégia de mitigação

A mitigação dos dois primeiros casos de exploração da condição de competição está fora do escopo deste texto, pois depende de fatores externos aos programas Lua envolvidos.

O terceiro caso de exploração pode ser mitigado se a referência ao arquivo for obtida e mantida por código fora das corrotinas. No Programa 4, as corrotinas não estão competindo pelo arquivo. O acesso é mantido pelo programa principal. Primeiro a função toctou2 escreve e depois a função toctou1 escreve.

```
01   f,m = assert(io.open(".\\teste.txt","w"));
02   print(f,m);
03
04   function toctou1 ()
05   -- Time of Check (ToC)
06      print(io.type(f))
07      coroutine.yield()
08   -- Time of Use (ToU)
09      m = assert(f:write("\n Corrotina 1 "..os.time()))
10      print(m);
```

```
11   end;
12
13   function toctou2 ()
14   -- Time of Check (ToC)
15      print(io.type(f))
16   -- Time of Use (ToU)
17      m = assert(f:write("\n Corrotina 2 "..os.time()))
18      print(m);
19   end;
20
21   co1 = coroutine.create(toctou1)
22   co2 = coroutine.create(toctou2)
23   coroutine.resume(co1)
24   coroutine.resume(co2)
25   coroutine.resume(co1)
27   f:close();
```

Programa 4: Competição eliminada pela gerência externa do recurso compartilhado.

5.4.2.3. Integridade de arquivos e código malicioso

Em Lua, tal e qual descrito no capítulo 12 de Lerusalimschy (2003), arquivos de dados podem ser lidos como programas, se formatados para tal, de modo que as estruturas de dados são carregadas diretamente e com facilidade. Este é um mecanismo extremamente simples de persistência e de recuperação de dados persistidos.

Em contrapartida, a facilidade de implementação sacrifica a segurança. Dois aspectos devem ser observados como causas raiz das vulnerabilidades. Primeira, o ambiente de execução Lua não verifica a integridade e nem a autenticidade de arquivos carregados em tempo de execução (com as funções dofile() e loadfile()). Segundo, visto que não haverá diferença entre dados e programas, códigos maliciosos podem ser embutidos nos arquivos de dados. Estes códigos serão executados quando a estrutura de dados for interpretada.

A vulnerabilidade de Integridade de arquivos e código malicioso em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Vulnerabilidade A1, Injection, de OWASP Top 10 (2010), e
- Erro de programação número 20 (CWE-494), Download of Code Without Integrity Check, de SANS/CWE Top 25 (2010).

5.4.2.3.1. Exploração da vulnerabilidade

Usando a estratégia de persistência de dados ilustrada no Programa 5 e Programa 6, para explorar a vulnerabilidade, basta acrescentar o código que se deseja executar ao arquivo de dados. Isto preserva o comportamento aparente esperado do programa e acrescenta o comportamento malicioso.

No Programa 5, o comando os.execute("Dir ..") foi acrescentado ao final do arquivo de dados. O código malicioso é carregado pelo Programa 6 e executado e antes da saída esperada do programa original.

Outra possibilidade é a substituição completa do arquivo de dados por outro arquivo Lua contendo apenas código malicioso. Neste caso, o comportamento aparente do programa original não seria preservado.

```
01   Registro { nome = "Alexandre", }
02   Registro { nome = "Alex",}
03   -- este eh comando injetado que corrompe o arquivo de dados
04   os.execute("dir ..");
```

Programa 5: Arquivo de dados codificado como programa Lua.

```
01   regs = {}
02   function Registro (r)
03      if r.nome then regs[r.nome] = true end
04   end
05   dofile("Registro.lua")
06   for nome in pairs (regs) do print(nome) end
```

Programa 6: Programa Lua para carregar e imprimir dados de um arquivo.

O Programa 7 mostra uma estratégia simples para geração do arquivo de dados. O arquivo de saída contém os registros mostrados no Programa 5.

```
01   function serialize (o)
02        if type(o) == "string" then
03        io.write('Registro { nome = "', o, '" } \n') end
04   end
05
06   io.output(".\\Registro.lua")
07
08   serialize("Alexandre");
09   serialize("Alex");
10
11   io.close();
```

Programa 7: Estratégia simples para persistência de dados.

5.4.2.3.2. Estratégia de mitigação

A estratégia de mitigação é a utilização da técnica de assinatura de código, baseada em criptografia de assinaturas digitais, capaz de garantir não somente a integridade da aplicação e seus arquivos constituintes, mas também a autenticação de origem, ou autoria da aplicação. Esta técnica tem sido usada com sucesso em diversas implementações de código móvel, tais como o Java e o .NET, assim como no Android.

Além disso, algumas medidas paliativas, descritas a seguir, podem ser adotadas. Programas ou dados sensíveis que não são persistidos não têm os problemas detalhados nesta seção. Persistir dados efetivamente cria um acesso público para os dados.

Sobre a persistência de tabelas em arquivos Lua. Depois que uma tabela foi persistida, qualquer controle de segurança da linguagem Lua não poderá mais ser executado, os atacantes poderão acessar campos particulares em uma tabela analisando o dado diretamente. Portanto, é uma boa prática não armazenar dados confidenciais em uma tabela gravada em arquivos.

Sobre a recuperação ou carga de tabelas armazenadas em arquivos. Recuperação cria uma nova instância de uma tabela sem invocar a função construtora convencional, caso exista. Duas medidas de segurança simples evitarão vulnerabilidades. Primeira, realizar, na recuperação, as verificações de validação de entrada do mesmo modo que realizado na entrada dos dados ou em função construtora. Segundo, atribuir valores padrão consistentes com aqueles atribuídos em uma função construtora, para os dados não preservados quando da persistência da tabela.

5.4.2.4. Script cruzado armazenado

Esta vulnerabilidade é semelhante ao tradicional *Cross Site Scripting* (XSS), uma das vulnerabilidades mais perigosas das aplicações web. A vulnerabilidade se manifesta quando o software não valida, filtra e nem decodifica a entrada de dados do usuário, e nem a codifica antes de colocá-la na saída de dados apresentados para outros usuários.

Esta manifestação da vulnerabilidade segue a linha de atuação do XSS armazenado. O aplicativo armazena dados em arquivo (mas poderia ter sido usada outro armazenamento). O atacante mal intencionado manipula a aplicação para que um script executável seja armazenado junto com os dados. Mais tarde, o dado perigoso (código malicioso) é posteriormente lido de volta para a aplicação e incluídos no conteúdo dinâmico, vitimando outro usuário.

A vulnerabilidade de script cruzado armazenado em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Vulnerabilidade A2, *Cross-site scripting*, de OWASP Top 10 (2010), e
- Erro de programação número 1 (CWE-79), *Failure to Preserve Web Page Structure – Cross-site Scripting*, de SANS/CWE Top 25 (2010).

5.4.2.4.1. Exploração da Vulnerabilidade

O exemplo clássico de vulnerabilidade XSS é a injeção de um script simples que quando refletido é executado. Neste exemplo, o script cruzado armazenado é exemplificado com a exploração da codificação de strings em Lua.

Exploração 1. O Programa 8 lê dados de entrada e, com uma estratégia simples de serialização, os salva no arquivo Programa 10. Os colchetes duplos usados como delimitadores de string pelo Programa 8 permite a inclusão de caracteres especiais.

O Programa 9 recupera o dado armazenado, o qual pode conter o script injetado, e o apresenta ao usuário. Os seguintes códigos script podem ser injetados durante a leitura de dados pelo Programa 8 e serão armazenados e refletidos na resposta ao usuário.

A seqüência de caracteres a seguir fecha uma string em aberto, injeta o script malicioso e reconstitui a atribuição de variável. O Programa 10 mostra o resultado.

```
]] ; print("Olá") ; aaaaa = [[
```

A seqüência a seguir fecha uma string em aberto, injeta o script malicioso e transforma em comentário tudo o que estiver após o trecho injetado.

```
]] ; print("olá") --
```

```
01   function serialize (o)
02      if type(o) == "string" then
03         io.write("variavel = [[", o, "]]")
04      end
05   end
06   io.output(".\\serial.lua")
07   o = io.read();
08   serialize("isto é um texto de teste "..o);
09   io.close();
```

Programa 8: Usando strings delimitadas por colchetes duplos.

```
01    dofile(".\\serial.lua");
02    print("\n")
03    print(variável)
```

Programa 9: Reconstituição da variável a partir de string armazenada.

```
01    variavel = [[isto é um texto de teste ]] ; print("Olá") ;
02   aaaaa = [[]]
```

Programa 10: variável armazenada, violada por código malicioso e colchetes duplos.

Exploração 2. O Programa 11 ilustra uma estratégia de serialização diferente da anterior. A string persistida será delimitada por aspas simples. Em termos práticos, o resultado é idêntico ao obtido pelo uso de aspas duplas. O método de recuperação é o mesmo da exploração anterior, o Programa 9. O Programa 12 mostra o arquivo gerado com a violação, o código injetado. A seqüência de caracteres a seguir fecha uma string em aberto, injeta o script malicioso e transforma em comentário tudo o que estiver após o trecho injetado.

```
' ; print("olá") --
```

```
01    function serialize (o)
02       if type(o) == "string" then
03          io.write("variavel = '", o, "'")
04       end
05    end
```

Programa 11: Usando strings delimitadas por aspas simples.

```
01    Variável = 'isto é um texto de teste ' ; print("ola") ; --'
```

Programa 12: variável armazenada, violada por código malicioso em aspas simples.

5.4.2.4.2. Estratégia de mitigação

O ataque XSS tradicional sobre HTML possui duas estratégias de mitigação, uma voltada para os dados de entrada e outra voltada para a saída. A validação, ou o saneamento, de dados de entrada evita a entrada de código malicioso. A formatação (ou

codificação) dos dados de saída evita a saída de informação mal formatada ou corruptível. Em Lua, o uso da função string.format com o parâmetro "%q" dificulta a construção de strings defeituosas, pois a string estará em aspas duplas e todo caractere especial estará formatado com "\". Conforme ilustrado no Programa 13. O Programa 14 ilustra o arquivo de dados armazenados quando uma aspas extra é inserida pelo usuário. A string não é corrompida, graças à estratégia de codificação de saída.

```
01   function serialize (o)
02      if type(o) == "string" then
03         io.write(string.format("variavel = %q", o))
04      end
05   end
```

Programa 13: Codificação de strings de saída evita a execução de código injetado.

```
01   variavel = "isto é um texto de teste \""
```

Programa 14: Aspas extra precedida por barra sem corromper a string.

5.4.2.5. Referência insegura a tabelas

Diversas linguagens de programação procedimentais que possuem vetores e acesso a dados por referência sofrem de uma dificuldade bastante peculiar: o acesso múltiplo, via diversas referências, a um mesmo dado. Nesta condição, o dado referenciado pode ser modificado por qualquer um dos detentores de referências, sem que os outros detentores tomem conhecimento, em tempo hábil, da modificação.

Em Lua, esta vulnerabilidade se manifesta no acesso a tabelas mutáveis. Isto é, tabelas cujas referências são compartilhadas. Tabelas mutáveis podem ser alteradas depois e até durante a execução de uma função, causando comportamento incorreto. Se uma função não é especificada para operar direta e cautelosamente sobre tabelas mutáveis compartilhadas, é recomendável criar uma cópia do dado e executar a lógica do programa apenas na cópia.

De fato, se a tabela é compartilhada por vários, os programas que usam a tabela podem sofrer da vulnerabilidade Condição de Competição, causada por diferenças de conteúdo e inconsistências de *Time-of-Check Time-of-Use* (ToCToU). Por exemplo, ao ser acessada e alterada por corrotinas distintas, a tabela compartilhada conteria um valor mutável durante o momento de verificação do valor, mas um valor diferente durante o uso da tabela.

A vulnerabilidade de acesso por referência a elementos de tabelas em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Vulnerabilidade A4, *Insecure Direct Object References*, de OWASP Top 10 (2010), e
- Erro de programação número 25 (CWE-362), *Race Condition*, de SANS/CWE Top 25 (2010).

5.4.2.5.1. Exploração da vulnerabilidade

Esta seção descreve uma exploração da vulnerabilidade de referência insegura a tabelas com acesso múltiplo a dados mutáveis. Todos os programas desta seção fazem uso do Programa 15, para a impressão de tabelas.

```
01    Function printtable (o)
02         if type(o) == "table" then
03            io.write("{")
04            for k,v in pairs(o) do
05              io.write("  "..k .." = ")
06              printtable(v)
07              io.write(",")
08            end
09            print("}\n")
10         end
11         if type(o) == "string" then
12         io.write(string.format("%q", o))
13         end
14         if o == nil then io.write("nil \n \n") end
15   end
```

Programa 15: rotina de apoio para impressão de tabelas.

A exploração da vulnerabilidade é construída sobre o conceito de *proxies* de tabelas, com *metatables* e *metamethods* de Lua. O Programa 16 contém o código de um proxy de acesso, um monitor de referências, que serve de intermediário ou mediador em todo acesso ao conteúdo de uma dada tabela original.

O monitor de referências, MR, possui duas funções principais. Primeira, a função MR.monitor(t), responsável pela construção da *metatable*, o proxy propriamente dito. Segunda, a função MR.interno() que retorna a tabela original.

O aspecto de implementação vulnerável deste monitor é o fato dele manter uma referência interna, não uma cópia ou instância única, da tabela mediada, a variável MR._t, e externá-la, espalhando descontroladamente as referências.

```
01    MR = { _t = nil}
02
03   function MR.monitor(t)
04               MR._t = t
05         local proxy = {}
06         local    mt = {
07         __index    = function (t,k) return MR._t[k] end,
08         __newindex = function (t,k,v) MR._t[k]="MR: ".. v end
09         }
10         setmetatable(proxy, mt)
11         return proxy
12   end
13
14   function MR.interno() return MR._t end
```

Programa 16: Monitor de referências a tabelas.

O Programa 17 mostra a utilização do monitor de referências. A tabela t é criada e preenchida. O proxy é criado. A tabela original é modificada. Em seguida a referência à tabela original é anulada. Uma alteração é feita via proxy. O estado interno do proxy é recuperado na variável t original. A saída do programa é mostrada no quadro Output 1.

```
01   T = {}
02   for i=0, 4 do t[i] = ""..i end
03
04   dofile('printtable.lua')
05   dofile('monitorreferencia.lua')
06
07   proxy = MR.monitor(t) -- cria proxy
08
09   printtable(t) -- print tabela original
10
11   t[1] = "A"    -- modifica tabela original
12   printtable(t) -- print tabela modificada
13
14   t = nil       -- perde referência original
15   printtable(t) -- print tabela original
16
17   proxy[2] = "A" -- modifica via proxy
18
19   t = MR.interno()
20   printable(t)
```

Programa 17: Manipulação de referências.

```
01   {  1 = "1",  2 = "2",  3 = "3",  4 = "4",  0 = "0",}
02   {  1 = "A",  2 = "2",  3 = "3",  4 = "4",  0 = "0",}
03   nil
04   {  1 = "A",  2 = "MR: A",  3 = "3",  4 = "4",  0 = "0",}
```

Output 1: Saída do programa de manipulação de referências.

5.4.2.5.2. Estratégia de mitigação

A estratégia geral de tratamento desta vulnerabilidade é a eliminação das referências extras e indesejadas às tabelas, pelo uso de cópias, valor a valor, da tabela original.

Uma variação da vulnerabilidade é a exposição do estado interno. Se uma função retorna uma referência a uma tabela interna mutável, em seguida, o código do cliente da função pode modificar o estado interno da tabela. Portanto, é uma boa prática de programação segura fazer cópias de tabelas mutáveis e retornar as cópias, a menos que a intenção seja de fato compartilhar o estado interno. O Programa 18 contém uma função de cópia, valor a valor, da tabela para o monitor de referência, MR.clone(). Uma função que retorne o estado interno, sem espalhar referência, usa a rotina de clonagem.

```
01   function MR.clone()
02         local c = {}
03         for k,v in pairs(MR._t) do c[k] = v end
04         return c
```

```
05   end
06
07   function MR.internoSeguro() return MR.clone() end
```

Programa 18: Clonagem da referência interna.

Outra variação da vulnerabilidade é a guarda de referências a tabelas mantidas externamente. Funções que recebem tabelas devem clonar as tabelas e armazenar a cópia para trabalho. Em vez de trabalhar sobre a referência original. Isso impede que as mudanças inesperadas na tabela original, realizadas sobre alguma referência perdida, afetem o funcionamento do programa. Expor tabelas internas diretamente permite alteração de algum código ou dado, sobre a referência original, a qual deveria ser mantida sob o controle interno. É mais seguro retornar uma cópia da tabela interna.

O monitor de cópias, mostrado no Programa 19, é uma solução mais abrangente para a questão das referências extras indesejadas. O monitor de cópias não é mais um proxy para a tabela original, mas sim um intermediário no acesso a uma cópia da tabela original. O Programa 20 usa o monitor de cópias e exercita seu funcionamento.

```
01    MC = { _t = {} }
02
03   function MC.copia(t)
04         for k,v in pairs(t) do MC._t[k] = v end
05   end
06
07   function MC.monitor(t)
08         MC.copia(t)
09
10         local proxy = {}
11         local mt = {
12         __index = function (t,k) return MC._t[k] end,
13
14         __newindex = function (t,k,v) MC._t[k] ="MC: "..v end
15         }
16         setmetatable(proxy, mt)
17         return proxy
18   end
19
20   function MC.interno() return MC._t end
21
```

Programa 19: Monitor de cópias

```
01   t = {}
02   for i=0, 4 do t[i] = ""..i end
03
04   dofile('printtable.lua')
05   dofile('monitorcopia.lua')
06
07   proxy = MC.monitor(t) -- cria proxy
08
09   printtable(t) -- print tabela original
10
```

```
11   t[1] = "A" -- modifica tabela original
12   printtable(t) -- print tabela modificada
13
14   t = nil -- perde referência original
15   printtable(t) -- print tabela original
16
17   proxy[2] = "A" -- modifica via proxy
18
19   t = MC.interno()
20
21   printtable(t)
```

Programa 20: Manipulação de cópias

O funcionamento do Programa 20 é o seguinte. A tabela t é criada e preenchida. O proxy é criado. A tabela original é modificada. Em seguida a referência à tabela original é anulada. Uma alteração é feita via proxy. O estado interno do proxy é recuperado na variável t original. A saída do programa é mostrada no quadro Output 2, onde se vê que os conteúdos de cada tabela são distintos das demais.

```
01   {  1 = "1",  2 = "2",  3 = "3",  4 = "4",  0 = "0",}
02   {  1 = "A",  2 = "2",  3 = "3",  4 = "4",  0 = "0",}
03   nil
04   {  1 = "1",  2 = "MC: A",  3 = "3",  4 = "4",  0 = "0",}
```

Output 2: Saída do programa de manipulação de cópias.

O Programa 21 acrescenta ao monitor de cópias a facilidade de expor cópias de seu estado interno, de modo análogo ao monitor de referências.

```
01   function MC.clone()
02        local c = {}
03        for k,v in pairs(MC._t) do c[k] = v end
04        return c
05   end
06
07   function MC.internoSeguro() return MC.clone() end
```

Programa 21: Exposição de clone do estado interno em monitor de cópia.

O Programa 22 e sua saída em Output 3 ilustram mais uma vez as diferenças entre cópias e referências. No monitor de cópias, as referências às tabelas são impressas para mostrar que a tabela original, o estado interno e a cópia externada são de fato tabelas diferentes. No monitor, as referências são impressas para mostrar que a tabela original e o estado interno do proxy são de fato a mesma tabela e a cópia externada é uma tabela diferente.

```
01   t = {}
02   for i=0, 4 do t[i] = ""..i end
03
04   print("\n clonagem da cópia \n")
05   dofile('printtable.lua')
06   dofile('monitorcopia.lua')
07
```

```
08   print("Origem",t)
09   MC.monitor(t)
10   print("Interno",MC.interno())
11   print("IntSeg",MC.internoSeguro())
12
13   print("\n clonagem da referencia \n")
14   dofile('monitorreferencia.lua')
15
16   print("Origem",t)
17   MR.monitor(t)
18   print("Interno",MR.interno())
19   print("IntSeg",MR.internoSeguro())
```

Programa 22: Programa de análise de clones.

```
01   Clonagem da cópia
02
03   Origemtable: 0x6b7ac0
04   Internotable: 0x6a3280
05   IntSegtable: 0x6a71f8
06
07   clonagem da referencia
08
09   Origemtable: 0x6b7ac0
10   Internotable: 0x6b7ac0
11   IntSegtable: 0x6a79f8
```

Output 3: Saída do programa de análise de clones.

5.4.2.6. Injeção de SQL

As aplicações interativas instaladas no receptor, quando bem construídas, provavelmente seguirão o modelo de arquitetura em três camadas, no qual apenas a camada de apresentação (escrita em Lua) estaria no receptor e acessaria serviços web, via rede IP, para consulta a bases de dados remotas. Porém, sempre há a possibilidade de que um SGBD (Sistema de Gerenciamento de Base de Dados) simples seja portado para um receptor e seja localmente acessado por programas Lua, via a biblioteca LuaSQL.

LuaSQL (www.keplerproject.org/luasql), uma interface ODBC para acesso de programas Lua à vários SGBDs, seria uma escolha possível para viabilizar o acesso de programas Lua a bases de dados.

Porém, LuaSQL não possui sentenças pré-definidas ou pré-compiladas, uma característica comum a pacotes semelhantes disponíveis em outras plataformas de software, tais como Java e dotNET.

A vulnerabilidade de Injeção de SQL em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Vulnerabilidade A1, *Injection*, de OWASP Top 10 (2010), e
- Erro de programação número 2 (CWE-89), *Failure to Preserve SQL Query Structure – SQL Injection*, de SANS/CWE Top 25 (2010).

5.4.2.6.1. Exploração da vulnerabilidade

A ausência de sentenças SQL pré-definidas tornaria os programas Lua, com LuaSQL, mais vulneráveis aos ataques de injeção de SQL. Isto se deve ao fato da API LuaSQL só aceitar sentenças SQL como strings. O Programa 23 ilustra o uso da API LuaSQL para execução de sentenças SQL. A primeira sentença executada é uma string simples. A segunda sentença executada é formatada (string.format) e parametrizada.

```
01   cur = assert (con:execute"SELECT name, email from people")
02   -- ou
03   res = assert (con:execute(string.format([[
04       INSERT INTO people
05       VALUES ('%s', '%s')]], name, email)
06     ))
```

Programa 23: Trecho de programa ilustrando o uso do pacote LuaSQL.

A construção e a formatação de strings em Lua podem ser manipuladas para a injeção de comandos SQL. Conforme exemplificado a seguir pelo Programa 24, que constrói três sentenças SQL para *login* com busca em uma tabela de usuários. Este exemplo é uma variação do caso mais simples de injeção SQL encontrado na literatura. A primeira sentença (SQL1) é construída por concatenação simples de strings. A segunda sentença (SQL2) usa *string.format* e parametriza *login* e senha como strings simples (%s). A terceira sentença (SQL3) também usa *string.format*, mas parametriza *login* e senha como strings cotadas e com tratamento dos caracteres de escape (%q). As sentenças com SQL injetado são mostradas no Output 4.

```
01   login  = '" or 1=1 --'
02   passwd = '1234'
03
04   sql1 = 'SELECT * FROM USERS WHERE LOGIN = \"'..
05   login ..'\" AND PASSWD = \"' .. passwd ..'\"'
06
07   sql2 = string.format(
08   [[SELECT * FROM USERS WHERE LOGIN = "%s" AND PASSWD =
09   "%s"]],
10   login, passwd)
11
12   login  = '\" or 1=1 --'
13
14   sql3 = string.format(
15   [[SELECT * FROM USERS WHERE LOGIN = %q AND PASSWD = %q]],
16   login, passwd)
17
18   print("SQL1 = " .. sql1)
19   print("SQL2 = " .. sql2)
     print("SQL3 = " .. sql3)
```

Programa 24: A construção de três sentenças SQL de forma maliciosa.

No Output 4, a sentença SQL1, gerada por concatenação simples de strings, foi corrompida com a injeção SQL mais simples, o trecho `" or 1=1 --`. A sentença SQL2 foi corrompida do mesmo modo, apesar de utilizar strings formatadas. A

sentença SQL3 pode não ser considerada pelo SGBD alvo uma sentença SQL válida, devido ao "\". Se for válida, então foi corrompida, apesar de utilizar strings formatadas e cotadas.

```
01   SQL1 = SELECT * FROM USERS WHERE LOGIN = "" or 1=1 --" AND
02   PASSWD = "1234"
03
04   SQL2 = SELECT * FROM USERS WHERE LOGIN = "" or 1=1 --" AND
05   PASSWD = "1234"
06
07   SQL3 = SELECT * FROM USERS WHERE LOGIN = "\" or 1=1 --" AND
08   PASSWD = "1234"
```

Output 4: Sentenças SQL mal construídas.

5.4.2.6.2. Estratégia de mitigação

As consultas pré-definidas ou pré-compiladas são uma das linhas de defesas mais comuns contra a injeção de SQL. Uma defesa alternativa e parcial seria o uso de procedimentos armazenados (*stored procedures*), se disponível, no próprio SGBD. Porém é necessário tomar cuidado com procedimentos armazenados, que são parametrizados, mas que ainda podem ser vulneráveis à injeção.

O uso de string.format e %q evita certas construções maliciosas, mas não deve ser usado como única linha de defesa. Ainda é necessário sanear os dados de entrada.

A melhor opção de proteção defensiva é usar uma API segura para acesso ao SGBD que evite o uso direto do interpretador SQL e forneça uma interface parametrizada. A API LuaSQL avaliada neste texto não é segura contra injeção SQL. Logo, existe uma oportunidade de melhoria deste pacote.

5.4.2.7. Mau uso de criptografia

De acordo com Anderson (1993), a maioria das falhas de segurança, devidas ao mau uso de criptografia, são de fato erros de configuração, de implementação e de funções administrativas; como por exemplo, a gestão inadequada de chaves criptográficas, a utilização de configurações vulneráveis de criptografia forte ou o uso de criptografia reconhecidamente fraca.

Assim como ocorre com as outras vulnerabilidades de programação insegura, as vulnerabilidades associadas ao mau uso de criptografia não são detectadas por ferramentas de varredura externas de vulnerabilidades. Ferramentas de análise estática de código fonte podem detectar o uso de APIs criptográficas conhecidas (pelos fabricantes de ferramentas), mas não podem detectar se a API criptográfica está sendo usada adequadamente e de acordo com regras de negócio.

No caso de Lua, a análise manual de programas ainda é a melhor maneira de verificar se uma aplicação cifra adequadamente os dados sensíveis e tem um bom mecanismo e gerenciamento de chaves. Pois as ferramentas de análise estática de programas ainda não tratam Lua. Foram analisadas manualmente as APIs e a documentação (incluindo programas exemplo) de 3 bibliotecas criptográficas em Lua publicamente disponíveis:

- LuaCripto é uma fachada Lua para as funções criptográficas do OpenSSL. Isto pode significar que o LuaCripto não seria mais seguro que o OpenSSL subjacente. Porém ele pode ser mais inseguro, ao fazer uso de partes inseguras e eliminar partes seguras.

LuaSec é um encapsulamento das rotinas OpenSSL para comunicação segura com SSL/TLS. Assim como no caso anterior, isto pode significar que o LuaSec não seria mais seguro que o OpenSSL subjacente. Porém ele pode ser mais inseguro, ao fazer uso de partes inseguras e eliminar partes seguras.

- LuaMD5 é uma biblioteca criptográfica simples, com implementações próprias de algoritmos criptográficos conhecidos.

Este é um trabalho em progresso. O resultado preliminar da análise das APIs das bibliotecas criptográficas é mostrado na próxima seção.

A vulnerabilidade de mau uso de criptografia em Lua tem semelhanças com as seguintes vulnerabilidades classificadas:

- Vulnerabilidade A7, *Insecure Cryptographic Storage*, de OWASP Top 10 (2010).
- Erro de programação número 24 (CWE-327), *Use of a Broken or Risky Cryptographic Algorithm*, de SANS/CWE Top 25 (2010).
- Erro de programação número 10 (CWE-311), *Missing Encryption of Sensitive Data*, de SANS/CWE Top 25 (2010).

5.4.2.7.1. Exploração da vulnerabilidade

De fato, não há nesta seção a exploração da vulnerabilidade, mas o nome da seção é mantido por consistência com as outras vulnerabilidades documentadas. O que há nesta seção é a listagem de características das APIs criptográficas que levariam a vulnerabilidade exploráveis, quando utilizadas em alguma aplicação.

A análise preliminar apresentada neste texto sugere que não há atualmente em Lua uma biblioteca criptográfica completa, plenamente funcional, disponível publicamente. O fato de tanto LuaCripto, quanto de LuaMD5, somente oferecerem acesso a algoritmos de hash ou algoritmos de cifração fracos é um grande limitante da aplicabilidade destas bibliotecas. Além disto, não foi encontrado um receptor de TVDi que oferecesse outra fonte de serviços criptográficos, diferente de LuaCripto e LuaSec.

LuaMD5 usa algoritmos quebrados ou inseguros. Esta biblioteca criptográfica não oferece algoritmos fortes, apenas algoritmos fracos, como o MD5, o DES e até XOR. LuaMD5 favorece o armazenamento inseguro de chaves criptográficas. Diferente de outras APIs criptográficas (JCA, CAPI, OpenSSL), LuaMD5 não oferece mecanismos para armazenamento seguro de chaves, tais como uma KEK (chave de cifração de chaves) ou um repositório seguro para chaves.

LuaCripto usa algoritmos quebrados ou inseguros. Esta biblioteca praticamente só oferece algoritmos considerados inseguros (md5, sha1). LuaCripto favorece o armazenamento inseguro de chaves criptográficas. Diferente de outras APIs criptográficas (JCA, CAPI, OpenSSL) , LuaCripto não oferece mecanismos para

armazenamento seguro de chaves, tais como uma KEK (chave de cifração de chaves) ou um repositório seguro para chaves.

LuaSec é uma fachada bastante flexível para estabelecimento de conexões SSL/TLS com uma instalação do OpenSSL subjacente. Porém LuaSec não oferece acesso às funções criptográficas do OpenSSL. Isto não é uma vulnerabilidade, dado que o objetivo da LuaSec é outro.

5.4.2.7.2. Estratégia de mitigação

Conforme as argumentações de Kocher et al (2004) e Ravi et al (2004), as restrições de hardware impõem limitações severas ao desempenho de algoritmos criptográficos. Os receptores de TVDi disponíveis atualmente são extremamente limitados em memória de trabalho, armazenamento e processamento. Este pode ser um dos motivos da grande dificuldade no oferecimento de serviços de segurança plenos.

A implementação do algoritmo DES, oferecida em LuaMD5, não deve mais ser usada profissionalmente. Implementações do AES ou do 3DES seriam preferíveis se estivessem disponíveis em LuaMD5.

Comparativamente a outras plataformas de software e linguagens de programação (tais como Java e .NET), existe uma lacuna a ser preenchida em Lua. LuaCripto e LuaMD5 poderiam ser estendidas para oferecer mais funções criptográficas e um repositório seguro de chaves. Esta extensão viabilizaria novas aplicações escritas em Lua, com requisitos fortes de sigilo e de autenticação.

Os exemplos de utilização das bibliotecas criptográficas contêm chaves *hardcoded*. Isto pode passar a idéia de que a gestão de chaves é uma tarefa simples. Quando de fato é bastante complicada e uma das principais causas de vulnerabilidades no uso de criptografia. Algumas questões genéricas a se considerar na proteção e na gestão de chaves criptográficas são as seguintes:

- Geração de chaves criptográficas robustas.
- Distribuição segura de chaves criptográficas.
- Armazenamento seguro de chaves criptográficas.
- Alteração periódica das chaves criptográficas.
- Inutilização ou substituição de chaves criptográficas comprometidas, antigas ou suspeitas.
- Prevenção contra a substituição não autorizada de chaves criptográficas.

5.5. Vulnerabilidades de arquitetura, de rede e de sistema operacional

Esta seção mostra, com ferramenta construída para tal, mas em ambiente simulado, como realizar avaliações de segurança em receptores de TVDi com Ginga. A ferramenta em questão dá apoio a um método de testes que utiliza o controle remoto de TV/receptor como dispositivo de entrada de dados e realiza testes de intrusão por injeção de comandos. Além disso, serão apresentados resultados gerais de avaliação de segurança obtidos em laboratório.

5.5.1. Deficiências arquiteturais e de projeto da plataforma Lua

Segundo Hoare (1981), há duas maneiras de construir um software: A primeira é fazê-lo tão simples que obviamente não há deficiências, a outra é fazê-lo tão complicado que não existem deficiências óbvias. O projeto da linguagem de programação Lua buscou a simplicidade extrema. Porém, o atual uso desta linguagem, em sistemas com requisitos de segurança complexos, levanta diversas questões de segurança sistêmica. A fim de realizar requisitos de sistemas mais complexos, muitas vezes são utilizadas construções artificiais que, de fato, aumentam a complexidade a ser tratada pelo programador, aumentando também a suscetibilidade a vulnerabilidades de programação insegura.

As questões de segurança tratadas nesta seção estão mais relacionadas ao projeto seguro, pois as vulnerabilidades não dependem da sintaxe da linguagem ou de bibliotecas predefinidas, mas sim do modo como as construções são projetadas e utilizadas.

5.5.1.1. A técnica de sand-boxing

Na Comunidade Lua (Lua-users.org/wiki/SandBoxes) é descrita uma técnica para a execução de código não confiável em um ambiente de execução Lua restrito. Isto é, um mecanismo de contenção de código. Esta técnica não considera que pode haver níveis de confiança de código. Além disso, não são consideradas mecanismos de promoção de confiança, tais como a assinatura de programas (code signing) e nem proteções contra ataques de negação de serviço por esgotamento de CPU ou de memória.

O método de sand-boxing proposto na referência supracitada é essencialmente programático e de fato muito flexível, pois quase tudo que não está implementado pode ser construído. Porém, com o sacrifício da legibilidade do código e a conseqüente maior suscetibilidade a erros. A principal vantagem deste método é que o sand-box não oferece aos programas acesso ao sistema de arquivos e variáveis globais fora do próprio sand-box. A principal desvantagem é que esta técnica de sand-boxing, por si só, não impõe limites ao consumo de memória e nem de CPU pelos programas maliciosos.

5.5.1.2. Metatables e Metamethods

Em Lerusalimschy (2003), este conceito possui vários usos interessantes para a segurança da informação. A saber, proxies, rastreamento de acessos e tabelas somente de leitura. Este conceito poderia ainda ser ampliado para oferecer mecanismos de segurança mais amigáveis ao programador, tais como o acesso transparente (autorizado) a uma tabela cifrada. Neste exemplo, uma tabela proxy guardaria as chaves criptográficas. Uma característica curiosa é que Lua oferece funções que burlam a suposta proteção oferecida pelas metatables. As rotinas rawget, rawset, e rawequal.

5.5.1.3. Orientação a Objetos

Apesar de um espaço significativo de Lerusalimschy (2003) se dedicado para a orientação a objetos em Lua, a linguagem não oferece qualquer mecanismo explícito (isto é, uma estrutura que faça parte da sintaxe da linguagem) para mecanismos presentes em muitas das linguagens orientadas a objeto na implementação do conceito de *information riding*, a saber: pacotes, controle de acesso (visibilidade variável) a atributos e a métodos.

Estas características de Orientação a Objetos ausentes na linguagem Lua podem ser obtidas a partir de construções lógicas e disciplina de programação. Isto é perigoso, pois tudo o que depende da disciplina do programador é geralmente negligenciado quando o prazo, orçamento ou qualidade precisam ser comprometidos. Sendo uma fonte comum de vulnerabilidades.

As questões de privacidade de dados, tanto em pacotes quanto em objetos são artificiais à linguagem e ilustram bem a suscetibilidade a erros. Por exemplo, uma vez que todas as variáveis são globais por default, uma variável local deve ser explicitamente declarada. Visto que a ausência da palavra reservada "local" não representa um erro de sintaxe, ela pode ser facilmente esquecida pelo programador, sem que o lapso seja detectado de forma automática no receptor da TV digital

Segundo Piccolo 2005, o receptor de TV digital terrestre é também denominado *set-top-box*, e é conhecido também no mercado como URD (Unidade Receptora e Decodificadora). Este equipamento é o responsável pela recepção do sinal digital recebido para que este sinal possa ser visualizado no aparelho de TV analógico convencional. Basicamente ele trata os sinais digitais recebidos por radiodifusão terrestre, cabo, ou satélite e os converte para o formato analógico. Possui como funcionalidades básicas segundo Nicholls: demultiplexar o sinal digital recebido; decodificar informações de áudio e vídeo; processar os dados recebidos e, se for o caso, sincronizá-los com a programação; enviar dados via canal de retorno; construir a imagem a ser exibida no aparelho de TV e a conversão desta para o sinal analógico.

A fonte do sinal pode ser recebida pelo receptor via interfaces distintas que o mesmo apresenta: porta ethernet (serviços *triple play*), cabo coaxial, antena VHF/UHF, ou ainda via modem 3G pela porta USB que o mesmo possui. O canal de retorno permite ao usuário enviar dados relativos a um programa ou emissora ou provedor de serviço via controle remoto. Para que isto seja possível, pode ser utilizado como canal de retorno, telefone fixo (serviços XDSL), modem 3G ou ainda PLC (*Power Line Communication*)

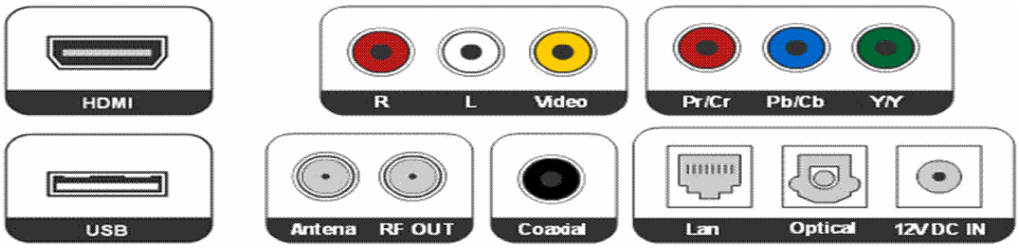

Figura 8 – Interfaces Presentes no Receptor

5. 5.2. Metodologia

A metodologia utilizada pelos autores deste minicurso está embasada nas boas práticas de diversas metodologias dentre as quais: o PMBOK – Project Managment Body of Knowledge, ISSAF – *Information System Security Assessment Framework*, e a

OSSTMM - *Open Source Security Testing Methodology Manual* (Livro Professional *Penetation Testing*), e o *OWASP Testing Guide* v3 somado a contribuições especificas e as adaptações que o cenário de coleta às informações do receptor de TV digital propiciou ao desenvolvimento deste trabalho.

O método foi estruturado em três módulos, a saber: a Injeção de comandos, a Execução dos comandos injetados, e a obtenção do resultado na Coleta dos comandos injetados. Levando em consideração as camadas no sentido do topo para baixo (top/down) do receptor digital: aplicação, *middleware*, software e hardware, como demonstrado na

Figura 9. Neste, sentido foi utilizado o ciclo PDCA (Plan-Do-Check-Act / Planejar, Fazer, Verificar e Agir) Deming, W. Edwards (1986). para analisar as vulnerabilidades encontradas. Foram utilizadas ainda as duas fases do *ISSAF*: Planejamento e Preparação – fase correspondente ao planejamento e elaboração dos testes, e a Avaliação – fase correspondente aos passos para a execução do teste de penetração em camadas. E por fim, o uso da metodologia OSSTMM, utilizada para auditoria de segurança, harmonizando os padrões e normas já existentes.

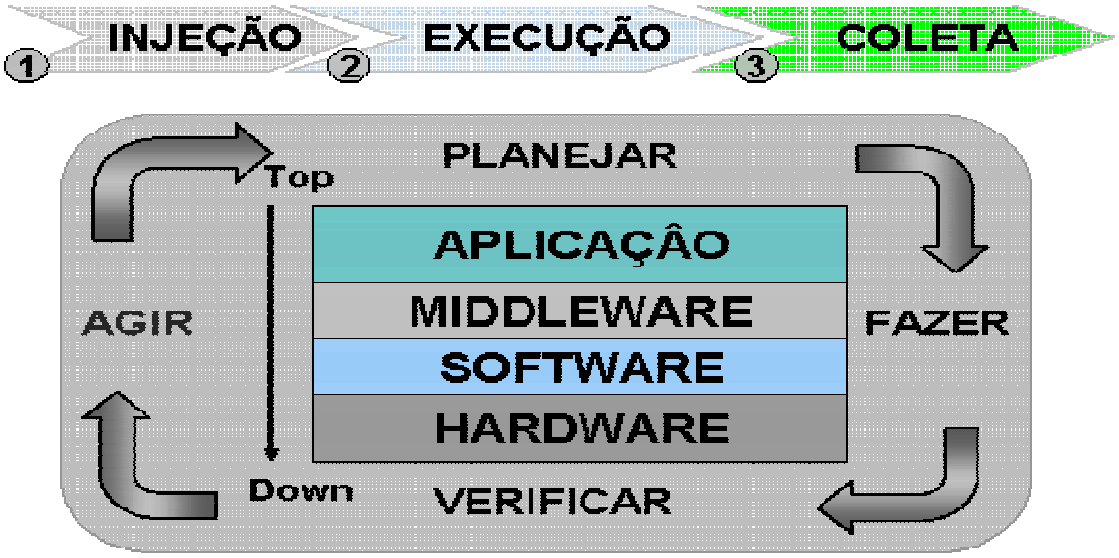

***Figura* 9 - Método de Analise de Vulnerabilidades no Receptor de TV digital interativaObtendo Informações do Receptor de TV digital.**

Segundo Wilhelm, a obtenção de informações é o primeiro passo para conduzir um teste de penetração. Esta ação pode ser segregada em dois tipos diferentes – passivo e ativo. Obter informação de modo passivo permite ter mais informações sobre a rede e os sistemas que estão conectados diretamente ao dispositivo, no caso deste trabalho, o receptor de TV digital. Mais a frente será demonstrado os resultados encontrados utilizando ferramentas de varreduras de rede e a ferramenta desenvolvida pelo CPqD , o AIC – Aplicativo de Injeção de Comandos, para injeção de comandos no receptor. O

segundo tipo para se obter informação é o ativo, ou seja, é possível a conexão a outros receptores alvos. Neste trabalho não foi utilizado este ultimo tipo.

5.5.3. Obtendo Informação Passiva

Foram realizados dois tipos de testes nos receptores. No primeiro foram realizadas varreduras simples e não destrutivas com as seguintes ferramentas comuns em redes de computadores: Nessus (versão livre/gratuita), Nmap/ZenMap (livre/gratuito) e WireShark (livre/gratuito). Para este teste buscou-se respostas para a seguinte questão: Há nos receptores TVDi de mercado vulnerabilidades comuns aos computadores em rede?

A primeira varredura foi executada na interface de rede existente (porta ethernet) no receptor. A preocupação era a obtenção da identificação dos receptores na rede (determinação de endereços IP). A segunda foi à realização de varredura simples com perfil básico das ferramentas, e a terceira foi o monitoramento da comunicação realizada por aplicação interativa a partir do receptor via rede.

Os resultados foram os seguintes. Foi possível capturar dados HTTP em texto claro enviados e recebidos por aplicação interativa instalada no receptor localizado no mesmo barramento da máquina de varredura. Neste caso a aplicação não possuía controles de segurança implementados, isto é, comunicação segura com HTTPS. E por fim, foram identificadas vulnerabilidades possíveis de exploração em ataques. Vulnerabilidades comuns aos equipamentos em rede: TCP *timestamps* (*uptime* pode ser computado); Ethernet; portas abertas sunrpc (111/tcp e 111/udp). Os serviços do protocolo RPC pode possibilitam identificação e conexão remota aos serviços ativos e o RPC *portmapper* permite a identificação de outros serviços RPC no próprio receptor; vulnerabilidade associada ao ICMP. *Timestamp Request Remote Date Disclosure* – receptor responde a requisições de tempo. Permite ao atacante determinar o tempo exato configurado no receptor. Pode ser usada na exploração de protocolos de autenticação.

A SANS1 fornece uma lista de vulnerabilidades para a porta TCP 111. Após esta analise foi possível concluir que a vulnerabilidade desta porta 111 se encontrada aberta só poderá ser explorada por um canal de retorno (acesso via Internet), não sendo tratado nesta analise preliminar.

Este primeiro teste possibilitou aos autores a descoberta do sistema operacional utilizado por um dos modelos na análise. A partir desta revelação foi possível investigar maiores detalhes quanto as vulnerabilidades neste sistema operacional embarcado, bem como as vulnerabilidades de hardware, do *middleware* Ginga NCLua, e da aplicação presentes no receptor. Para a exploração destas vulnerabilidades foi identificada a necessidade de desenvolvimento de uma Aplicação que possibilitasse a investigação interna no receptor digital, a qual será detalhada a seguir.

[1] http://www.sans.org/security-resources/idfaq/blocking.php

5.5.4. A Interface restrita no Receptor

Um dos aspectos mais diferenciados do desenvolvimento de aplicações para os receptores de televisão digital brasileiros é a interface restrita para a interação homem-maquina. De modo geral, o único dispositivo de entrada de comandos pelo usuário é o controle remoto da TV, o qual é geralmente mais limitado em opções de comandos que um telefone celular comum com teclado numérico e poucas teclas extras.

Esta dificuldade (independe da resolução da imagem ou do tamanho da tela da TV) relativa de interação homem-máquina afeta não somente a usabilidade dos aplicativos, mas também a exploração, a partir da interface com usuário, das vulnerabilidades presentes nas camadas de software subjacentes. Trata-se neste texto do uso do receptor como plataforma de computação, daí a dificuldade de interação e entrada de dados.

Dado o padrão atual de utilização destes receptores e a hipótese de evolução das técnicas de ataque aos receptores ser análoga às dos *smartphones* e PCs, surgem às seguintes questões:

1. Seria possível explorar as vulnerabilidades conhecidas em aplicações web ou móveis (p.ex. OWASP top 10) nas aplicações para TVDi?
2. Ainda, como seria possível testar a segurança de aplicações (realizar testes de intrusão) destas aplicações, dadas as limitações de IHC?
3. Outra questão é a execução de scripts de comandos a partir de ferramenta construída para injeção de comandos no receptor. Os scripts de teste de segurança foram exercitados no simulador Ginga-NCL Virtual STB e em receptores TVDi de mercado com Ginga-NCL e Lua embarcados.

Durante a analise dos receptores digitais verificou-se uma facilidade presente em uma caixa comercial onde o usuário habilitado no sistema operacional era o root, e esta descoberta do usuário facilitou a investigação mais detalhada das vulnerabilidades presentes em alguns receptores de TVDi atualmente disponíveis no mercado.

A analise possibilitou ainda, o desenvolvimento de uma investigação automatizada e a alteração dos dados existentes no equipamento em questão, por exemplo, a substituição de contas e senhas do arquivo *passwd*, a remoção de arquivos e as alterações de dados relevantes ao sistema.

Para a execução do teste de penetração no receptor houve a preocupação em obter informações de hardware, do sistema operacional, do *middleware,* da aplicação, e de usuário. Para conseguir uma interface junto ao receptor, foi necessário o desenvolvimento de uma ferramenta denominada AIC – Aplicativo de Injeção de Comandos. Esta ferramenta é um conjunto de interfaces (Figura 10) somadas a alguns scripts automatizadas os quais permitem a Injeção de Comandos, e a Coleta de resultados após a execução pelo receptor. A ferramenta é carregada via USB.

Figura 10 - Aplicativo de Injeção de Comandos

Na camada de Hardware do receptor, por meio de comandos injetados via AIC foi possível identificar a capacidade do equipamento em termos de memória, processamento. A questão da limitação do hardware, quando os comandos de investigação exigiam maior carga de processamento por parte do receptor demandou maior tempo na investigação.

Um dos equipamentos (caixa comercial) investigados apresentava um hardware cuja capacidade de memória é de 1 kbps, e com um processador interno.

```
1. Filesystem           1k-blocks      Used Available Use% Mounted on
2. /dev/mtdblock2           20992     20992         0 100% /
3. /dev/mtdblock3           21504     11660      9844  54% /root
4. /dev/sda1              1952192   1763872    188320  90% /mnt
```

Coleta 1 - Informações de Memória do Receptor

```
1. machine: dados omitidos por medidas de sigilo
2. processor: 0
3. cpu family: sh4
4. cpu type: IDx (nome do equipamento omitido)
5. cpu flags: fpu
6. cache type: split (harvard)
7. icache size: 16KiB (2-way)
```

```
8.  dcache size: 32KiB (2-way)
9.  pll0_clk    : 530.00MHz
10. pll1_clk    : 400.00MHz
11. sh4_clk     : 265.00MHz
12. sh4_ic_clk  : 132.50MHz
13. module_clk  : 66.25MHz
14. slim_clk    : 265.00MHz
15. comms_clk   : 100.00MHz
16. tmu0_clk    : 16.56MHz
```

Coleta 2 - Informações da CPU do Receptor

Foi possível verificar que em alguns receptores o SO embarcado encontra-se com vulnerabilidades e que não há preocupação com a segurança, apresentando algumas vulnerabilidades como excesso de contas onde os usuários não estão marcados como *no login*, et al.

```
1.  root::0:0:root:/root:/bin/sh
2.  bin:*:1:1:bin:/bin:
3.  daemon:*:2:2:daemon:/usr/sbin:
4.  sys:*:3:3:sys:/dev:
5.  adm:*:4:4:adm:/var/adm:
6.  lp:*:5:7:lp:/var/spool/lpd:
7.  sync:*:6:8:sync:/bin:/bin/sync
8.  shutdown:*:7:9:shutdown:/sbin:/sbin/shutdown
9.  halt:*:8:10:halt:/sbin:/sbin/halt
10. mail:*:9:11:mail:/var/spool/mail:
11. news:*:10:12:news:/var/spool/news:
12. uucp:*:11:13:uucp:/var/spool/uucp:
13. operator:*:12:0:operator:/root:
14. games:*:13:100:games:/usr/games:
15. ftp:*:15:14:ftp:/var/ftp:
16. man:*:16:100:man:/var/cache/man:
17. nobody:*:65534:65534:nobody:/home:/bin/sh
18. httpd:*:75:75:httpd:/home/httpd:/bin/false
19. sshd:*:74:74:sshd:/var/empty/sshd:/bin/false
```

Coleta 3 - Arquivo passwd

5.6. Recomendações de segurança

Esta seção traz uma abordagem sistêmica para a segurança da informação e dos sistemas embarcados nos receptores de TVDi. Serão tratados aspectos de desenvolvimento seguro de aplicações, em particular aspectos de projeto seguro de sistemas, programação segura, robustecimento de sistemas operacionais, segurança em redes e implantação e operação seguras. Todos estes aspectos de segurança serão contextualizados na utilização do receptor de TVDi como dispositivo de computação e comunicação.

5.6.1. Limites da segurança de sistemas

Com uma freqüência bastante grande, os programadores são induzidos a falhas de julgamento quando o assunto é segurança da informação. Alguns exemplos são os seguintes: o uso de criptografia seria suficiente para garantir o sigilo das informações; ou ainda, revisões de código e testes de penetração seriam capazes de identificar todas as vulnerabilidades em programas; entre outras. O programa absolutamente seguro é um mito, simplesmente pelo fato de que é impossível determinar se um programa está livre de defeitos que levem a vulnerabilidades e falhas de segurança.

Harrison, Ruzzo, and Ullman (1976) desmistifica a noção de que existe segurança absoluta (por exemplo, "site 100% seguro") em sistemas de computação. Uma das provas é que não há algoritmo genérico capaz de decidir sobre a segurança de uma configuração qualquer de um sistema de proteção qualquer. A força desta afirmação deve ser entendida. Não há esperança de encontrar um algoritmo que possa atestar a segurança de uma configuração arbitrária de um sistema de proteção qualquer, ou de todas as configurações de um determinado sistema. Tal problema é indecidível e remonta ao problema da parada, estabelecido por A. Turing nos anos 30 do século passado.

Outro ponto relevante é o levantado por Thompson (1984). Programas são gerados e executados por outros programas, em uma pilha de execução que vai até o hardware. Thompson (1984) argumenta que vulnerabilidades podem ser exploradas em qualquer nível da pilha de software. Por isso, seria necessário verificar a correção e a integridade de toda a pilha de execução para garantir a confiança no funcionamento do programa do topo da pilha. Thompson (1984) sugere que em casos críticos só é possível confiar em programas de produção doméstica própria. Qualquer outra fonte de programas deve ser considerada suspeita e não merece confiança. Em uma situação extrema, um programador só deveria confiar nos programas escritos por ele mesmo, deste o sistema operacional, até o compilador e as bibliotecas de apoio.

Estes resultados não devem inibir os esforços em verificação de sistemas e segurança de sistemas. Pois, mesmo sem uma solução genérica ampla, há aplicações práticas em casos restritos capazes de atender a maior parte das situações do dia-a-dia. A questão levantada neste texto é como aplicar as boas práticas de segurança da informação ao desenvolvimento seguro de sistemas e a programação segura em Lua. Como estas boas práticas se manifestam em Lua e em NCL?

5.6.2. Desenvolvimento seguro de software

De acordo com Viega and McGraw (2001), Howard and LeBlanc (2002) e Braga (2007), a tendência atual para a segurança de sistemas de software é a prevenção de

vulnerabilidades pela consideração dos aspectos de segurança o mais cedo possível no desenvolvimento dos sistemas. Esta tendência se opõe à cultura reativa de remediação após incidentes. A prevenção de vulnerabilidades nos primeiros estágios do desenvolvimento dos sistemas diminui o custo da segurança, pois costuma ser mais barato evitar a inclusão de vulnerabilidades, que achá-las em testes e corrigi-las. Ainda, a correção de vulnerabilidades encontradas durante testes tem custo menor que a remediação de sistemas em produção ou de produtos finalizados.

A segurança é um aspecto da qualidade dos sistemas. A inclusão da segurança antecipadamente no desenvolvimento dos sistemas aumenta a confiança em um produto final com qualidade. Não é possível ter certeza absoluta da ausência de defeitos, porém a maior confiança no software resultante seria justificada pela utilização de ferramentas apropriadas juntamente aos processos de garantia de qualidade e controle de qualidade do produto final. A Figura 11 ilustra um processo de desenvolvimento seguro de sistemas que a implementação de mecanismos de segurança (segurança funcional) e a consideração de propriedades arquitetônicas na segurança (segurança não funcional), assim como atividades de garantia de qualidade. A Figura 12 ilustra os entregáveis de software, relativos a segurança, distribuídos pelo desenvolvimento de sistemas e acompanhados dos entregáveis tradicionais.

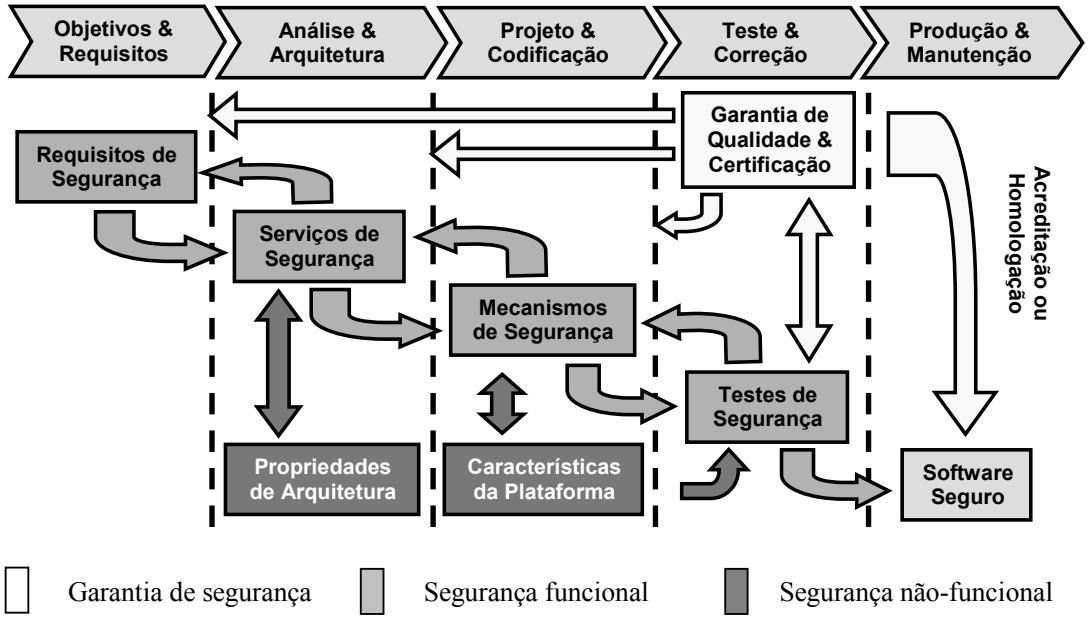

Figura 11: Processo de desenvolvimento seguro de software.

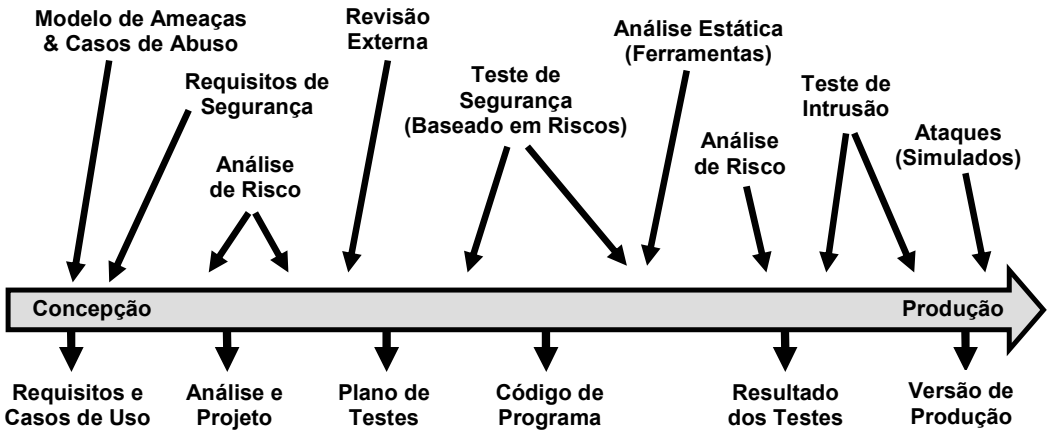

Figura 12: Entregáveis do desenvolvimento seguro de software.

5.6.3. Regras de ouro do software seguro

Em um trabalho pioneiro, Saltzer and Schroeder (1975), foi estabelecido um conjunto bastante famoso de princípios para o desenvolvimento de softwares seguros. Este conjunto é apresentado a seguir.

(1) Economia de Mecanismos: O projeto de segurança deve ser o mais simples e menor possível. Em sendo simples, conterá menos vulnerabilidades. Em sendo pequeno, poderá ser verificado.

(2) Falha Segura: Diante de uma falha, o sistema deve sempre adotar o estado mais seguro para o recurso de valor. Por exemplo, se as decisões de acesso são baseadas em permissões e não em proibições; então tal configuração torna possível ao sistema negar o acesso sempre que não for possível verificar a existência de uma permissão explícita.

(3) Mediação Completa: Todo acesso a todo recurso computacional deve ser verificado. Deve sempre existir um caminho confiável entre o usuário e o recurso computacional, de tal forma que seja sempre possível verificar as permissões de acesso. Atalhos para administradores (backdoors) quebram este princípio.

(4) Projeto Aberto: A segurança não deve ser baseada em segredos de construção e tecnologias proprietárias. Muitas plataformas de software modernas (bastante usadas em sistemas corporativos) possuem programas facilmente descompilados. A segurança por obscuridade, considerada uma prática de eficácia duvidosa, está bastante comprometida pelo avanço das técnicas de engenharia reversa.

(5) Separação de Privilégios ou de Responsabilidades: A criação de usuários todo-poderosos deve ser evitada. Um único usuário (por mais importante que seja) nunca dever ser capaz de realizar sozinho todos os passos de uma tarefa importante, pois tal usuário poderia agir em benefício próprio, sem que outros saibam. Para inibir a fraude, a responsabilidade sobre uma tarefa complexa deve ser dividida entre dois ou mais usuários. Nesta situação, a fraude só seria possível pela conspiração das partes envolvidas.

(6) Privilégios Mínimos: O usuário deve possuir apenas as permissões necessárias para a realização das tarefas necessárias a sua função da organização. Este princípio

complementa o anterior. O usuário com permissões insuficientes é ineficiente, pois não consegue realizar suas funções. Por outro lado, o usuário com permissões em excesso expõe os recursos da organização a riscos desnecessários.

(7) Mínimo Denominador Comum: O compartilhamento de informação de segurança entre muitos usuários deve ser evitado. O compartilhamento de senhas, de chaves criptográficas, de segredos e de outros parâmetros de segurança, entre vários usuários deve ser evitado. Este compartilhamento dificulta a autenticação inequívoca e a responsabilização do usuário.

5.6.4. Codificação (ou programação) segura

Normas de codificação são ferramentas de desenvolvimento seguro, na parte de codificação segura. Deve ser observado que qualquer norma de codificação deverá ser complementada por templates, cujo uso correto poderia ser validado por ferramentas automáticas de análise estática de código.

No que se refere à aplicação de orientação a objetos em Lua, as técnicas de segurança como as sugeridas em (Programming best practices for Java) não podem ser aplicadas diretamente, mas podem servir de inspiração para as boas práticas OO específicas em Lua.

Em relação a boas práticas genéricas para programação segura, diversas práticas podem ser aplicadas. De fato, o catálogo de vulnerabilidades de programação em Lua, descrito em outra parte deste texto, sinaliza diversas estratégias de mitigação das vulnerabilidades. Estas estratégias são aplicações das técnicas e boas práticas de programação segura em Lua.

Não há, até onde vai o conhecimento dos autores deste texto, documentação ou ferramentas de programação segura para Lua. Foram investigadas diversas fontes de informação e observou-se uma ausência completa de informação formal sobre boas práticas de segurança de software em Lua. Dentre as fontes investigadas, podem ser citadas as seguintes como mais relevantes: Lerusalimschy (2003), Lerusalimschy, Figueiredo e Celes (2006), Barbosa e Soares (2008), NCLua Tutorial, Comunidade Lua.

Dois catálogos mais tradicionais de vulnerabilidades de software, SANS/CWE Top 25 (2010) e OWASP Top 10 (2010), não fazem qualquer referência as vulnerabilidades de Lua e nem de NCL.

A norma ABNT NBR 15605-2 define diversos mecanismos de segurança, mas não trata do uso programático seguro destes mecanismos. O mesmo pode ser dito sobre a documentação dos pacotes de software LuaCrypto e LuaSec, ambos responsáveis pelo uso de criptografia em NCLua.

Por outro lado, há em Lua a oportunidade para ferramentas de análise estática. Em duas modalidades, a análise léxica e a análise sintática. Segundo Evans and Larochelle (2002), a ferramenta de análise léxica faz, a grosso modo, uma busca por palavras chave da linguagem e, em princípio, não precisaria entender a sintaxe da linguagem de programação. Um exemplo simples deste tipo de ferramemta é o OWASP CodeCrawler. Ainda de acordo com Evans and Larochelle (2002), a ferramenta de análise sintática faz a varredura de código fonte do programa Lua e, por isto, precisa entender a sintaxe da

linguagem e identificar programas válidos de acordo com ela. Exemplos simples são a ferramenta PMD e o *bytecode verifier* da plataforma Java.

5.6.5. Separação de responsabilidades, compartimentação

Dois conceitos geralmente presentes em políticas de segurança voltadas para controle de acesso são a compartimentação e a separação de responsabilidades.

A compartimentação estabelece a separação de um conjunto ou grupo de entidades em partes isoladas, geralmente sem comunicação entre si. Cada parte agrupa elementos semelhantes e relacionados, de modo que a compartimentação também serve para classificação. Em controle de acesso, a compartimentação não apenas evita a interferência das ações de uma entidade sobre as de outra, mas também impede o conhecimento de uma entidade sobre os assuntos de outras, garantindo o sigilo das informações e a privacidade das pessoas. A técnica de sand-boxing, utilizada atualmente de modo informal em Lua, pode ser um mecanismo poderoso de compartimentação e isolamento de aplicações e do sistema operacional, tal e qual ocorre nos *smartphones* com a plataforma Android da Google, segundo Enck, Ongtang and McDaniel (2009) e Shabtai et al (2010).

A separação de responsabilidades estabelece a divisão de uma transação (tarefa maior) em tarefas menores realizadas por entidades distintas, a fim de evitar que uma única entidade sozinha seja capaz de realizar a transação completa. Uma vantagem direta é a inibição de fraudes individuais, pois cada entidade verifica o trabalho realizado nas etapas anteriores. Com a separação de responsabilidades, a fraude só é possível pela conspiração ou conluio entre as entidades. Quando as aplicações estão associadas a um único usuário root. O usuário de serviço ganha privilégios de administrador e os princípios de separação de responsabilidades e compartimentação não são obedecidos.

5.6.7. Monitor de Referências

Um conceito bastante usado na implementação de mecanismos de controle de acesso é o de monitor de referências, explicado pela primeira vez em Anderson (1972). O monitor é uma entidade mediadora que autoriza ou proíbe o uso de um recurso toda vez que um acesso é requisitado por uma entidade do sistema. Ainda de acordo com Anderson (1972), uma implementação ideal de um monitor de referências deve possuir as seguintes características: (1) Deve ser inviolável (*tamper-proof*) ou, no mínimo, oferecer meios de detecção de violação (*tamper-evident*). (2) Deve mediar todos os acessos a qualquer recurso, isto é, deve ser invocado para todo e qualquer acesso a um recurso. (3) Deve estar confinado em um perímetro definido e facilmente identificável (*security kernel*). (4) Deve ser suficientemente pequeno para que possa ser analisado e verificado quanto a correção de funcionamento.

Uma implementação real do monitor de referências, em geral, negligencia uma ou outra característica. Em particular, implementações em software não costumam ser invioláveis. Por exemplo, as implementações em software do monitor de referências são geralmente encapsuladas em bibliotecas de software e usam códigos de integridade ou de autenticação de programas como mecanismo de detecção de violação.

5.7. Considerações Finais

Em relação à segurança de aplicações, foram identificadas e documentadas diversas vulnerabilidades de programação insegura em Lua. Tais como injeção de comandos, condição de competição, corrupção de arquivos e código malicioso, script cruzado armazenado, referencia insegura a tabela, injeção de SQL e mau uso de criptografia.

Em relação à avaliação de segurança de receptores, foram realizadas varreduras de vulnerabilidades via interface de rede e testes caixa branca de injeção de comandos. Foram encontradas vulnerabilidades graves na configuração dos sistemas operacionais e de proteção de aplicações e seu ambiente de execução.

No futuro próximo, os receptores de TVDi, assim como está acontecendo com os *smartphones* e já aconteceu com os PCs, se tornarão uma fonte importante de vazamento de informações privadas em redes públicas e ambientes de computação em nuvem. Além disso, eles estarão na fronteira de proliferação dos softwares maliciosos e, em conjunto aos computadores pessoais, os *smartphones* e as redes corporativas serão vetores de ataques maciços às redes de computadores e de telecomunicações. As três ameaças que oferecem maior risco seria as seguintes: a contaminação de receptores por softwares maliciosos, o uso destes receptores contaminados como vetores de ataques maciços de negação de serviço e as violações de sigilo e de privacidade decorrentes da descentralização dos ambientes de computação. A primeira ameaça causa uma variedade de problemas, desde a revelação de informação privada até o desvio de recursos do aparelho para outras finalidades. A segunda ameaça afeta diretamente a disponibilidade de serviços na rede subjacente, devido à interconexão das redes a Internet. A terceira ameaça revela a exposição de informações privadas em ambientes de computação em nuvem.

Muito pode ser aprendido com as estratégias de segurança adotadas pelos fabricantes de dispositivos móveis. A despeito de não sofrerem da vulnerabilidade de consumo de bateria, os receptores de TVDi, no que diz respeito a segurança de aplicativos, possuem praticamente o mesmo modelo de ameaças dos *smartphones*. Não é surpresa que a norma brasileira para segurança de aplicativos Ginga seja, em diversos aspectos, semelhante ao modelo de segurança das plataformas modernas de *smartphones*.

5.7.1. Trabalhos futuros

Este texto é o início de um trabalho longo, de esforço contínuo, para dar um tratamento mais amplo à segurança da plataforma de televisão digital interativa brasileira. Há diversas oportunidades para trabalhos futuros.

Em primeiro lugar, a investigação contínua das vulnerabilidades de programação, não somente em Ginga-NCL e Lua, mais também em Ginga-J. Levando a expansão do catálogo de vulnerabilidades iniciado neste texto. Ainda, a elaboração de boas práticas de programação e desenvolvimento de software seguro com a plataforma de TVDi.

Em seguida, avaliação de segurança em implementações de mercado da plataforma Ginga de TVDi, incluindo receptores, simuladores e outros pacotes de software, podendo levar a determinação de normas de segurança mais específicas.

Além disso, outra área a ser explorada é a definição e especificação de requisitos de segurança para serviços ou aplicações de *T-commerce, T-gov e T-banking*, que oferecem no mínimo o mesmo grau de segurança dos serviços equivalentes na Internet.

Finalmente, a construção de ferramentas e de componentes de segurança específicos para a plataforma de TVDi brasileira. Abrangendo tanto ferramentas de apoio ao desenvolvimento seguro, como também a implementação de controles que atendam a determinados requisitos de segurança de aplicações.

5.7.2. Agradecimentos

Os autores gostariam de agradecer a Andre Dorte Dos Santos, a Jéssica Caroline da Silva, a Tadeu Aparecido L. Secco Gessoli, e a todos os profissionais envolvidos no Projeto SMTVI ,pela grande dedicação e ajuda inestimável durante a realização deste trabalho e sem as quais os resultados obtidos não teriam sido possíveis.

Referências

ABNT NBR 15606-2 (2007). Associação Brasileira de Normas Técnicas. Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital. Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações. ABNT 2007. ISBN 978-85-07-00583-4.

ABNT NBR 15605-1 (2008). Associação Brasileira de Normas Técnicas. Televisão digital terrestre — Tópicos de Segurança. Parte 1: Controle de cópias. ABNT 2008. ISBN 978-85-07-01041-8

ABNT NBR 15605-2 (em preparação). Associação Brasileira de Normas Técnicas. Televisão digital terrestre — Tópicos de Segurança. Parte 2: Mecanismos de segurança para aplicativos interativos.

Anderson, R. (1993). Why cryptosystems fail. In Proceedings of the 1st ACM Conference on Computer and Communications Security (Fairfax, Virginia, United States, November 03 - 05, 1993). CCS '93. ACM, New York, NY, 215-227.

Braga, A. M. (2007). Visão geral das boas práticas para construção de softwares seguros. Revista Técnica IPEP, São Paulo, SP, V. 7, N. 2, p. 65-78, jul./dez. 2007. ISSN 1807-8125.

Bragra A. M.; Restani, G. S**.** (2010). Hacking Ginga: uma avaliação de segurança da plataforma de aplicações interativas da TV digital brasileira Anais do X Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (No prelo 2010).

CWE/SANS Top 25 (2010). CWE/SANS Top 25 Most Dangerous Programming Errors. Version 2.0, 2010. Disponível on-line nas URLs www.sans.org/top25-programming-errors e cwe.mitre.org/top25

Deming, W. Edwards (1986). Out of the Crisis. MIT Center for Advanced Engineering Study. ISBN 0-911379-01-0.

Ginga-NCL Virtual STB. Máquina virtual Linux para VMWare e simulador de receptor com Ginga-NCL. Disponível on-line na URL www.gingancl.org.br

Howard, M and LeBlanc, D (2002). Writing Secure Code, Second Edition. December 04, 2002. ISBN 9780735617223

Kocher, P., Lee, R., McGraw, G., and Raghunathan, A. 2004. Security as a new dimension in embedded system design. In Proceedings of the 41st Annual Design Automation Conference (San Diego, CA, USA, June 07 - 11, 2004). DAC '04. ACM, New York, NY, 753-760.

Lerusalimschy, R. (2003). Programming in Lua, 1st. Ed. 2003. ISBN 85-903798-1-7. Disponível on-line na URL www.lua.org/pil.

Lerusalimschy, R., Figueiredo L. H. e Celes, W. (2006). Lua 5.1 Reference Manual. 2006. ISBN 85-903798-3-3. Disponível on-line na URL www.lua.org/manual/5.1/pt.

Lua MD5. Cryptographic Library for Lua. MD5 and DES56 basic cryptographic facilities for Lua. http://www.keplerproject.org/md5

LuaCrypto. LuaCrypto - A Lua Frontend to OpenSSL. Disponível on-line na URL luacrypto.luaforge.net/index.html

LuaSec. LuaSec - TLS/SSL Support for Lua. Disponível on-line na URL www.inf.puc-rio.br/~brunoos/luasec/index.html.

Nicholls, R. (2000) SMS – Today´s Interactive Television. Australia, http://www.broadcastpapers.com.

Oberheide, J. and Jahanian, F. (2010) When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications (Annapolis, Maryland, February 22 – 23, 2010). HotMobile '10. ACM, New York, NY, 43-48. 2010.

OWASP Top 10 (2010). The Ten Most Critical Web Application Security Risks. 2010. Disponível on-line na URL www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. (2004). Security in embedded systems: Design challenges. ACM Trans. Embed. Comput. Syst. 3, 3 (Aug. 2004), 461-491.

Viega, J. and McGraw, G. (2001). Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley Professional. October 4, 2001. ISBN 978-0201721522.

Wilhelm, Thomas. (2010) Professional Pentration Testing, Creating and Operating a Format Hacking Lab – SYNGRESS 2010. ISBN: 978-1-59749-425-0.