UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DE VULNERABILIDADES EM APLICAÇÕES ANDROID COM O USO DE FERRAMENTAS DE TESTE DE INTRUSÃO E A METODOLOGIA OWASP

DIMAS ALBUQUERQUE MENDES

RECIFE – PERNAMBUCO
DEZ/2017

DIMAS ALBUQUERQUE MENDES

ANÁLISE DE VULNERABILIDADES EM APLICAÇÕES ANDROID COM O USO DE FERRAMENTAS DE TESTE DE INTRUSÃO E A METODOLOGIA OWASP

Projeto de Graduação do aluno **Dimas Albuquerque Mendes**, apresentado ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Sergio Castelo Branco Soares

RECIFE – PERNAMBUCO
DEZ/2017

DIMAS ALBUQUERQUE MENDES

ANÁLISE DE VULNERABILIDADES EM APLICAÇÕES ANDROID COM O USO DE FERRAMENTAS DE TESTE DE INTRUSÃO E A METODOLOGIA OWASP

Projeto de Graduação do aluno **Dimas Albuquerque Mendes**, apresentado ao Departamento de Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Aprovada em ____ de _____ de 2017.

COMISSÃO EXAMINADORA:

Prof. Dr. Sergio Castelo Branco Soares
UFPE – Professor
Orientador

_____

Prof. Dr. Leopoldo Motta Teixeira
UFPE – Professor Adjunto
Examinador

“Acho que vírus de computador deve contar como vida. Creio que dizem algo sobre a natureza humana que a única forma de vida que criamos até agora é puramente destrutiva. Nós criamos vida à nossa própria imagem.”
Stephen Hawking.

À minha família e ao Centro de Informática da UFPE.

AGRADECIMENTOS

Agradeço a todos que sempre depositaram suas esperanças em mim ou que me ajudaram durante esta escalada.

Agradeço a meu pai, Anyovaldo, e minha mãe, Josana, por sempre me darem condições de seguir o rumo da minha vida, tomar minhas decisões e me dar todo o apoio possível nos momentos mais difíceis e pelos puxões de orelha. Sem eles eu não seria ninguém.

Agradeço a meus colegas de turma e todas as amizades que fiz nela, que sempre me deram dicas, me ajudaram nessa longa caminhada e noites de sono perdidas comigo em projetos, em especial ao “Team Guido”, a melhor equipe do CIn.

Agradeço a meus amigos pelas boas risadas e desestresses durante o período, por me tirarem da mesmice, aguentarem minhas delongas e dores de cabeça.

Agradeço à equipe docente do Centro de Informática, um centro incrível! especialmente ao professor Sergio Castelo Branco Soares por me dar esta oportunidade de aplicar meu projeto da maneira a qual eu havia pensado.

Agradeço a Google, por me proporcionarem o suporte necessário a encontrar todas as referências, e ao StackOverflow, por me salvar de perder tantas horas com bugs.

A todos que contribuíram direta e indiretamente, deixo aqui meu muitíssimo obrigado.

RESUMO

A forma explosiva com que o mercado de smartphones Android cresceu desde sua adoção conjunta por parte de grandes empresas, fez com que desenvolvedores seguissem a bolha da criação de aplicativos para deliberadamente focar no produto final em prol do rápido desenvolvimento. Por muitas vezes, os desenvolvedores vêm negligenciando questões de segurança, assim expondo milhares de usuários todos os dias à vazamentos de dados e ações maliciosas. O trabalho se propõe a analisar as principais categorias de vulnerabilidades no desenvolvimento de aplicativos Android, utilizando ferramentas auxiliares para detecção destas vulnerabilidades e fazer uma análise com aplicações reais, assim incentivando os desenvolvedores a deixarem suas aplicações mais seguras, no intuito de criar um ambiente de desenvolvimento com menos ameaças.

PALAVRAS-CHAVE: Android. Pentesting. Aplicativos. Vulnerabilidades. OWASP. Segurança.

ABSTRACT

The explosive way in which the Android smartphone market has grown since their joint adoption by large companies has prompted developers to follow the application design bubble to deliberately focus on the end product for rapid development. Often, developers have been neglecting security issues, exposing thousands of users every day to data leaks and malicious actions. The work aims to analyze the main categories of vulnerabilities in the development of Android applications, using auxiliary tools to detect these vulnerabilities and perform an analysis with real applications, thus encouraging developers to make their applications more secure, in order to create a less threatening development environment

PALAVRAS-CHAVE: Android. Pentesting. Applications. Vulnerabilities. OWASP. Safety.

LISTA DE ILUSTRAÇÕES

Figura 1 – Objetivos principais do trabalho 3
Figura 2 – Relação entre ameaça, ativos, vulnerabilidade e riscos 6
Figura 3 – A arquitetura do sistema operacional Android 8
Figura 4 – Conteúdo de um APK. 11
Figura 5 – Assinar um aplicativo com a assinatura de aplicativos do Google. 14
Figura 6 – Assinatura de um aplicativo ao gerenciar sua própria chave de assinatura de aplicativo 14
Figura 7 – Black-box testing vs white-Box testing 16
Figura 8 – Metodologia utilizada neste trabalho. 18
Figura 9 – Logo do AppenBox 22
Figura 10 – Interface do AppenBox 24
Figura 11 – Ferramentas de teste disponíveis no AppenBox 24
Figura 12 – Passos para abrir o Drozer 25
Figura 13 – Configuração do proxy no smartphone Android 26
Figura 14 – Configuração da interface de rede no Burp Suite 27
Figura 15 – Pentest em uma aplicação Android. 28
Figura 16 – Utilizando o Drozer para verificar a superfície de ataque 33
Figura 17 – AndroidManifest.xml 34
Figura 18 – Drozer mostrando activities exportadas e não exportadas 36
Figura 19 – Evento de bateria baixa pode ser usada tanto pelo app quanto pelo atacante. 37
Figura 20 – Broadcast receiver sendo utilizado para enviar uma mensagem no FourGoats. 38
Figura 21 – Confirmação de mensagem mesmo com o app fechado. 38
Figura 22 – Recuperando mensagens enviadas pelo celular por meio de content provider. 39
Figura 23 – Demonstração de service exportado no aplicativo FourGoats. 40
Figura 24 – Iniciando um serviço do FourGoats pelo ADB e Drozer. 41
Figura 25 – Service antes de ser ligado (à esquerda) e depois de ser ligado (à direita). 41
Figura 26 – Listagem de packages 43

Figura 27 – Lista dos 10 aplicativos grátis mais baixados do Brasil. 44
Figura 28 – Pacotes listados em um arquivo 45
Figura 29 – Extraindo uma APK com dois métodos diferentes. 46
Figura 30 – Arquivo .jar descompilado com o dex2jar 46
Figura 31 – Processo de geração de relatórios com o Androbugs 48
Figura 32 – Saída do aplicativo APKTool 49
Figura 33 – Superfície de ataque do app PalcoMP3 50
Figura 34 – Services exportados no PalcoMP3 51
Figura 35 – Activities exportadas no PalcoMP3 51
Figura 36 – Broadcasts exportados no PalcoMP3 52
Figura 37 – Trecho de código Smali DownloadReceiver 52
Figura 38 – Query SQLite de músicas em mp3 52
Figura 39 – Requisição HTTP POST para baixar música revelando e-mail. 54
Figura 40 – Formulário de e-mail do PalcoMP3 54
Figura 41 – Requisição HTTP Post com dados sensíveis 54
Figura 42 – Interface visual do JD-GUI 55
Figura 43 – Código parcialmente ofuscado. 56
Figura 44 – Exemplo de código não ofuscado. 58
Figura 45 – Informações acerca de Activities exportadas, no Drozer. 59
Figura 46 – Módulo Roboguice ainda ativo 59
Figura 47 – Geração de certificados no aplicativo Pessoa Física 60
Figura 48 – Trecho de código que demonstra o uso do HTTPS no Pessoa Física .. 60
Figura 49 – Consulta com Burp Proxy, à esquerda, e sem o Burp proxy, à direita... 61
Figura 50 – Activity não protegida mostrando dados sensíveis com o app em background. 62
Figura 51 – Verificação de permissões de acesso no diretório onde ficam as SharedPreferences. 62
Figura 52 – Tela inicial e menu gaveta do app Caixa. 64
Figura 53 – Classe FingerPrint utilizando KeyStore 66
Figura 54 – Protocolo HTTP sendo utilizado. 67
Figura 55 – Erro demonstrando o uso do JavaScript, no Pidcat. 67
Figura 56 – Tela de cadastro do aplicativo Caixa. 68

Figura 57 – Requisição POST referente à Figura 56. 69
Figura 58 – Erro no cadastro, conta inexistente no Banco, injetando JavaScript. 69
Figura 59 – JavaScript Injection no aplicativo Caixa. 70
Figura 60 – Dados de sessão salvos em banco de dados sem nenhuma proteção. 71

LISTA DE QUADROS

Quadro 1 – Ferramentas auxiliares para pentest em aplicações Android 20
Quadro 2 – Permissões do aplicativo PalcoMP3..... 49
Quadro 3 – Permissões do aplicativo Pessoa Física 57
Quadro 4 – Permissões do aplicativo Caixa..... 65

LISTA DE TABELAS

Tabela 1 - porcentagem relativa de dispositivos por distribuição Android. 1

LISTA DE ABREVIATURAS E SIGLAS

ADT	Android Development Tools
ART	Android Runtime
API	Application Programming Interface
APK	Android Package Kit
DVM	Dalvik Virtual Machine
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
RFC	Request For Comments
JVM	Java Virtual Machine
JNI	Java Native Interface
SDK	Software Development Kit
SSL	Security Socket Layer
TLS	Transport Layer Security
UI	User Interface
OWASP	Open Web Application Security Project

SUMÁRIO

1 INTRODUÇÃO 1

1.1 Contexto e Motivação 1

1.2 Objetivos 3

1.3 Estrutura 4

2. AMEAÇAS, AMBIENTE ANDROID E FERRAMENTAS 5

2.1 Vulnerabilidade, Exploit ou Risco? 5

2.2 Android 6

2.3 A Arquitetura Android 7

2.3.1 O Kernel 7

2.3.2 HAL 8

2.3.3 Bibliotecas 9

2.3.4 Android Runtime 9

2.3.5 Framework de Aplicação 10

2.3.6 Aplicação 10

2.4 Fundamentos de Aplicações Android 10

2.4.1 A estrutura de um APK 10

2.4.2 Activities 12

2.4.3 Services 12

2.4.4 Broadcast Receivers 12

2.4.5 Modelo de Permissão Android 13

2.4.6 Assinatura de APKs 13

2.5 Análise de uma Aplicação 14

2.5.1 Top 10 Vulnerabilidades Android 14

2.5.2 Tipos de Testes 15

2.5.3 Tipos de Análise 16

2.5.4 Pentest 16

3 METODOLOGIA 18

3.1 Ferramentas Necessárias 19

3.2 AppenBox 22

3.2.1 AppenBox - Componentes 23

3.2.2 AppenBox – Licença e Distribuição 24
3.2.3 AppenBox – Configuração do Drozer 25
3.2.4 AppenBox – Configuração do Burp Suite 26
3.3 Metodologia de Pentest Android da OWASP 28
3.3.1 Análise Preliminar 28
3.3.2 Engenharia Reversa 29
3.3.3 Componentes e Arquivos do Sistema 30
3.3.4 Ataques de Rede 31
3.4 Metodologia de Testagem Adaptado 32
3.4.1 Testes de Fluxo de Aplicação 32
3.4.2 Superfície de Ataque 33
3.4.3 Permissões 34
3.4.4 Engenharia Reversa 35
3.4.5 Leitura do Código Fonte 35
3.4.6 Ataques à Activities 36
3.4.7 Ataques à Broadcast Receivers 37
3.4.8 Ataques à Content Providers 38
3.4.9 Ataques à Serviços 40
3.4.10 Ataque à rede 41
3.4.11 Outros Ataques 42
4 TESTES E RESULTADOS OBTIDOS 43
4.1 Bateria de Testes 43
4.2 Aplicativo PalcoMP3 47
4.2.1 PalcoMP3 - Relatório de Vulnerabilidades 47
4.2.2 PalcoMP3 – Componentes Explícitos 50
4.2.3 PalcoMP3 – Intent Implícito 53
4.2.4 PalcoMP3 - Checagem SSL 53
4.2.5 PalcoMP3 - JavaScript 55
4.2.6 PalcoMP3 - Conclusão 56
4.3 Aplicativo Pessoa Física 56
4.3.1 Pessoa Física - Relatório de Vulnerabilidades 57
4.3.2 Pessoa Física - Ofuscação de Código e Testes Rápidos 58
4.3.3 Pessoa Física - Fragment Injection 58

4.3.4 Pessoa Física - KeyStore 59
4.3.5 Pessoa Física - SharedPreferences e SSL Connection 62
4.3.6 Pessoa Física - Conclusão 63
4.5 Aplicativo Caixa Econômica Federal 64
4.5.1 Caixa - Relatório de Vulnerabilidades 64
4.5.2 Caixa - KeyStore e SSL Checking 66
4.5.3 Caixa - Vulnerabilidade JavaScript 67
4.5.4 Caixa - Outras Vulnerabilidades 70
4.5.5 Caixa - Conclusão 71
5 CONSIDERAÇÕES FINAIS 72

1 INTRODUÇÃO

1.1 Contexto e Motivação

Nos últimos anos, o mercado de smartphones cresceu de uma maneira que dispositivos são utilizados a todo o momento, seja em negócios, no governo, em hospitais, hotéis, família e inclusive sendo introduzidos dentro do espaço de trabalho, como meio de comunicação ou até extensão do mesmo se utilizado ferramenta de trabalho (NORTH, 2017). Isso representa inúmeros riscos para organizações tais como invasão à sua rede interna, vazamento de dados sigilosos, processos não autorizados e outras brechas de segurança quando dispositivos utilizados no trabalho são perdidos, especialmente quando falamos de Android, que tem mais de 80% da fatia de mercado (IDC, 2017).

A rápida ascensão do Android se deve há alguns fatores, e o mais importante é sobre sua natureza *open source*, ou seja, em código aberto. Isso significa que é gratuita a implementação por parte das empresas de suas próprias funcionalidades de hardware ou software, mas também traz um grande problema: atualizações (TUNG, 2015). Devido ao fato de que grandes fabricantes falham em entregar suas atualizações, há uma grande fragmentação de versões e um número maior de dispositivos vulneráveis, como se pode ver na tabela abaixo. A primeira coluna indica a versão do sistema, juntamente com o codinome relativo à sua versão (exemplo: Android 8.0 Oreo), na segunda coluna. O número da API significa o número da distribuição em ordem de lançamento da versão e a coluna distribuição indica a porcentagem de dispositivos que estão utilizando aquela distribuição comparado ao total de smartphones rodando o sistema operacional Android.

Tabela 1 - porcentagem relativa de dispositivos por distribuição Android.

Versão	Codinome	API	Distribuição
2.3.3 - 2.3.7	Gingerbread	10	0.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.6%
4.1.x	Jelly Bean	16	2.3%
4.2.x		17	3.3%

Versão	Codinome	API	Distribuição
4.3		18	1.0%
4.4	KitKat	19	14.5%
5.0	Lollipop	21	6.7%
5.1		22	21.0%
6.0	Marshmallow	23	32.0%
7.0	Nougat	24	15.8%
7.1		25	2.0%
8.0	Oreo	26	0.2%

Fonte: GOOGLE (2017).

Somado a isso, desenvolvedores colocam suas criações todos os dias na Play Store (loja de apps do Android), disponibilizando-as para milhões de potenciais usuários ao redor do mundo. Segundo Dave Burke (2016), apenas no ano de 2016, o *Google Play* totalizou 65 bilhões de downloads. Por simples erros, falta de conhecimento ou até falta de importância na parte de segurança, muitos deles criam aplicativos vulneráveis à ataques, permitindo assim o ganho a informações privilegiadas, acesso não autorizado, spam, entre outros tipos de danos ao usuário e/ou ao ambiente onde o smartphone está inserido.

A *Open Web Application Security Project* (OWASP) é um projeto aberto, sem fins lucrativos, onde uma há uma comunidade focada em criar documentos, ferramentas, metodologias e tecnologias para melhoria da segurança de aplicações (CURPHEY, 2003). Um de seus projetos é o OWASP *Mobile Security Project*, que visa dar recursos à desenvolvedores para criarem e manterem suas aplicações mobile seguras. São recursos e padrões de projeto onde o foco principal é na camada de aplicação, dando todo suporte de documentação necessário na detecção de uma variedade de vulnerabilidades, bem como os meios de ataque, impactos técnicos, impactos de negócios, como saber se uma aplicação está vulnerável a determinado tipo de *exploit* e também como prevenir os ataques. Este projeto é bastante importante não só para o mundo mobile, mas para aplicações de computador em geral, pois dá diretrizes para um desenvolvedor atingir um patamar de excelência em segurança, evitando erros básicos de projeto (OWASP, 2006).

Dessa forma, é possível que novos desenvolvedores também contribuam na busca por vulnerabilidades em aplicações, melhoria em seus próprios aplicativos além da própria busca por apps mal-intencionados.

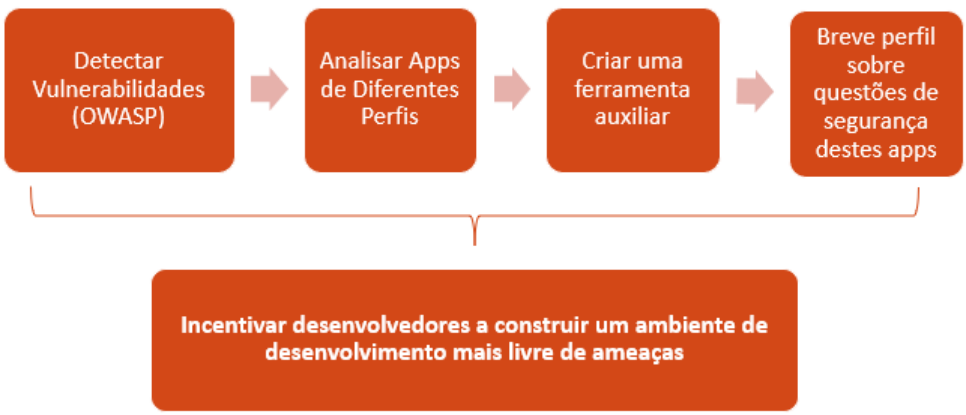
1.2 Objetivos

Objetivando-se o contexto de segurança de aplicações Android, este trabalho contém os seguintes objetivos:

Objetivo Geral – Detectar e analisar vulnerabilidades segundo critérios da OWASP em aplicações Android de diferentes perfis com o uso de ferramentas auxiliares de *pentesting*, para melhoria da segurança no processo de desenvolvimento de aplicativos.

Objetivos Específicos – Criar uma fundamentação teórica a respeito de vulnerabilidades e problemas que estas trazem juntamente com visão geral sobre ambiente de ameaças no Android; Categorizar as vulnerabilidades mais importantes; Levantar ferramentas que auxiliarão na identificação destas vulnerabilidades; Selecionar ferramentas de avaliação com base nos testes dos aplicativos; Criar uma ferramenta própria a partir da seleção das ferramentas; Selecionar alguns aplicativos da loja dado alguns critérios para fazer o perfil comparativo de segurança; Demonstrar a análise; Apresentar dados acerca das vulnerabilidades encontradas; Fazer uma conclusão em cima destes resultados.

Figura 1 – Objetivos principais do trabalho



Fonte: O autor.

1.3 Estrutura

Este trabalho é organizado no seguinte formato: no Capítulo 2 são discutidos conceitos relacionados ao ambiente Android, vulnerabilidades e estruturação de uma aplicação Android. No Capítulo 3 é apresentada a metodologia empregada para realização da avaliação de aplicativos da *Play Store*. No Capítulo 4 os resultados destas avaliações são apresentados. Por fim, no Capítulo 5 estão as considerações finais a respeito dos resultados.

2. AMEAÇAS, AMBIENTE ANDROID E FERRAMENTAS

Neste capítulo, serão apresentados conceitos necessários para o entendimento geral do trabalho.

2.1 Vulnerabilidade, Exploit ou Risco?

Para entendermos o conceito por trás dos tipos de vulnerabilidades primeiro precisamos entender o que seria uma vulnerabilidade, um *exploit* e qual sua relação com o atacante. No campo da computação, segundo a IETF RFC 2828[1], uma vulnerabilidade é "Uma falha ou fraqueza no projeto, implementação ou operação de um sistema que pode ser explorada para violar a política de segurança do sistema."

Com isso, podemos dizer que uma vulnerabilidade ocorre quando há uma fraqueza nas medidas protetivas do sistema que pode ser explorada por um usuário mal-intencionado. Um *exploit* é um termo muito confundido com uma vulnerabilidade, mas deriva do verbo em inglês de mesmo nome, e significa usar alguma coisa para tomar vantagem para si próprio. Esta vantagem não se dá necessariamente pelo uso de um software, mas também por mecanismos como uma sequência de comandos ou até por engenharia social, sem necessariamente usar conhecimentos computacionais (PISCITELLO, 2015). Antes de entender o que seria um risco, deve-se conceituar os *assets*, no português, ativos ou bens. No âmbito geral, seriam todas as coisas que têm um valor associado. Há basicamente dois tipos de valores:

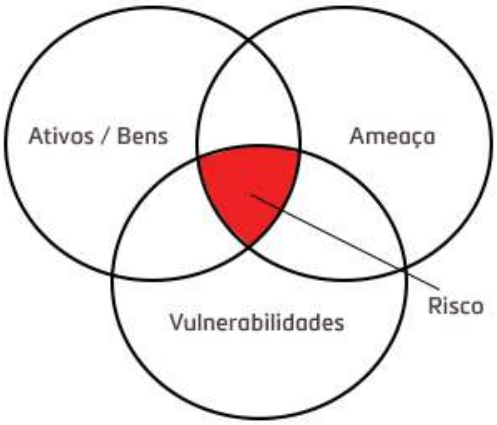
a) tangível ou físico: dinheiro, bens de consumo, e até pessoas;
b) intangível ou não físico: patentes, direitos autorais, informações sensíveis sobre funcionários.

Exemplificando, poderíamos ter uma empresa de consultoria em informática que usasse um sistema de gerenciamento onde haveriam dados de seus funcionários, parceiros, projetos, sua rede interna, seu maquinário de computadores, sua rede interna e sistema de acesso ao prédio. Tudo isso seriam os ativos da empresa, que usaria de medidas protetivas para proteger estes bens, pois em caso de vazamento prejudicaria desde a confiabilidade da empresa até seu funcionamento. Supondo que houvesse uma falha no sistema de acesso remoto por

[1] O Glossário de Segurança da Internet contém abreviações, explicações, recomendações e para uso de uma determinada terminologia. Disponível em: <https://tools.ietf.org/html/rfc2828>.

meio de seu portal para usuários, isso seria uma vulnerabilidade. Um usuário mal-intencionado poderia querer invadir o sistema interno da empresa, e esta intenção seria uma ameaça. Caso este mesmo usuário tomasse esta falha como base para conseguir acesso à rede interna via comandos, ele estaria tomando vantagem para benefício próprio, logo seria um *exploit*.

Figura 2 – Relação entre ameaça, ativos, vulnerabilidade e riscos



Fonte: O autor.

De acordo como Threat Analysis Group (2010), “o potencial de perda, dano ou destruição de um bem como resultado de uma ameaça que explora uma vulnerabilidade". Como vemos na Figura 2, o risco seria o potencial dano que esta empresa fictícia estaria correndo caso uma ameaça quisesse utilizar-se uma vulnerabilidade para ganhar acesso e danificar o sistema interno dela, ou até vazar dados sensíveis para o público.

2.2 Android

Android é uma plataforma *open source* (código aberto) criada inicialmente para smartphones e tablets e posteriormente adaptada para diversos outros sistemas como relógios com o Android Wear, óculos com o Google Glass, TV com Android TV, carros autônomos com o Android Auto. O projeto inicial foi criado secretamente pela Android Inc., que foi fundada em 2003 por Andy Rubin, e apenas em 2005 foi adquirida pelo Google. Por conta do sucesso de vendas do iPhone no ano de 2007, a Google anunciou a criação da Open Handset Alliance (OHA), uma aliança de diversas grandes empresas de tecnologia, tais como Intel, Samsung,

ASUS, Acer, HTC, Qualcomm, Nvidia e T-Mobile (OHA, 2017). Seu objetivo foi entrar mercado de dispositivos móveis com o intuito de desenvolver padrões abertos e desbancar a hegemonia do iPhone, e a maior parte deste código foi lançado em 2007 sob a licença Apache 2.0 (QUERINO FILHO, 2013). Com este tipo de licença, cada fabricante pode customizar o código de seus produtos sem necessitar compartilhar estas alterações (APACHE, 2017). Além desta facilidade, o Android é gratuito, desta forma o Google conseguiu atrair muitas destas empresas para desenvolver em sua plataforma.

Na seção seguinte, veremos os principais componentes arquiteturais da plataforma Android.

2.3 A Arquitetura Android

A arquitetura do Android é dividida nos seguintes componentes, do mais interno para o mais externo: Kernel Linux, Camada de Abstração de Hardware, Bibliotecas auxiliares, Android Runtime, Framework de aplicação e seus aplicativos próprios na camada superior, apresentados na Figura 3.

O conhecimento dos principais componentes da plataforma é de suma importância tanto no campo do desenvolvimento móvel quanto para identificação de vulnerabilidades em aplicações criadas usando o SDK Android.

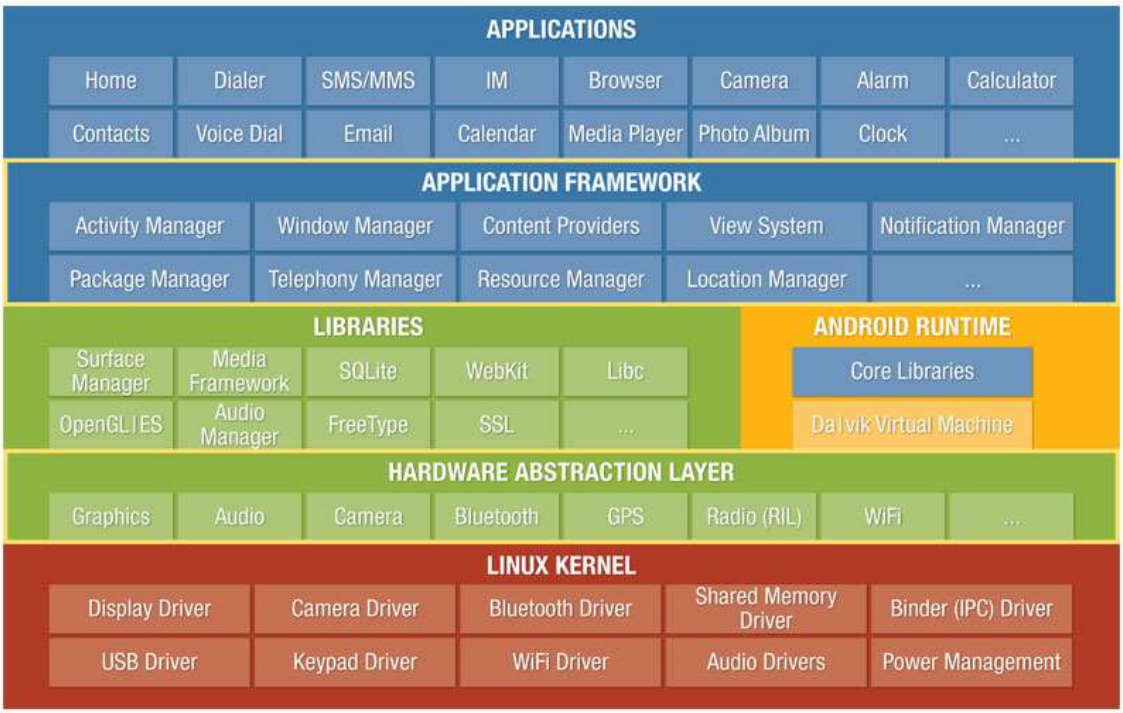
2.3.1 O Kernel

A camada mais baixa possível da arquitetura é chamada de *kernel*, que em português quer dizer núcleo, ou seja, é uma parte essencial e responsável por gerenciar a memória, os processos do sistema, rede, controle de acesso à arquivos e pastas, *drivers* e a segurança em geral.

O Kernel Android é um *kernel* Linux modificado para portabilidade em sistemas embarcados, o que significa que este possa ser utilizado em diversos dispositivos de vários fornecedores. A versão do *kernel* base utilizada varia de acordo com a versão de Android, com a primeira versão, Android 1.5 (Cupcake), criada sob a base do Linux Kernel 2.6.27 até atualmente com o Android 8.0 (Oreo)

utilizando a versão mínima 3.18 ou mais recente segundo as novas especificações do Google (GOOGLE, 2017).

Figura 3 – A arquitetura do sistema operacional Android



Fonte: MANIFEST SECURITY (2016).

Além disso, a modificação deste *kernel* permitiu a adição de alguns mecanismos como gerenciamento de energia, comunicação inter processo (IPC), alarme, *logger* e o *low memory killer*, que é um “matador de processos” e atua quando o sistema necessita liberar recursos (DUBEY; MISRA, 2013). O *logger* fornece um registro de tudo o que ocorre no sistema, e veremos mais sobre ele em um exemplo prático da seção de testes.

2.3.2 HAL

Logo acima, temos a camada de abstração de *hardware* (HAL), que fornece acesso direto aos recursos de hardware por meio da comunicação entre o *kernel* e os níveis acima dele. “Quando uma Framework API faz uma chamada para acessar o hardware do dispositivo, o Android carrega o módulo da biblioteca para este componente de hardware.” (GOOGLE DEVELOPERS, 2017).

2.3.3 Bibliotecas

A camada seguinte corresponde às Bibliotecas, e é referenciado muitas vezes como “camada nativa”, pois os códigos ali escritos estão em C e C++ e são otimizados para rodar sob o hardware Android. O acesso à esta biblioteca por meio dos desenvolvedores se dá por uma interface chamada *Java Native Interface* (JNI), que liga estas bibliotecas à camada de aplicação (DUBEY; MISRA, 2013).

2.3.4 Android Runtime

Apesar do desenvolvimento de aplicativos para Android se dê por meio da linguagem Java, o sistema operacional não utiliza a *Java Virtual Machine* (JVM), a máquina virtual que executa códigos Java, mas sim uma otimizada para dispositivos móveis. Atualmente, Kotlin também é uma linguagem oficial para desenvolvimento de Android a partir da nova versão, o Android 8.0 (Oreo), e tem 100% de interoperabilidade com o Java e Android, porém sua abordagem ficaria além do escopo do trabalho (GOOGLE DEVELOPERS, 2017). Esta máquina virtual foi evoluindo ao longo de sete anos inicialmente com o nome de Dalvik até a versão do Android 4.4 (KitKat). Nesta mesma versão, uma nova máquina chamada *Android Runtime* (ART), ainda mais otimizada e com melhor responsividade à Interface do Usuário (UI) foi introduzida, mas como uma nova *feature* (funcionalidade, característica). Somente na versão 5.0 (Lollipop) a Dalvik foi substituída pela ART de forma integral (GEORGIEV; SILLITTI; SUCCI, 2014, p. 93-102).

Após o processo de compilação dos arquivos Java, se obtêm arquivos com extensão .class, que são arquivos que contém código Java que pode ser executado pela JVM, na qual seu *bytecode* é recompilado para o formato .dex, executável assim pela máquina virtual do Android. Logo em seguida, os arquivos são compactados em um arquivo do tipo .apk (*Android Package Kit*), que representa a versão a ser distribuída e instalada em dispositivos Android (LECHETA, 2015). Falaremos mais sobre APKs na Seção 2.4.

2.3.5 Framework de Aplicação

A camada de Framework de Aplicação é onde fica o Kit de Desenvolvimento de Software Android (Android SDK), que provê diversos módulos para que desenvolvedores possam construir facilmente suas aplicações. Dentre as mais importantes estão: as *Views* (providencia tudo que faz parte da UI de um aplicativo, a parte visual), o *Resource Manager* (gerencia o acesso à recursos), o *Notification Manager* (permite respostas à eventos, em forma de notificações personalizadas), o *Activity Manager* (gerencia o ciclo de vida de uma determinada aplicação) e os *Content Providers* (que permitem que uma aplicação compartilhe seus dados com outras) (GOOGLE DEVELOPERS, 2017).

2.3.6 Aplicação

A camada mais externa, referente às aplicações tanto do próprio sistema (e-mail, mapas, tocador de áudio, contatos telefônicos) quanto aplicativos desenvolvidos por outros desenvolvedores, que são adquiridos oficialmente pela loja Google Play, mas também por outros mercados para Android, porém de maneira não oficial. Cada aplicação é identificada por meio de um identificador único chamado de *package*, ou pacote. Os usuários podem inclusive substituir aplicações oficiais por outras não oficiais escolhidas pelo próprio (DUBEY; MISRA, 2013).

2.4 Fundamentos de Aplicações Android

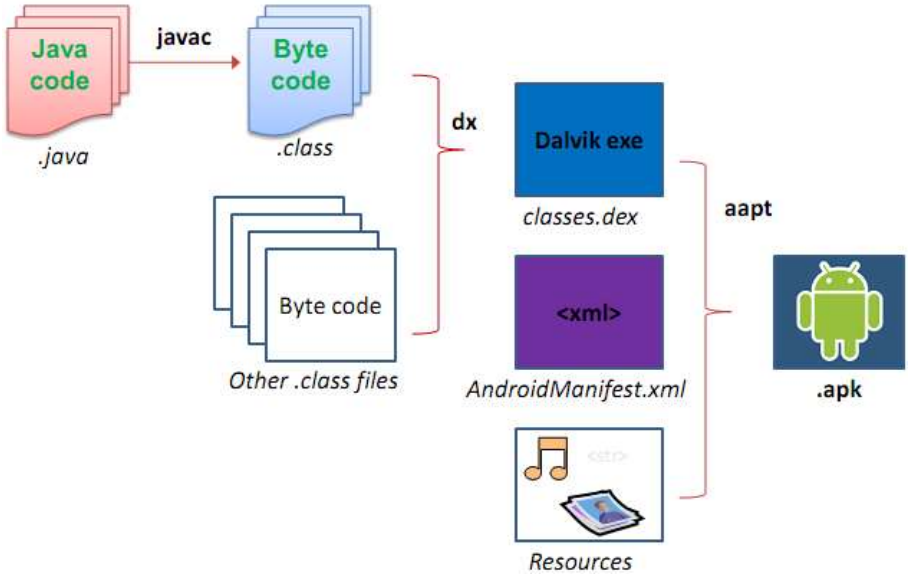
Nesta seção, para melhor preparação ante as ferramentas de testes, iremos ver os conceitos necessários acerca dos componentes de um *Android Package Kit* (APK) e sobre algumas *features* de segurança do Android.

2.4.1 A estrutura de um APK

De forma simplista podemos dizer que um APK nada mais é do que um conjunto de códigos .class compilados em .dex, o manifesto no qual há especificações sobre o aplicativo em geral, e por fim recursos, que são diretórios

contendo strings, layouts, menus e tudo que não era código Java antes da compilação. Veremos cada um deles a seguir, a Figura 4 mostra esta divisão.

Figura 4 – Conteúdo de um APK.



Fonte: MANIFEST SECURITY (2015)

Um APK contém uma variedade de arquivos e recursos, mas normalmente contém certos diretórios e arquivos. O Google Developers (2017), site oficial para desenvolvedores Android, especifica todos estes arquivos. Porém, serão citados apenas os mais importantes para o contexto deste trabalho:

a. **AndroidManifest.xml**: este documento é utilizado para informações organizacionais, contendo 27 tipos pré-definidos de elementos, dentre os quais pode-se especificar componentes da aplicação, permissões necessárias pela aplicação para interagir com outras, bibliotecas necessárias, filtros para *intents* e *broadcasts*;
b. **META-INF:** é um diretório que contém meta informações necessárias para assinatura da aplicação. Alguns arquivos importantes são: MANIFEST.MF (o manifesto em formato de metadado), CERT.RSA (o certificado da aplicação), CERT.SF (contém o sumário SHA-1 correspondente ao manifesto);
c. **lib:** um diretório que contém código compilado especificamente para um processador, e é dividido em diretórios de mesmo nome da arquitetura - armeabi, armeabi-v7a, x86 e mips;
d. **assets:** um diretório contendo arquivos que serão compactados junto com a aplicação;
e. **res:** um diretório que contém recursos que não são pré compilados. Estes incluem layouts, valores (strings, ids, cores) e drawables, que são arquivos

de desenho. A classe R armazena todos os IDs de todos os recursos desta pasta, uma vez definidos em seu XML (string.xml, layout.xml e afins);

f. **proguard-project.txt:** arquivo de configuração do proguard, que serve para ofuscação de código fonte, com o objetivo de evitar com que alguém leia este código.

2.4.2 Activities

No Android, uma *activity* representa uma tela de interface com o usuário, como uma tela de login, ajustes ou tela principal. Ela contém os principais elementos visuais que farão parte da iteração com o usuário, sejam botões, caixas de diálogos ou outras coisas. Cada aplicação possui múltiplas *activities,* que são lançadas via ajuda de um *intent*, que é uma espécie de mensageiro assíncrono que responde à pedidos da aplicação ou sistema e que vincula vários componentes a outros em tempo de execução (BRAHLER, 2010).

2.4.3 Services

Os *services* são componentes sem interface visual que rodam em background e seu objetivo é realizar tarefas longas sem que isso atrapalhe a interação do usuário com a aplicação (AGRAWAL, 2015). Por exemplo, em um aplicativo que sincroniza pastas, esta sincronização estaria acontecendo de forma que não impediria o usuário de navegar pelas pastas, adicionar novos arquivos, ou remover arquivos antigos. Ouro exemplo neste mesmo contexto seria um aplicativo que faz download de músicas não bloquear nenhuma ação do usuário enquanto faz o download da música, tudo isso para não comprometer a experiência de navegação.

2.4.4 Broadcast Receivers

Análogo aos *services*, os *broadcast receivers* também rodam em background e não tem interface, porém executam tarefas de curta duração. A principal diferença para os services são que os *broadcast receivers* respondem à *intents* de eventos em todo o sistema, por isso a maioria deles se originam do próprio sistema e o restante dos próprios aplicativos (customizados), anunciando por exemplo quando a bateria está baixa para o usuário escolher se quer entrar em modo de economia ou não, ou criando uma resposta no momento que o usuário aperta o botão de volume ou ligar a

Wi-Fi. Apesar de não possuírem uma interface visual, eles podem vir acompanhados de notificações na barra de status para alertar o usuário que um determinado evento está ocorrendo (AGRAWAL, 2015).

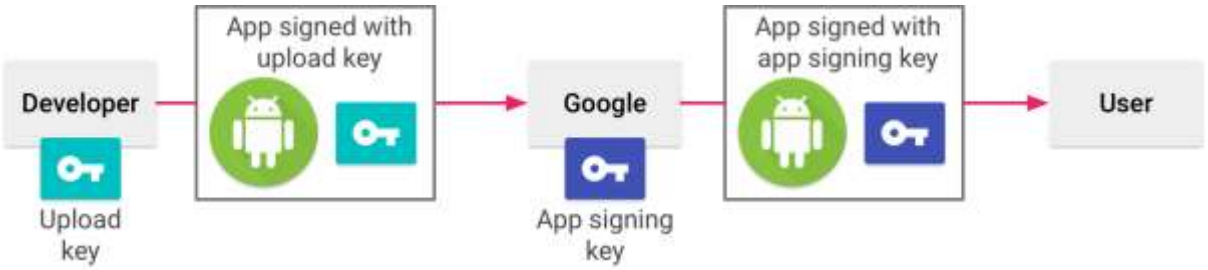
2.4.5 Modelo de Permissão Android

Como falado na seção sobre o Kernel Linux modificado, algumas funcionalidades foram implementadas para garantir a integridade do sistema. Existem as chamadas APIs protegidas, que apenas podem ser acessadas pelo sistema operacional e incluem diversas funções como câmera, localização (GPS), bluetooth, telefonia, sms, rede e dados (LECHETA, 2015). API ou Interface de Programação de Aplicativos, é um conjunto de sub-rotinas, protocolos, ferramentas e padrões de programação que permitem o acesso a um aplicativo de *software*. Basicamente todos os aplicativos Android rodam em máquinas virtuais Dalvik emuladas, assim criando um *sandbox*, uma espécie de caixa de proteção que não permite o acesso à dados de outra aplicação por meio dos provedores de conteúdo, a não ser que esteja especificado nas permissões.

2.4.6 Assinatura de APKs

Todos os aplicativos Android necessitam ser digitalmente assinados com um certificado antes de serem enviados à *Google Play*, a loja do Android. Basicamente há dois tipos de mecanismos de assinatura disponíveis. O site para desenvolvedores Android, o Google Developers (2017), específica estes dois mecanismos. O primeiro mecanismo utiliza uma *Certificate Authority* (CA), que é uma autoridade digital que emite certificados e no caso do Android é a assinatura da Google Play na qual existe uma chave de assinatura de aplicativo, a qual o Google gerencia e protege, e uma chave de upload, que ficará com o desenvolvedor. Este mecanismo é mostrado na Figura 5.

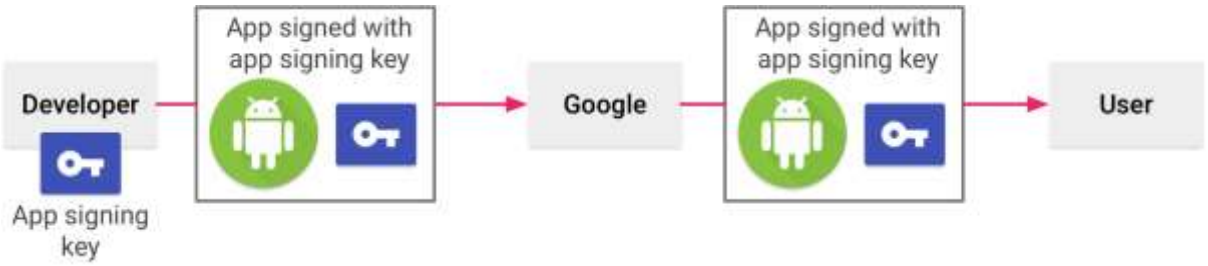
Figura 5 – Assinar um aplicativo com a assinatura de aplicativos do Google.



Fonte: GOOGLE DEVELOPERS (2017).

O segundo mecanismo seria o próprio desenvolvedor gerenciar sua chave de assinatura de aplicativo e um repositório de chaves, e é mostrado na Figura 6. Quando a chave é criada, sua assinatura é feita localmente e o APK poderá ir diretamente para a Google Play. Desse modo, há uma senha que protege o repositório que fica toda por responsabilidade do desenvolvedor. Caso ele perca esta chave, ele não conseguirá lançar novas versões do aplicativo original.

Figura 6 – Assinatura de um aplicativo ao gerenciar sua própria chave de assinatura de aplicativo



Fonte: GOOGLE DEVELOPERS (2017).

2.5 Análise de uma Aplicação

Já falamos de vulnerabilidades, *exploit* e riscos na Seção 2.1. Nesta seção, iremos focar mais na categorização de vulnerabilidades juntamente a ferramentas para testes e análises de aplicações Android.

2.5.1 Top 10 Vulnerabilidades Android

Em seu top 10, a OWASP dá resumo das categorias das principais vulnerabilidades no Android, divididos em categorias segundo seu guia, o *Mobile Security Testing Guide* (MSTG) (MUELLER; SCHLEIER, 2017). São elas:

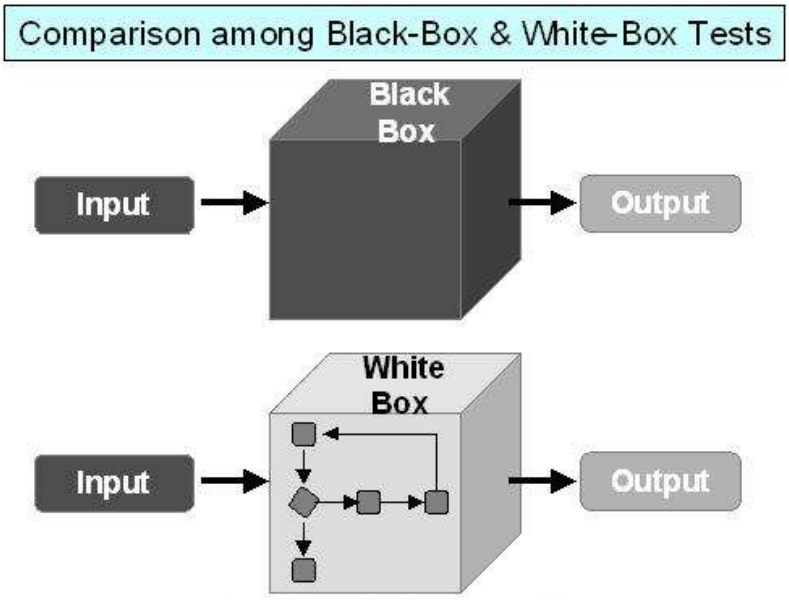
1. **Uso Impróprio da Plataforma:** corresponde ao mal-uso de alguma *feature* ou falha em utilizar controles de segurança. No Android pode ser o mal-uso de intents, permissões, chamadas à API de terceiros ou até algum controle de segurança como Keychain (chaves de acesso privadas);
2. **Armazenamento Inseguro de Dados:** como o nome já diz, compreende o armazenamento de dados de maneira segura ou o seu vazamento não intencional. No Android, os dados podem ser vazados por exemplo, por meio de banco de dados SQLite, logs, XMLs com dados e cartão SD;
3. **Comunicação Insegura:** remete à problemas relacionados ao tráfego de rede do aplicativo, geralmente por não proteger os dados que estão sendo trocados com o servidor;
4. **Autenticação Insegura:** os controles de autenticação são falhos, principalmente porque no mundo mobile, não se sabe se o usuário vai estar *online* ou *offline*, assim dando margem, por exemplo, à execução de métodos no backend da aplicação;
5. **Criptografia Insuficiente:** o código aplica criptografia à alguma informação sensível, porém esta criptografia é insuficiente para proteger estes dados;
6. **Autorização Insegura:** o aplicativo não faz as decisões corretas de autorização. Um exemplo disso seria um usuário com menos privilégio executando uma função de alto privilégio enquanto o aplicativo está em modo *off-line*;
7. **Qualidade do Código:** nesta categoria haveriam todos os problemas relacionados à nível de implementação de código do lado do cliente, como *buffer overflows,* formatação de string e vários outros;
8. **Adulteração de Código:** modificação do código, memória dinâmica e dados do aplicativo por parte do atacante para obter vantagens pessoais ou monetárias;
9. **Engenharia Reversa:** compreende à análise de código binário, revelação de código fonte, algoritmos e outros recursos a nível de código da aplicação;
10. **Funcionalidade Estranha:** funcionalidades que foram utilizadas para propósitos de teste estão acidentalmente na versão r*elease* (lançamento) da aplicação.

2.5.2 Tipos de Testes

Existem basicamente dois tipos de análises quando vamos falamos de códigos em geral, a chamada *white-box testing* que é quando o código fonte é aberto para o teste, ou seja, podemos ver todas as funções, métodos, variáveis, basicamente o que o desenvolvedor ver, todo o código fonte. Já o outro tipo de teste é o chamado *black-box testing*, no qual o testador não tem acesso ao código fonte. No Android, a diferença entre os testes não é tão grande, visto que a maioria dos aplicativos podem ser descompilados para códigos similares ao original. Descompiladores fazem o papel inverso dos compiladores, ao invés de

transformarem código de alto nível para código de máquina, transformam o código de máquina novamente para código de alto nível e, a menos que o código esteja bem ofuscado por um mecanismo que impeça de recuperar totalmente o código original, ou a maior parte dele. Logo, utilizamos um misto de testes *white-box* e *black-box*, que é geralmente a melhor abordagem (DUBEY; MISRA, 2013).

Figura 7 – Black-box testing vs white-Box testing



Fonte: Software Testing Genius (2017).

2.5.3 Tipos de Análise

Para análise de aplicativos Android existem dois tipos de ferramentas. Elas podem ser de Análise Estática ou Análise Dinâmica, sendo a primeira executada quando temos acesso ao APK e executamos as ferramentas sem que o aplicativo esteja em execução, verificando potenciais problemas que podem a ter quando o aplicativo for executado. Já no segundo caso, a ferramenta ajuda na análise enquanto o aplicativo está sendo executado, monitorando desde mudanças em tempo real à arquivos do sistema até comunicações ponto-a-ponto (MUELLER; SCHLEIER, 2017).

2.5.4 Pentest

Um *pentest (penetration test)*, ou teste de penetração no português, é um método de avaliar a segurança de um sistema simulando ataques como se o

testador fosse uma espécie de usuário malicioso e objetiva-se descobrir brechas para ataques antes que sejam descobertos por outros *hackers*, para assim efetivar o necessário e eliminar estes problemas (DUBEY; MISRA, 2013). Segundo o NIST 800-115 (NIST, 2008) e o OSSTMM (HERZORG, 2010), *pentests* podem ser divididos em:

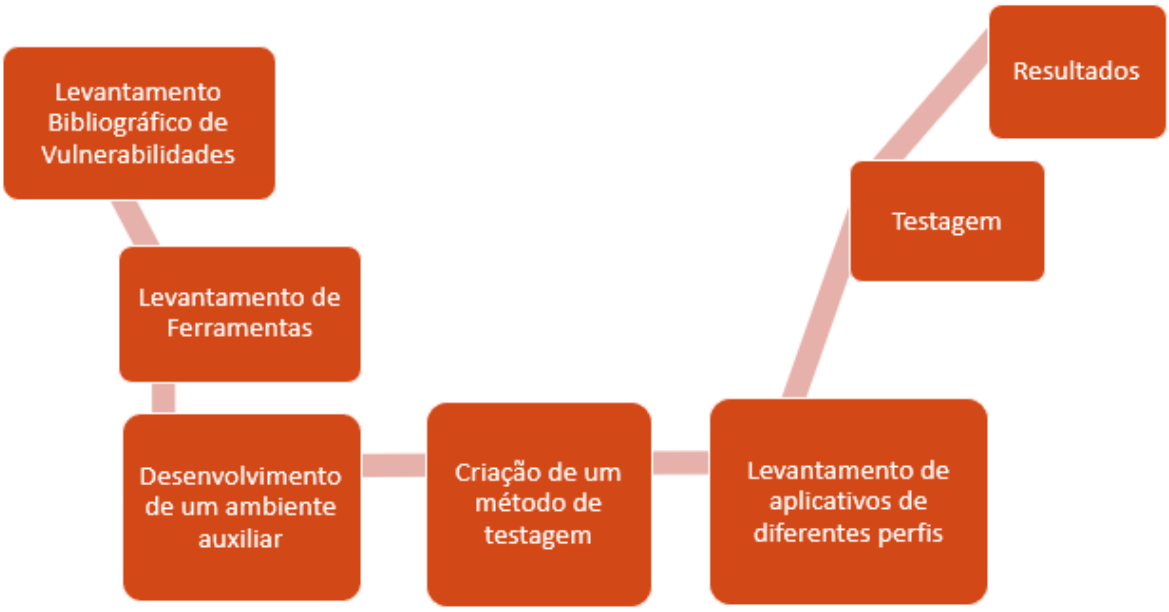
1. **Planejamento:** ocorre antes do início do teste. É levantamento do alvo para saber a abordagem dos testes, escopo, ferramentas necessárias, impacto, valores e contrato com o cliente, além de todo o necessário para começar a coleta de informações;
2. **Avaliação:** é dividida em várias etapas, seria onde ocorrem os testes de intrusão propriamente ditos. Há busca por informações públicas como endereços IP, contatos, tipo e versão banco de dados, potenciais meios de conexão, identificação e exploração das vulnerabilidades;
3. **Ataque:** Utiliza as informações encontradas na fase anterior para planejar ataques em um vetor ou meio de ataque, validar os impactos dessas vulnerabilidades e se necessário voltar a “fase 2”;
4. **Relatório:** Ao final, é gerada uma documentação com os níveis de impacto de cada problema encontrado, datas e horas de testes, ferramentas, informações sobre o projeto além de recomendações para o gerente, que vai executar os devidos cuidados para melhoria.

3 METODOLOGIA

Como descrito no Capítulo 1, o objetivo deste trabalho é a análise de vulnerabilidades em Aplicações Android utilizando ferramentas auxiliares. Para tal, foi feito um levantamento prévio sobre ferramentas de análise estática e dinâmica, levando em conta as vulnerabilidades a serem atacadas. Em seguida, alguns aplicativos foram selecionados de diversas categorias utilizando um *website* de estatísticas da Google Play. O passo-a-passo da metodologia é descrito na Figura 8.

Inicialmente, foi realizado um levantamento bibliográfico das técnicas e vulnerabilidades mais comuns em aplicações Android. As bases para metodologia de testes foram montadas a partir das principais recomendações do guia da OWASP MSTG (OWASP, 2017), apresentado no capítulo anterior em conjunto com o livro *Android Security: Attack and Defenses* (DUBEY; MISRA, 2013) e o guia prático de testes de penetração em Android, *Learning Pentesting for Android Devices* (GUPTA, 2014).

Figura 8 – Metodologia utilizada neste trabalho.



Fonte: O autor.

Ainda sobre os testes de vulnerabilidade, como dentro de cada uma das dez categorias há diversas possibilidades de teste, a pesquisa iria necessitar de um aprofundamento bem maior em diversas áreas como criptografia, *hardware* e redes, além de conter bastante detalhes técnicos que seriam aquém com o propósito do

trabalho. Com isso em mente, foram escolhidos testes simples de cada uma das categorias nos quais desenvolvedores mobile com qualquer nível de habilidade podem executar.

A partir das ferramentas pré-selecionadas também foi desenvolvido um ambiente portátil contendo todas estas ferramentas de modo a facilitar e otimizar o processo de testagem, já que necessita de diversas ferramentas diferentes para determinados tipos de análises.

Para executar os testes de maneira ainda mais segura e concreta, foi desenvolvido um método de testagem com base na metodologia de testes da OWASP na qual foram dados diversos passos para se testar o básico de cada uma das categorias propostas.

Quanto aos aplicativos testados, primeiramente fez-se um estudo no site AppAnnie[2] de quais aplicativos de diferentes perfis são os mais baixados no Brasil no momento, lista em constante mudança por conta do sistema de ranking de apps, que considera diversos pontos como downloads na semana, avaliações, se é um aplicativo novo ou antigo e posição em relação a outros apps gratuitos e com compras. Dentro deste escopo, adicionou-se ainda um filtro a mais: apenas aplicativos de desenvolvedores brasileiros, com propósito de dar mais importância ao mercado brasileiro de desenvolvimento.

3.1 Ferramentas Necessárias

Em relação as ferramentas utilizadas, os principais componentes do ambiente Android são utilizados, incluindo:

1. Computador ou Laptop com 4GB de RAM e pelo menos 500MB.
2. Android SDK.
3. Java Development Kit (JDK) ou Java Runtime Environment (JRE).
4. Python 2.7.
5. Ferramentas Auxiliares de Análise.
6. Smartphone com *root* ou emulador, em modo desenvolvedor.

Em relação às ferramentas auxiliares nas buscas de vulnerabilidades, para cada categoria de vulnerabilidades existem uma ou mais ferramentas que testam a

[2] Atual “top chart” do AppAnnie. Disponível em: <https://www.appannie.com/apps/google-play/top-chart/>

mesma e encontram-se na Tabela 2. Logo, a utilização de ambientes prontos para testes foi uma opção que facilitou a busca por estas ferramentas, foram eles:

- **Santoku Linux** (https://santoku-linux.com/download/) – Com 2.5GB, é uma distro de código aberto focado em segurança de aplicativos móveis. Sua última atualização foi em 2014, porém há uma comunidade ativa e é um ambiente interessante justamente por possuir todas as ferramentas necessárias para testes básicos.
- **PentestBox** (https://pentestbox.org/pt/#download) – É uma modificação do ConEmu, que é uma linha de comando para Windows capaz de abrir múltiplos consoles e tem diversas configurações de personalização. Atualmente na versão 2.2, o PentestBox tem cerca de 1,7GB e dá para ser instalado em um pendrive, além de trazer todas as ferramentas de pentest em um pacote e é possível rodá-las nativamente no Windows, podendo-se até executar alguns comandos importantes de linux. Caso tenha problemas com esta versão, pode-se instalar o Appie v3.1 (https://sourceforge.net/projects/appiefiles/), versão do PentestBox apenas com ferramentas forenses para Android.
- **Kali Linux** (https://www.kali.org/) – Com cerca de 2,7GB na versão *standard* e 800MB na versão *light*, é a distro linux baseada em Debian e favorita dos hackers, profissionais de segurança, pentesters e outros, pois contém mais de 300 ferramentas voltadas aos testes de intrusão, principalmente de Web porém há diversos pacotes de ferramentas adicionais para baixar de acordo com o que se precisa. Não vem com ferramentas pentest Android por padrão.

Todos estes ambientes vêm pré equipados com ferramentas que serão utilizadas para análise de aplicações, seja ela de análise estática ou dinâmica, uma ferramenta serve muitas vezes para mais de uma análise. No escopo do projeto, foram utilizadas apenas ferramentas *open source* ou grátis com foco em pentest Android. Para uma rápida compreensão do uso das ferramentas, fez-se uma tabela com cada uma delas:

Quadro 1 – Ferramentas auxiliares para pentest em aplicações Android

Ferramenta	Descrição	Link do Projeto
ADB	Um programa cliente-servidor em 3 partes: o computador roda o servidor em segundo plano, o adb envia comandos como cliente e um *daemon* que recebe e executa comandos no dispositivo. Ainda conta com um *shell*, um *debugger* e um analisador de *logs*.	Disponível no Android SDK.
AndroBugs	É um *framework* que analisa potenciais vulnerabilidades em aplicações Android. Além disso, procura por comandos perigosos, coleta informações de outros apps, checa se o código segue os padrões	https://github.com/AndroBugs/AndroBugs_Framework

Ferramenta	Descrição	Link do Projeto
	e tem uma GUI simples na linha de comando.	
Androwarn	Detecta diversos tipos de comportamentos maliciosos analisando o código de um app e reporta-os de forma automática.	https://github.com/maaaaz/androwarn/
Androguard	É um conjunto de ferramentas feito em Python que serve para engenharia reversa e análise de malware. Descompila .dex para java, faz análise estática de código, tem uma database de malwares, indicador de risco, mede eficiência de ofuscadores de código como o proguard e outras funcionalidades.	https://github.com/androguard/androguard
APKTool	Engenharia reversa, atua na compilação e descompilação de APKs em Smali e permite a instalação adicional de *frameworks*.	https://ibotpeaches.github.io/Apktool
Burp Suite	Atua como proxy para realização de testes de segurança em aplicações web. Ele possui outras funcionalidades como *crawling* de aplicações, *scanner* de requisições HTTP, ataques automatizados, re-envio de requisições, análise de tokens de sessão, codificação e decodificação mais comuns e comparador de bits.	https://portswigger.net/burp
Bytecode Viewer	Utiliza de outros descompiladores como Procyon e CRF e fornece uma uma interface de usuário (GUI) para navegar pelo código.	https://github.com/Konloch/bytecode-viewer
CFR	Descompilador feito em Java 6 que suporta os lambdas Java 8 além dos switches do Java 7. Descompila arquivos .jar em .class. Ele é grátis porém não é *open source*.	http://www.benf.org/other/cfr/
d2j-dex2jar	Descompila uma aplicação .dex diretamente em .jar.	https://github.com/pxb1988/dex2jar
Drozer	É uma ferramenta *open source* de análise dinâmica de aplicações Android que possui diversas funcionalidades, desde interagir com processos de inter-comunicação como *content providers* até o sistema operacional, lançando *activities*. É necessário possuir o Python 2.7 para rodar a aplicação.	https://labs.mwrinfosecurity.com/tools/drozer/
Enjarify	Similar ao dex2jar porém mais novo e necessita do Python 3 para ser executado, foi criado a partir de códigos da Google e converte arquivos .dex em .jar.	https://github.com/Storyyeller/enjarify
jadx	Uma GUI de navegação fácil para abrir arquivos .jar e assim conseguir acessar os arquivos .class.	https://github.com/skylot/jadx
JD-GUI	Talvez um dos mais utilizados, ainda que não consiga descompilar totalmente o Java 8, possui uma GUI fácil de navegar e abre arquivos .jar para inspecionar os arquivos .class.	http://jd.benow.ca/

Ferramenta	Descrição	Link do Projeto
Pidcat	Um *logcat* (*logger*) que mostra *logs* de um *package* específico. Os erros e avisos são mostrados de maneira bem definida e colorida, o que facilita a visualização. Existe um versão para linux e uma em Python.	https://github.com/JakeWharton/pidcat
Procyon	Tem praticamente as mesmas funcionalidades do CFR, porém é *open source* e necessita do Java 7 para rodar.	https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler
SQLite	O banco de dados sqlite propriamente dito. Atualmente na versão 3.x, é o banco de dados oficial do Android e por causa disso está presente no ADB.	Disponível no ADB do Android.
Wireshark	É um Sniffer (farejador) de rede e analisa o tráfego por meio de uma interface gráfica, podendo ser utilizado para capturar pacotes ethernet, wireless, bluetooth e outros. Além disso, descodifica diferentes protocolos, sendo possível a reconstrução de arquivos.	https://www.wireshark.org/

Fonte: O autor.

3.2 AppenBox

Para facilitar a integração de todas as ferramentas que estão neste trabalho, foi desenvolvido um ambiente com um pacote de softwares pré configurados para rodar em qualquer máquina Windows, o AppenBox (*Android Portable Pentest Box*) - Figura 9.

Figura 9 – Logo do AppenBox

Fonte: O autor.

O AppenBox pode ser descrito como um ambiente de testes portátil em forma de terminal, e contém todas as ferramentas listadas na seção anterior e que permite com que o desenvolvedor carregue elas para qualquer lugar sem a necessidade de criar uma máquina virtual ou configuração prévia, necessitando apenas fazer o download e extrair no computador ou máquina virtual com Windows ou em um pendrive. A ideia é similar ao conceito do PentestBox e Appie e foi baseada da necessidade de se reunir ferramentas indispensáveis para o *pentesting* em aplicações Android.

3.2.1 AppenBox - Componentes

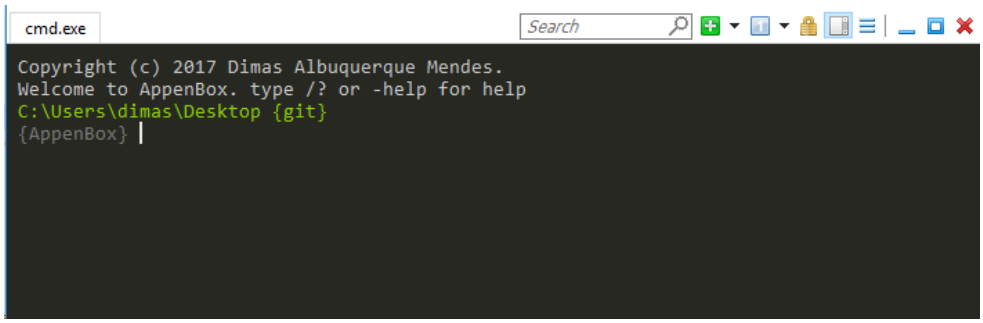
O AppenBox foi criado em cima de alguns softwares-chave que juntos atenderam as necessidades do momento de criação desta ferramenta, são eles:

1. **ConEmu** (https://conemu.github.io/) – É o console/terminal base do AppenBox, um emulador de console Windows personalizável e capaz de apresentar diversos consoles de uma só vez. A partir dele foram utilizados os outros componentes como forma de extensão de programa, como se fossem módulos.
2. **GitHub** (https://github.com/) – É um sistema de controle de versão utilizado por muitos desenvolvedores para gerenciar seus projetos. Ainda conta com diversos comandos linux em seu terminal próprio, tais como *pwd*, *ls*, *chmod*, *find*, *cat*, *more*, *df*, *grep*, entre muitos outros. Utilizou-se do Git como variável de ambiente para o desenvolvedor também poder utilizar os comandos Linux nele disponíveis, no Windows. Geralmente se usa alguma distribuição Linux para programação pessoal, tanto pela facilidade quanto por seus recursos no terminal, e trazer isso para o Windows torna ainda mais fácil de desenvolvedores que trabalham em empresas que utilizam deste sistema operacional adotarem o AppenBox.
3. **Clink** (https://mridgers.github.io/clink/) – É naturalmente uma extensão para o *command* (console do Windows) já que é usado para editar linhas de comando. Conta com alguns scripts, comandos de Unix e pode ser estendido via linguagem de programação Lua para criar *aliases* (nomes alternativos), histórico de comandos, auto completar e criar variáveis de ambiente, assim facilitando a vida do desenvolvedor, que poderá adicionar seus próprios macros, novos *aliases* para não precisar digitar caminho inteiro para as ferramentas, além de dar mais poder ao console ConEmu.

3.2.2 AppenBox – Licença e Distribuição

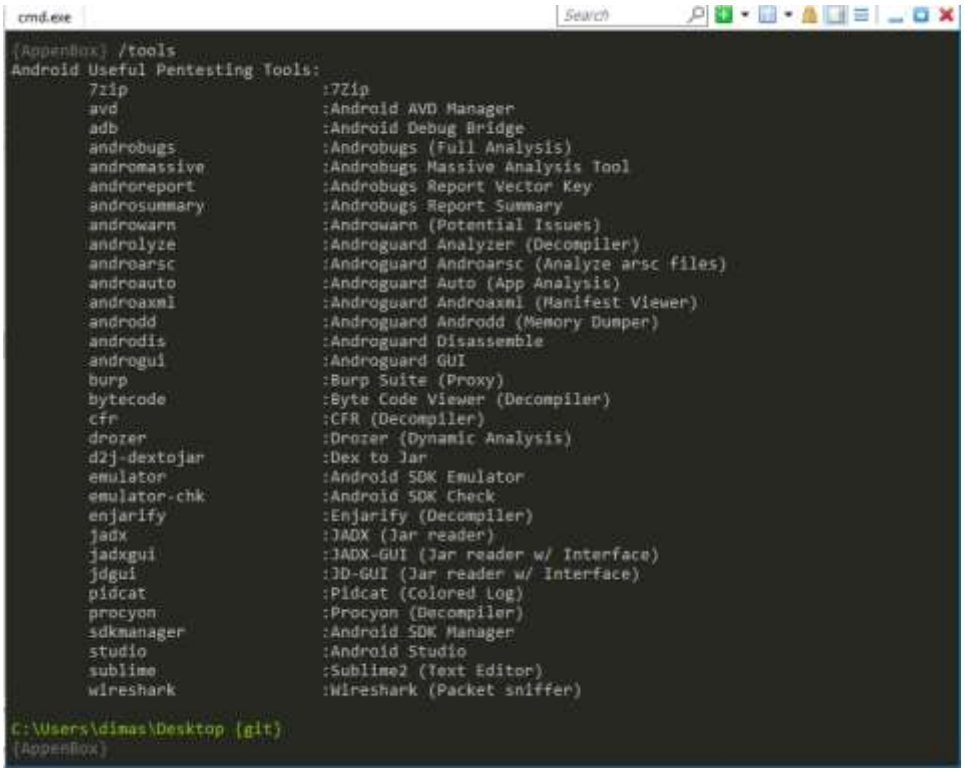
O software ainda contém o Java e o Python embutidos, assim facilitando a execução de softwares que dependem deles. A licença utilizada foi a MIT, que é uma licença permissiva que permite o uso pessoal e comercial, modificação, distribuição e sub licenciamento, contanto que a licença original esteja incluída. Além disso, todas as ferramentas constam com suas licenças individuais. A Figura 10 e Figura 11 mostram o console do AppenBox. O software está disponibilizado no link a seguir: https://github.com/DimasDark/AppenBox.

Figura 10 – Interface do AppenBox



Fonte: o autor.

Figura 11 – Ferramentas de teste disponíveis no AppenBox



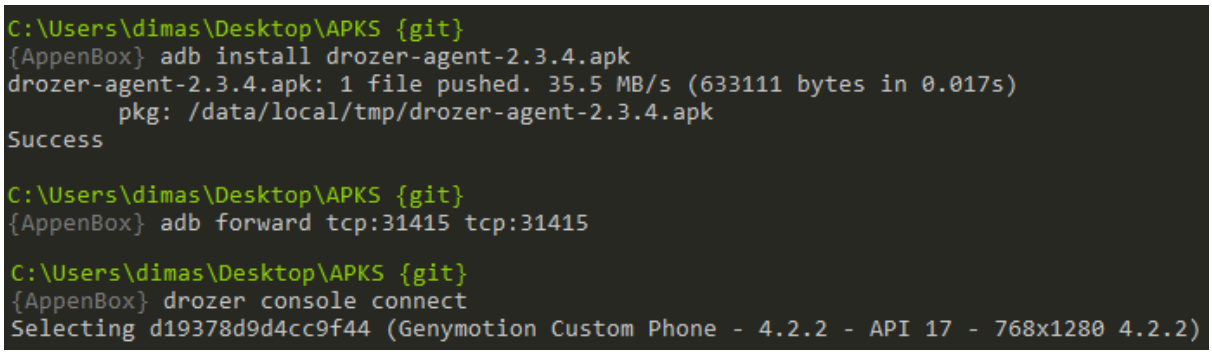
Fonte: O autor.

3.2.3 AppenBox – Configuração do Drozer

A maioria das vezes o teste de uma APK requer muito trabalho manual além do uso das ferramentas, para facilitar isso o empregou-se o Drozer, uma ferramenta poderosa capaz de identificar vulnerabilidades em poucos segundos com seus módulos. Para utilizá-lo, no entanto, é necessário instalar seu aplicativo agente/cliente no smartphone a ser testado, o download deste apk pode ser feito a partir do site oficial do Drozer[3], e já vem com o AppenBox. Depois de baixado, para instalar basta utilizar o comando `adb install [apk]`.

Após a instalação do Drozer, deve-se redirecionar o tráfego da porta TCP 31415 para ela mesma além de abrir o aplicativo no *smartphone*. Para isso, basta digitar `adb forward tcp:31415 tcp:31415`, e para abrir o server `drozer console connect`. Todos os comandos são mostrados na Figura 12.

Figura 12 – Passos para abrir o Drozer



Fonte: O autor.

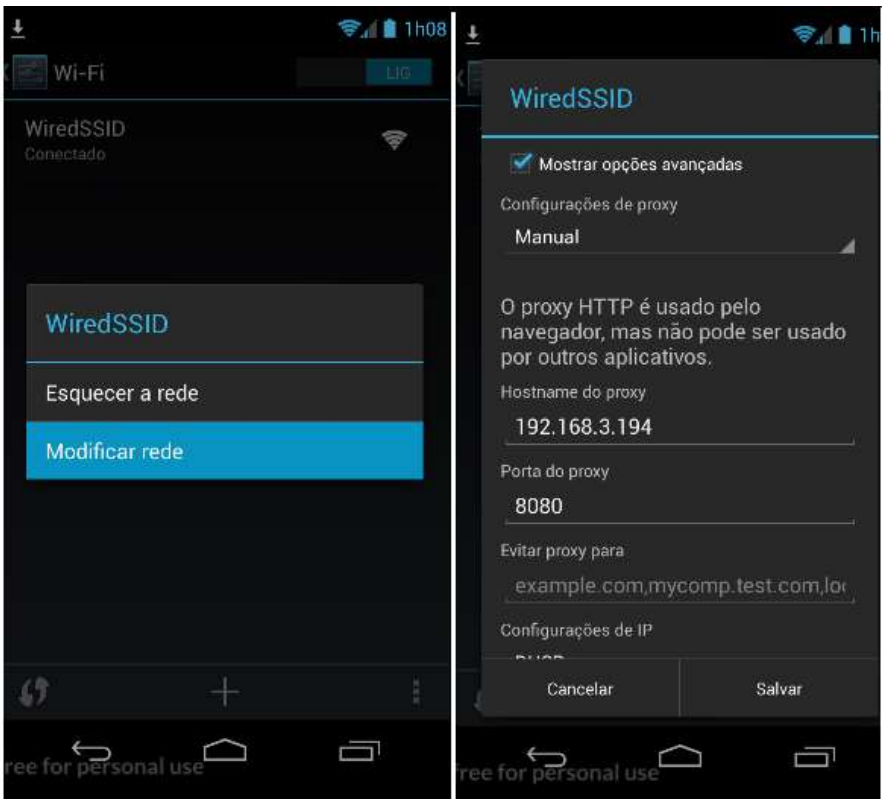
O Drozer conta uma gama imensa de módulos que podem ser listados com todas as suas descrições digitando `list` em seu console, tais como detecção de informações gerais, superfície de ataque, pacotes debugáveis e SQL Injection.

[3] Disponível em: <https://labs.mwrinfosecurity.com/tools/drozer/>

3.2.4 AppenBox – Configuração do Burp Suite

Para verificar vulnerabilidades relacionadas à dados saindo do smartphone como falha de certificados SSL, podemos usar o Burp Suite Proxy como Proxy do Android e capturar requisições, basta digitar o comando `burp` no terminal AppenBox. Em seguida, no Burp, deve-se configurar a interface para o IP do computador host junto com a porta (Figura 13). Para configurar a rede do smartphone a utilizar o computador com Burp como *proxy*, é necessário ir até Configurações > Wi-fi, e com um clique longo na rede Wi-fi conectada, seleciona-se a opção “Modificar Rede” e em seguida deve-se marcar “Opções Avançadas”, no qual o IP deve ser o IP do computador e a porta a que será utilizado no Burp (geralmente 8080), como podemos ver na Figura 14. O IP do computador pode ser adquirido por meio do comando `ipconfig` no console. Ao final, ainda será preciso instalar o certificado da Autoridade de Certificação (CA), para assim não haver problemas para o dispositivo confiar, processo descrito na documentação do Burp[4].

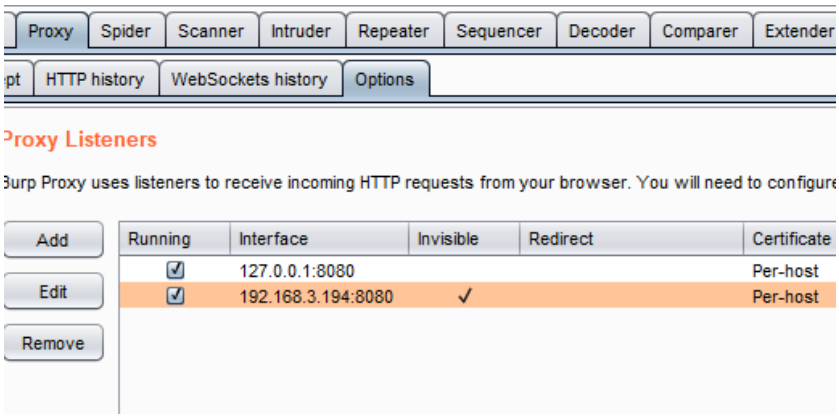
Figura 13 – Configuração do proxy no smartphone Android



Fonte: O autor.

[4] Disponível em: <https://support.portswigger.net/customer/portal/articles/1841102-installing-burp-s-ca-certificate-in-an-android-device>

Figura 14 – Configuração da interface de rede no Burp Suite



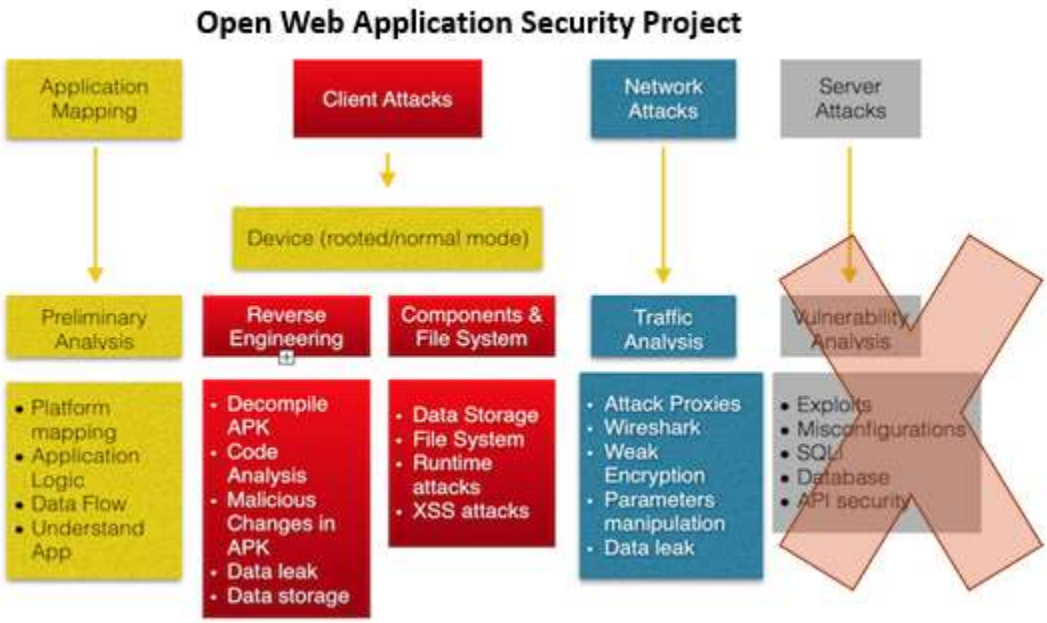
Fonte: O autor.

3.3 Metodologia de Pentest Android da OWASP

Como dito anteriormente, com base nos guias fez-se os testes simples em diversos pontos necessários ao pentest Android. No diagrama da Figura 15, pode-se observar todos os testes que podem ser feitos para detectar as top 10 vulnerabilidades.

Foram feitas algumas pequenas alterações para adequar os testes ao escopo do trabalho, retirando a seção de Server Attacks (Ataques ao Servidor), que estaria fora do escopo por fazer parte do *pentest* de *Web,* já que a aplicação é instalada completamente no smartphone do cliente, logo são focados nos dados que saem dele.

Figura 15 – Pentest em uma aplicação Android.



Fonte: OWASP (2017) – Adaptado.

3.3.1 Análise Preliminar

Primeiramente há um mapeamento com diversas análises manuais, de forma que o testador irá instalar e utilizar o aplicativo recolhendo quaisquer informações possíveis como a lógica da aplicação, qual o gênero da aplicação (negócios, viagens, banco, etc), funcionalidades, APIs usadas (se identificáveis), fluxo de dados, fluxo de telas e se o aplicativo é nativo, híbrido ou web.

3.3.2 Engenharia Reversa

É a primeira parte dos possíveis ataques por meio do cliente. Como visto na Seção 2.3.1 – Estrutura de uma APK, um arquivo .apk nada mais é do que um conjunto de arquivos compilados que contém o código da aplicação em formato .dex. Porém, os arquivos .dex apenas são entendidos por máquinas, o que nos leva a precisar trabalhar com eles de outra forma com intuito de recuperar o código original. Para isso, utilizamos ferramentas como a Apktool que transformarão .dex em .smali, um intermediário entre o código que a máquina entende e o que os humanos entendem, facilitando assim sua compreensão, num processo chamado de *baksmaling*. Entretanto, o código .smali não pode ser convertido em .class pois parte da informação é perdida, então outra opção é tentar recuperar os arquivos .class por meio da conversão de .dex para .jar, utilizando para isso ferramentas como dex2jar ou enjarify.

Para visualizar de um jeito melhor e facilitar o trabalho de abrir todos os códigos de uma só vez, utilizou-se ferramentas de leitura de arquivos .jar como JD-GUI, JADX, que suportam até a versão do Java 7. Para o Java 8, pode-se usar descompiladores mais recentes também como Procyon e CRF, porém os mesmos têm interfaces gráficas à parte, por meio de extensões. O Bytecode Viewer é *open source* e utiliza destes descompiladores de arquivos .jar já citados e oferece uma interface gráfica para facilitar a visualização por meio do usuário.

Com isso pode-se observar se a aplicação está bem ofuscada ou está aberta, ou seja, se quando houver o processo de engenharia reversa é possível produzir um código-fonte tão bom como o original ou um código similar, porém não tão legível, a depender das funcionalidades do ofuscador utilizado no processo de compilação. O Proguard, ofuscador padrão do Android, gratuito e *open source*, tem funcionalidades como eliminação de código não utilizado, remoção de informação de *logs* e depuração e otimização de código, contudo não oferece encriptação de strings estáticas, alteração de fluxo e outras funcionalidades que ofuscadores pagos oferecem.

Com o código resultante em mãos, seja ele ofuscado ou não, é possível analisa-lo em busca de APIs utilizadas, funcionalidades estranhas, controles de *checksum*, controles de debug, aceitação de certificados, confirmar hipóteses da

análise preliminar e procurar por strings ou palavras-chave como as funções hash criptográficas SHA256, SHA ou MD5.

3.3.3 Componentes e Arquivos do Sistema

Esta parte está interligada com a de engenharia reversa, pois as duas dependem da análise do código. É necessário que se use o aplicativo por um tempo para que dados de armazenamento apareçam em bancos de dados como SQLite (*database* oficial do Android), Realm (alternativa ao SQLite) ou outras formas de armazenamento como *Shared Preferences*, uma estrutura em XML que armazena um par (chave, valor) e forma persistente no sistema. Além disso, é essencial checar as permissões do aplicativo no arquivo AndroidManifest.xml (caso não se tenha feito na fase anterior).

Este passo requer a disponibilidade de um smartphone que sofreu processo de *root* e assim obter total acesso como administrador do sistema, acessando pastas onde ficam o cartão de memória como **/sdcard/** e **/sdcard1/** além da pasta da aplicação que contém as *Shared Preferences,* que fica no diretório **/data/data/<nome_do_package>/shared_prefs/** ou bancos de dados. Para os passos anteriores, deve-se usar o adb (Android Device Bridge), que é uma ferramenta que já vem com a SDK do Android e se comunica com um dispositivo Android conectado via USB ou emulador. Com isso, sabe-se se há informações sensíveis espalhadas por arquivos, logs, além de saber se é possível fazer SQL *injection* ou SQLi, um tipo de ataque no qual a aplicação não checa o *input* ou entrada de dados de forma apropriada, e isso possibilita a manipulação de uma consulta no banco de dados SQL.

Outro ponto para testar nessa parte são os *content providers*, no português, provedores de conteúdo. Eles são declarados no arquivo de manifesto do aplicativo e como dito anteriormente, permite a aplicação acessar dados de outra aplicação, assim deixando-a muitas vezes vulnerável a ataques. Para esta tarefa, é utilizado tanto o ADB de modo manual, ou o Drozer de modo automático, que facilita bastante o trabalho de identificação de vetores de ataque.

3.3.4 Ataques de Rede

Aqui são identificadas vulnerabilidades de rede do lado cliente, que é o que mais interessa, avaliando o tráfego que sai do aparelho para o servidor. Com isso, pode-se identificar problemas com SSL, criptografia, autorização de tokens, uso de geradores randômicos e outros ataques de rede se considerarmos aplicações totalmente em *Web* adaptadas para mobile, o que não é o foco do trabalho, já que estamos falando de código nativo Java. Neste passo basicamente coloca-se um *Network Sniffer* como o Wireshark ou um proxy como Burp Suite, Charles Proxy, ou OWASP ZAP e todo o tráfego HTTP(S) é redirecionado por este *proxy*, que estará rodando na máquina que faz os testes.

3.4 Metodologia de Testagem Adaptado

Antes da realização da bateria de testes, foi criado uma metodologia de testagem correspondente com base no teste das dez categorias da OWASP e de sua metodologia, vistas anteriormente. Nas próximas seções, serão explicados o que acontece em cada fase dos testes, que compreendem: fluxo da aplicação, superfície de ataque, permissões, engenharia reversa, ataques à *activities*, *broadcast receivers*, *content providers* e *services* exportados, ataques à rede e outros ataques. Qualquer aplicação vulnerável pode ser utilizada. Para algumas demonstrações foi utilizada a OWASP GoatDroid – Fourgoats, que é vulnerável e próprio para testes. Ela está disponível no link à seguir junto com outros aplicativos vulneráveis: https://github.com/netbiosX/Pentest-Bookmarks/blob/master/Training-Labs/Mobile-Testing/Android-Applications.mdown.

3.4.1 Testes de Fluxo de Aplicação

Esses testes são baseados puramente no controle manual da aplicação, no qual o testador irá abrir o aplicativo, passear por todas as telas observando se há informação que possa ajudar na descoberta de vulnerabilidades ou telas com informações sensíveis. Informações sensíveis são todas as informações que quando utilizadas para obter vantagem ou para o mal, podem causar danos pessoais, materiais ou financeiros (ROUSE, 2014).

Informação sensível pessoal seria um número de CPF, RG, digital, e-mail ou qualquer dado que pudesse levar a identificar o indivíduo por trás daqueles dados, que poderia querer ficar em anônimo. A nível de negócios, seria qualquer informação que pusesse a empresa em risco se fosse divulgada ao público, ou que poderia ser utilizada para ganhar vantagens como acesso não autorizado dentro da mesma. Um terceiro tipo de informação sensível seria de algum órgão governamental no qual houvessem informações classificadas, confidenciais ou secretas que poderiam ser utilizadas para o mal.

Logo, procurar por telas que contenham login, informações pessoais, envio de e-mail, formulários, é essencial. Outro ponto que deve ser considerado neste teste é tentar ocasionar um bug, seja por fazer ações que normalmente não se fariam no fluxo de um usuário comum, como abrir e fechar uma tela diversas vezes,

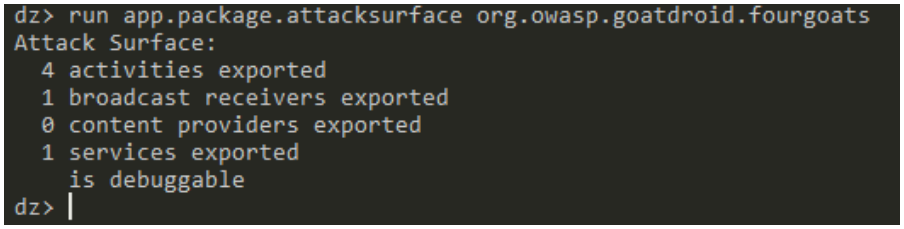
colocar mais dados do que os campos suportam e ver se não ocorre erro de estouro de buffer de memória (*buffer overflow)* ou outras ações que fechem o aplicativo ou não retornem mensagem de erro. Estes geralmente são pegos nos *logs*, e caso o aplicativo não tenha desativado seus *logs* internos é bem possível capturar informações que irão ajudar a identificação de vulnerabilidades, utilizando o *logcat* do ADB ou o Pidcat.

3.4.2 Superfície de Ataque

A superfície de ataque pode ser compreendida como todo e qualquer meio que possa ser utilizado para um ataque tais como as funcionalidades da aplicação, que são testadas no passo anterior. Fora as funcionalidades, podemos ver a superfície de ataque executando o Drozer, que tem módulos que funcionam para verificar este tipo de ataque. Por exemplo, basta digitar `run app.package.attacksurface [package]`, no qual `[package]` é o nome do pacote da aplicação. A figura a seguir mostra as informações que o módulo extrai, tais como *activities*, *broadcast receivers*, *content providers* e *services* que foram exportados e ainda se o aplicativo é debugável ou não. As Seções 3.4.6-3.4.9 mostram ataques à componentes exportados.

Fora isso, O Drozer será utilizado na Seção 4.1 – Bateria de Testes para algumas situações. Fora o Drozer, esta informação também pode ser recuperada por meio do *AndroidManifest.xml*, descompilando o APK por meio do APKTool ou outro descompilador que extraia este arquivo. Com o *manifest* em mãos basta utilizar um editor de texto e procurar por componentes que estejam com `exported=true` (intents exportados), e pela linha `android:debuggable="true"` para saber se o aplicativo é debugável. Não foi utilizado a ferramenta *debugger* do ADB (adb jdwp) neste trabalho por sair fora da proposta de testes básicos a que foi proposto.

Figura 16 – Utilizando o Drozer para verificar a superfície de ataque



Fonte: O autor.

3.4.3 Permissões

As permissões são importantes para saber se há alguma permissão sendo utilizada para outras coisas que não sejam a funcionalidade para qual o aplicativo foi feito ou até permissões soltas e que não são utilizadas, por análise manual. Para olhar as permissões basta abrir o *AndroidManifest.xml* (Figura 17) com qualquer editor de texto e procurar pelas informações com as tags *<uses-permission>*. Também pode-se extrair do manifesto diretamente do arquivo .apk com o *Android Asset Packaging Tool* (aapt), disponível no Android SDK, pelo subcomando *dump*, digitando `aapt dump permissions [apk]` ou `getpermissions [apk]` no AppenBox, no qual `[apk]` é o nome do arquivo .apk. Ou ainda, pegá-las por meio do Drozer, por meio do comando `run app.package.info -a [package]`. Há uma ferramenta de análise estática chamada QARK[5], do LinkedIn, que verifica automaticamente a interpendência de componentes e demonstra permissões que realmente são utilizadas, as que não são e as perigosas, porém como o foco é Windows, a ferramenta não foi utilizada para este trabalho pois apenas funciona em Unix.

Figura 17 – AndroidManifest.xml

Fonte: O autor.

[5] Disponível em: <https://github.com/linkedin/qark>

3.4.4 Engenharia Reversa

Um outro passo ainda na fase de reconhecimento da aplicação é a leitura do código fonte, mas para isso é necessário a extração do mesmo com o uso de descompiladores. Utilizando as ferramentas listadas no trabalho, pode-se usar o d2j-dex2jar, CFR, Procyon ou jadx para descompilar um arquivo .apk para .jar e assim abri-lo em algum visualizador como JD-GUI. Outra opção é utilizar o Bytecode Viewer, que além de descompilar com os três primeiros descompiladores listados, ainda é um visualizador de arquivos .jar. Muitas vezes podemos conseguir informações através de código Smali, que foi introduzido anteriormente e que segundo a Google (2017), é um assembler/disassembler utilizado pela máquina virtual Dalvik, ou seja, converte código de máquina em código escrito na linguagem de entrada, e vice-versa. Para tal, deve-se utilizar o APKTool.

3.4.5 Leitura do Código Fonte

O próximo passo é ler o código fonte a procura de informações que possam ser úteis, a depender do nível de ofuscação do código. Os pontos mais importantes a se determinar são:

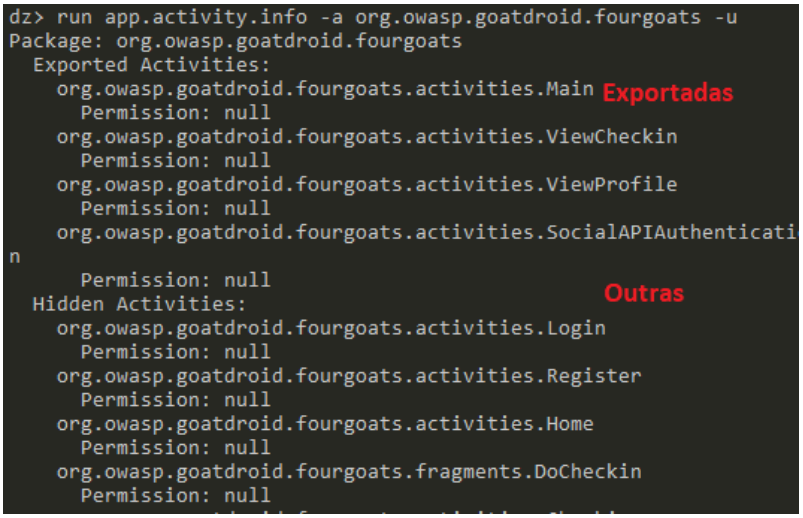
a) As *libraries* / APIs utilizadas;
b) Se são salvos dados sensíveis em SharedPreferences;
c) Determinar o tipo de banco de dados, se há alguma proteção aos seus dados e se há informações sensíveis sendo salvas lá;
d) Se o aplicativo usa criptografia como chaves públicas para validar certificados;
e) Se a comunicação com os servidores (em casos de aplicativos que usam internet) é feita de forma segura, como os parâmetros são passados nas requisições e que tipos de requisições são feitas (GET ou POST);
f) Intents implícitos e se é factível tomar vantagem deles;
g) Em caso de telas de login, como é feita a validação;
h) Se a validação de dados é feita de maneira imprópria ou ideal;
i) Como é feito o tratamento de erros no código, para poder explorar alguma vulnerabilidade relacionada à um erro em particular;
j) Identificar funcionalidade que foi utilizada nas versões de teste e ainda está no código (funcionalidade estranha);

Feito isso, praticamente todas as informações básicas estarão disponíveis para fazer o possível ataque à componentes exportados, que serão abordados nas próximas seções.

3.4.6 Ataques à Activities

Por meio do *AndroidManifest.xml*, encontram-se as *activities* exportadas, e por consequência podem ser acessadas por outros apps. Para pegar as *activities* automaticamente, deve-se usar o Drozer com o comando `run app.activity.info -a [package]` onde `[package]` é o nome do *package* do aplicativo, com o parâmetro opcional `-u` ao fim para mostrar activities não exportadas.

Figura 18 – Drozer mostrando activities exportadas e não exportadas



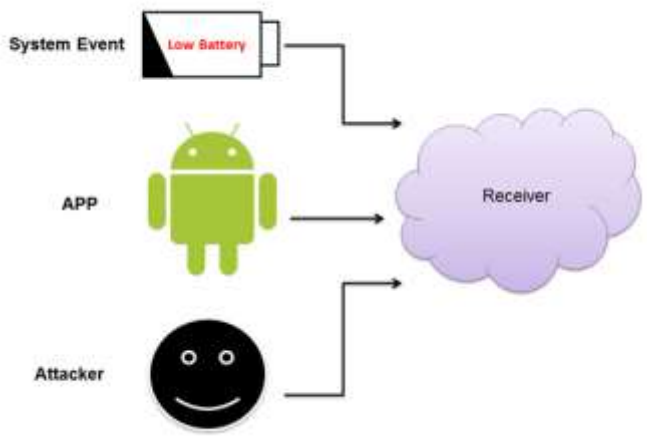
Fonte: O autor.

Casos as *activities* indicadas não tenham nenhuma permissão especial, isso quer dizer que se pode abrir as mesmas fora do fluxo padrão do aplicativo, assim sendo possível fazer testes como burlar telas de login, verificar se há telas com informações sensíveis que podem ser acessadas, entre outros. Para abrir uma *activity* basta utilizar o comando do ADB shell: `adb shell am start -n [package]/[activity]` onde `[packge]` é o *package* e `[activity]` a activity que será aberta. É importante ressaltar que há parâmetros extras que incluem enviar dados para a *activity* a ser aberta, como strings, inteiros, booleanos e URIs. No Drozer, o comando para abrir uma activity é `run.app.activity.start --component [package] [activity]`.

3.4.7 Ataques à Broadcast Receivers

Quando algum evento ocorre, *broadcasts* registrados para aquele evento específicos são acionados. Eles realizam tarefas como por exemplo verificar e-mails não lidos. Há dois tipos principais de *broadcasts*: Do sistema, que são ações como BOOT_COMPLETE, INPUT_METHOD_CHANGED, AIRPLANE_MODE_CHANGED e POWER_CONNECTED, e os customizados, que podem ser gerados por aplicações Android. A figura abaixo demonstra como um atacante pode se aproveitar de um *receiver* para pegar dados ou realizar um *exploit*.

Figura 19 – Evento de bateria baixa pode ser usada tanto pelo app quanto pelo atacante.



Fonte: Infosec Institute (2014).

Os *broadcast receivers* exportados podem ser tanto encontrados no Android Manifest com a tag `<receiver>` como no Drozer com o comando run `app.broadcast.info -a [package]`, no qual `[package]` é o nome do *package* do aplicativo. Para enviar alguma ação para um *broadcast* também pode-se utilizar tanto o ADB quanto o Drozer:

- ADB – adb shell am broadcast -a [action] -n [package/broadcast] [extras];
  - [action] – evento a ser acionado (opcional);
  - [package/broadcast] – nome da package e broadcast completo, com prefixo do pacote;
  - [extras] – parâmetros extras como -es [string] (opcional).
- Drozer – run app.broadcast.send --action [action] --component [package/broadcast] --extra [extras] ;
  - O parâmetro [action] é opcional
  - O parâmetro [extras] é opcional

- A explicação dos parâmetros é a mesma do adb.

As figuras a seguir mostram um *exploit* no aplicativo intencionalmente vulnerável FourGoats utilizando um *broadcast* que há dentro do aplicativo.

Figura 20 – Broadcast receiver sendo utilizado para enviar uma mensagem no FourGoats.

```
F:\AppenBox\support\drozer {git}
{AppenBox} adb shell am broadcast -a org.owasp.goatdroid.fourgoats.SOCIAL_SMS -n
 org.owasp.goatdroid.fourgoats/org.owasp.goatdroid.fourgoats.broadcastreceivers.
SendSMSNowReceiver --es phoneNumber 123456789 --es message Dimas
Broadcasting: Intent { act=org.owasp.goatdroid.fourgoats.SOCIAL_SMS cmp=org.owas
p.goatdroid.fourgoats/.broadcastreceivers.SendSMSNowReceiver (has extras) }
Broadcast completed: result=0
```

Fonte: O autor.

Figura 21 – Confirmação de mensagem mesmo com o app fechado.

Fonte: O autor.

3.4.8 Ataques à Content Providers

Content providers como já dito anteriormente, servem como uma ponte ou interface quando aplicações querem compartilhar dados com outras aplicações. Eles são identificados no manifesto por meio da tag `<provider>` e no Drozer por meio do comando `run app.provider.finduri [package]` em que `[package]` é o nome do pacote do aplicativo, que se estiverem exportados, podem ser acessados por outros apps. Em Android, os maiores problemas que os *providers* exportados

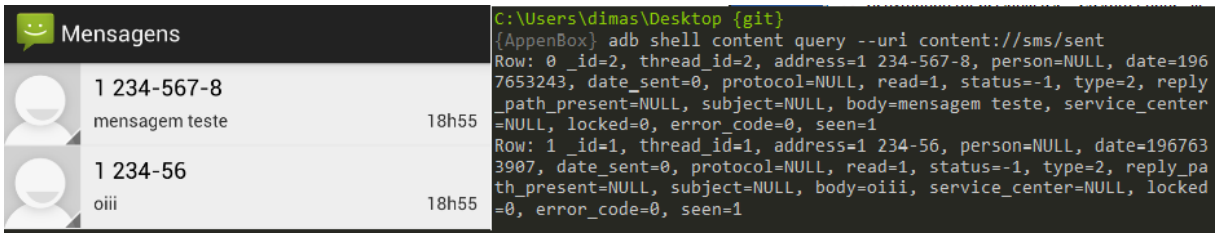
podem causar seriam o *SQL Injection*, que já foi introduzido em seções anteriores e o *Path Traversal*, o qual o atacante pode obter acesso à pastas e arquivos do sistema e o ocorre quando um *content provider* não é bem implementado, sendo similar ao comando `cd ../`, onde se acessa um diretório superior ao atual. Há basicamente três formas de testá-los com as ferramentas providenciadas, se exportados: por meio do ADB, do Drozer ou de algum aplicativo malicioso criado para solicitar dados do *content provider*. Utilizando o aplicativo SMS do próprio Android podemos fazer a requisição à caixa de saída de suas mensagens (content://sms/sent):

a) ADB: adb shell content query --uri [content]
b) Drozer: run app.provider.query [content]
c) Criando um app malicioso:

```
Uri uri = Uri.parse("content://sms/sent");
String[] colunas = new String[] {"_id", "address",
"body"}
ContentResolver cr = getContentResolver();
Cursor c = cr.query(uri, colunas, null, null, null);
```

No lugar de “query” (ação para solicitar), pode-se usar também delete (deletar), insert (inserir) ou update (atualizar). Isso também serve para criar comandos de *SQL Injection* que seriam inseridos na query. O resultado das saídas da requisição anterior será similar nos três casos e pode ser representado pela Figura 22.

Figura 22 – Recuperando mensagens enviadas pelo celular por meio de content provider.

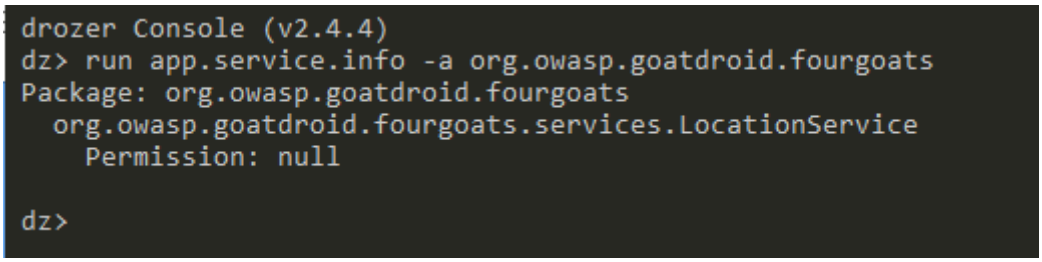


Fonte: O autor.

3.4.9 Ataques à Serviços

Como já foi dito, *services* servem para fazer alguma ação em *background*, ou seja, em segundo plano. Também são utilizados para acesso à recursos entre diversas aplicações, como por exemplo o uso de sensores. No *AndroidManifest.xml* eles são identificados pela tag `<service>`, e no Drozer pelo comando `run app.service.info -a [package]` com o parâmetro opcional -u para incluir *services* não exportados. A Figura 23 demonstra os *services* do aplicativo FourGoats:

Figura 23 – Demonstração de service exportado no aplicativo FourGoats.



Fonte: O autor.

Ter um *service* disponível sem nenhuma permissão de controle é essencialmente perigoso, pois pode ser utilizado por qualquer aplicação maliciosa instalada no mesmo *smartphone*. Como esta é uma aplicação intencionalmente vulnerável, é bem fácil de se identificar e realizar o *exploit* porém em uma aplicação trabalhada dificilmente um *service* será exposto assim. Para utilizar o *service* se quase a mesma sintaxe dos *broadcast receivers*:

- ADB – adb shell am startservice -a [action] -n [package/service] [extras] (inicia um serviço);
  - [action] – evento a ser acionado, opcional;
  - [package/service] – nome da package e service completo, com prefixo do pacote;
  - [extras] – parâmetros extras como -es [string]. Opcional.
- Drozer – run app.service.start --action [action] --component [package/service] --extra [extras] ;
  - O parâmetro [action] é opcional
  - O parâmetro [extras] é opcional
  - A explicação dos parâmetros é a mesma do adb.

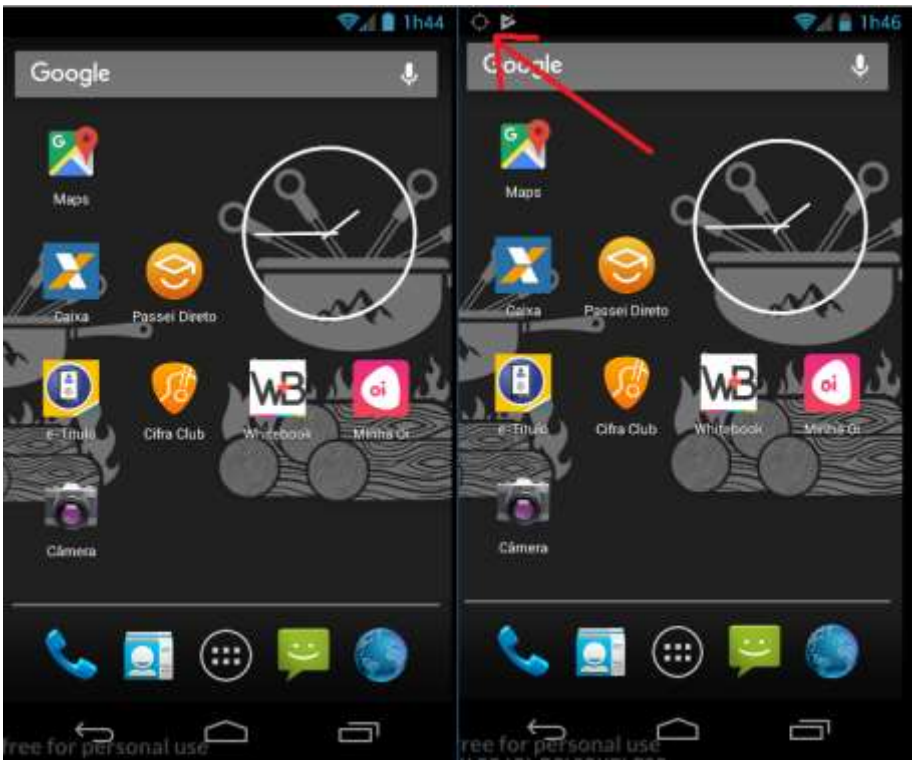
Quando o comando é executado no FourGoats para iniciar um serviço de GPS, a aplicação nem mesmo precisa estar em execução. As figuras abaixo mostram este *exploit*.

Figura 24 – Iniciando um serviço do FourGoats pelo ADB e Drozer.

```
F:\AppenBox\support\drozer {git}                    ADB
{AppenBox}  adb shell am startservice -n org.owasp.goatdroid.fourgoats/org.owas
p.goatdroid.fourgoats.services.LocationService
Starting service: Intent { cmp=org.owasp.goatdroid.fourgoats/.services.Location
Service }

drozer Console (v2.4.4)                             DROZER
dz> run app.service.start --component org.owasp.goatdroid.fourgoats org.owasp.g
oatdroid.fourgoats.services.LocationService
dz>
```

Fonte: O autor.

Figura 25 – Service antes de ser ligado (à esquerda) e depois de ser ligado (à direita).



Fonte: O autor.

3.4.10 Ataque à rede

A parte de ataque à rede na particularidade deste projeto, os testes de rede seriam apenas do lado cliente, pois como dito antes, foi adaptado para focar na aplicação, que está toda instalada no dispositivo do usuário. Neste ponto deve-se

utilizar um proxy para interceptar pacotes de rede e procurar por alguns trechos de código que levem a vulnerabilidades como: SQL Injection, JavaScript Injection (XSS), dados sensíveis não criptografados na rede e Path Traversal. Basta configurar o Burp como mostrado na Seção 3.2.4 e testar algumas suposições:

- SQL Injection: é provável de acontecer quando há *content providers* exportados e que acessam banco de dados na internet, com queries não parametrizadas (verificar no código);
- JavaScript Injection: verificar se o uso do JavaScript é permitido em alguma WebView, principalmente se o app também roda em versões do Android 4.2.2 (Jelly Bean), que será discutido na Seção 4;
- Dados não criptografados: verificar o código e procurar como se checam os certificados e se usa SSL;
- Path Traversal: verificar se o acesso à arquivos do sistema é permitido em alguma WebView.

3.4.11 Outros Ataques

Outros ataques como armazenamento de dados inseguro e dados sensíveis podem ser procurados na pasta /data/data/{package}, em que {package} é o pacote do aplicativo. Lá é onde ficam as SharedPreferences além de bancos de dados SQLite quando utilizado.

4 TESTES E RESULTADOS OBTIDOS

O desenvolvimento do projeto dividiu-se em quatro fases, das quais três já foram abordadas: a modelagem e adequação da metodologia de testes; critério de escolha de amostras de aplicativos para testagem; a criação do ambiente para os testes de vulnerabilidade a partir das ferramentas. Neste capítulo será abordado a obtenção e bateria de testes final e análise de resultados.

4.1 Bateria de Testes

Ao todo foram baixados e testados em todos os aspectos três aplicativos. Ao utilizar as ferramentas indicadas, pudemos demonstrar de maneira intuitiva como realizar o processo de *pentesting* em aplicações Android. Todos os testes foram feitos com o auxílio do ambiente AppenBox, desenvolvido para este trabalho. A organização do relatório teve como base o capítulo 9 do livro *Learning Pentesting For Android Devices* (GUPTA, 2014). Primeiramente baixou-se os devidos aplicativos segundo critérios já citados no capítulo anterior: apps top 10 do mercado brasileiro no momento, de desenvolvedores brasileiros. Somado a isso, foram usados apenas apps *free* e *freemium*, ou seja, aplicativos grátis ou que tem compras dentro dele. A primeira coisa que se pode observar é que dos 10 aplicativos que aparecem na frente dos aplicativos grátis, é que nenhum deles é brasileiro, apenas na posição 11 o primeiro aplicativo brasileiro aparece, o PalcoMP3, como pode-se ver na Figura 27.

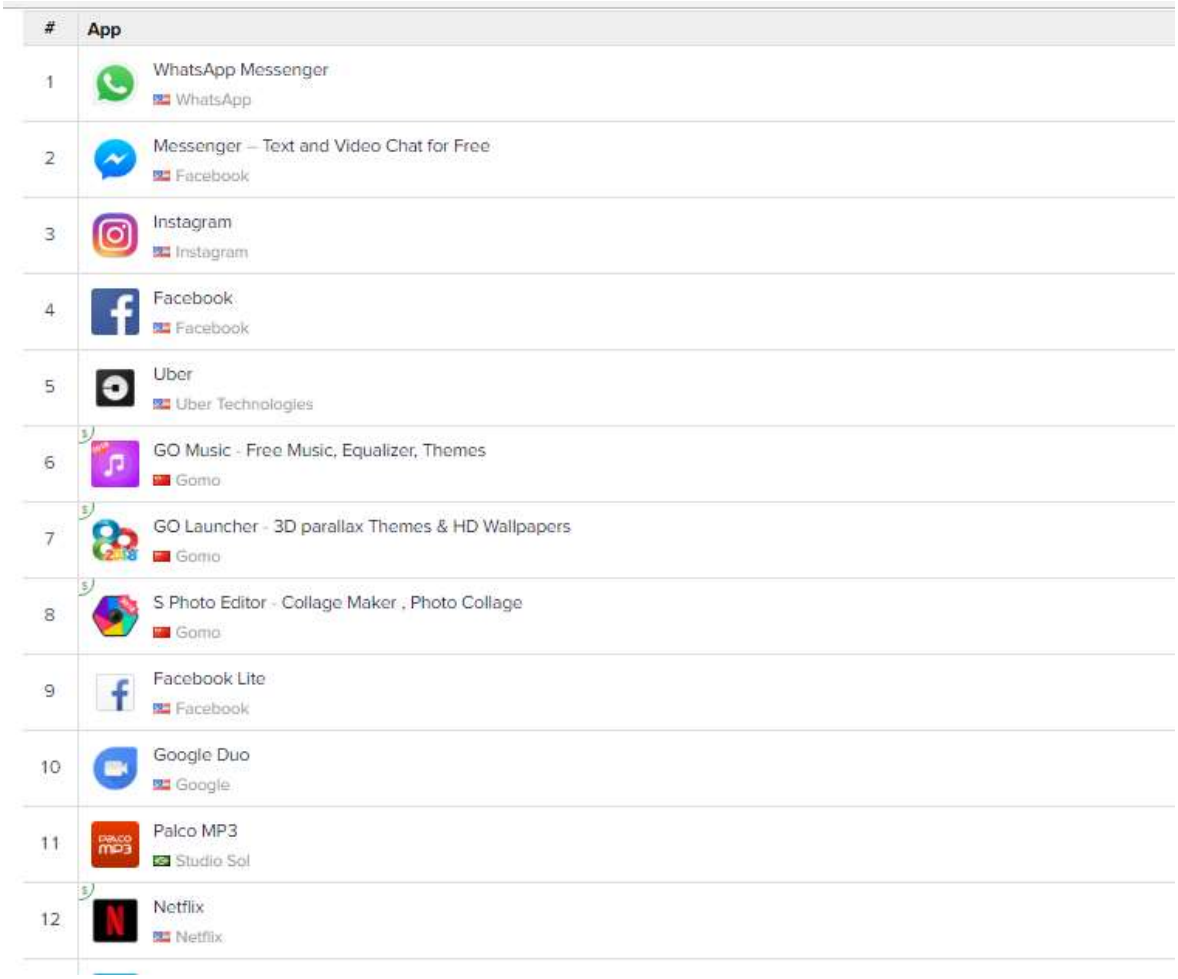
O primeiro passo para testar os aplicativos o primeiro passo foi simplesmente instalá-los na máquina virtual Genymotion com Android 4.2.2 (Jelly Bean) e root. Depois navegar entendendo o fluxo de cada aplicativo, funcionalidades, recursos e outros, foi feito uma listagem de *packages* (pacotes) utilizando o `adb shell` e salvando-os em um arquivo *packages.txt* como é mostrado na Figura 26.

Figura 26 – Listagem de packages

```
C:\Users\dimas\Desktop\APKS {git}
{AppenBox} adb shell pm list packages >> packages.txt
```

Fonte: o autor.

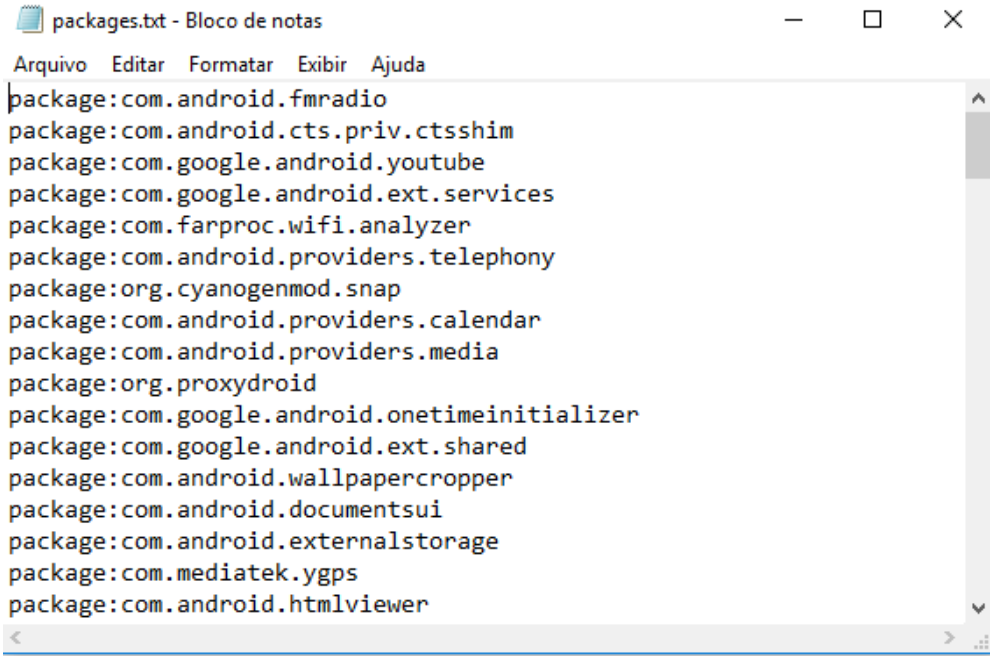
Figura 27 – Lista dos 10 aplicativos grátis mais baixados do Brasil.



Fonte: O autor

Com o nome dos pacotes em mãos (Figura 28), pode-se utilizar a *shell* do adb para navegar até o pacote selecionado que fica no diretório **/data/app/{pacote}-{n}** onde {pacote} é o nome do pacote listado e {n} é um número que o Android cria para a pasta do app. Outra alternativa é utilizar o comando `adb shell pm path [pacote]`. Este diretório é onde ficam os aplicativos não pagos instalados no dispositivo.

Figura 28 – Pacotes listados em um arquivo

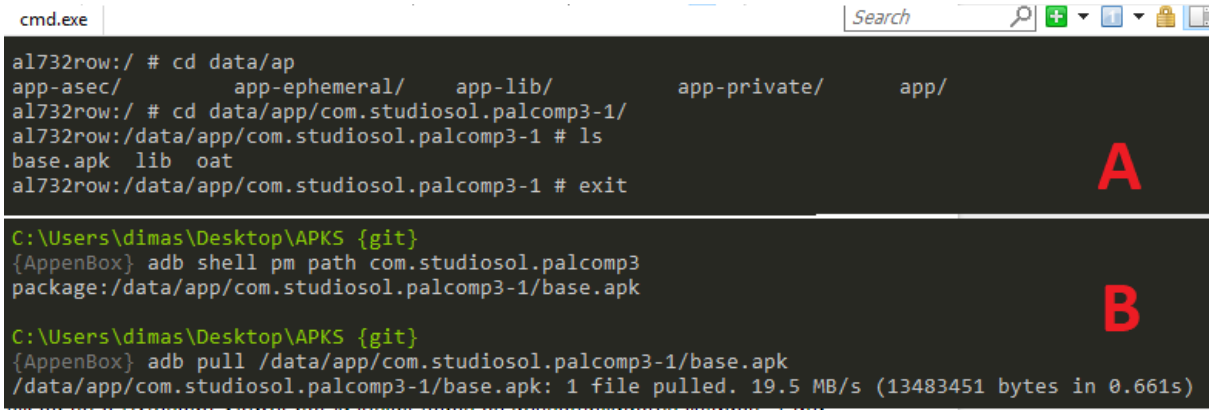


Fonte: O autor.

Em seguida, para extrair o arquivo .apk deve-se copiar todo o caminho específico para o arquivo, sair da shell e utilizar o `adb pull` com o caminho específico do arquivo além do diretório de destino para extrair o aplicativo. A Figura 29 demonstra os dois métodos (A e B) de achar o caminho do arquivo bem como a extração.

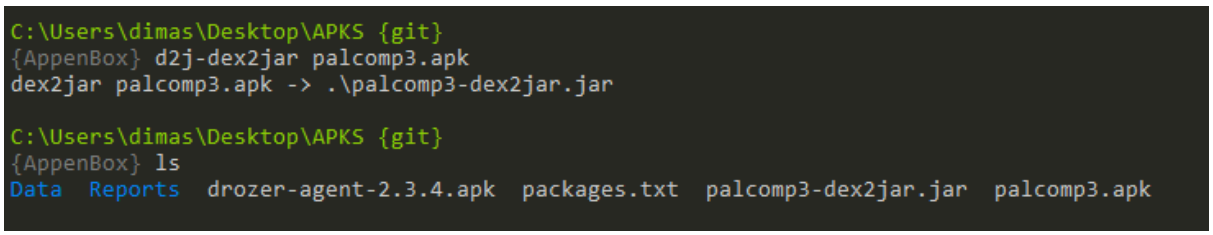
Para analisar o código também deve-se extraí-lo do arquivo .apk, com os descompiladores listados no Quadro 1. É importante ressaltar que diversas ferramentas de análise mais completas já contêm seu próprio descompilador. Utilizou-se o dex2jar por ser mais simples e o mais utilizado, digitando o comando `d2j-dex2jar [apk]`, onde apk é o nome do arquivo apk a ser descompilado em jar. Ele cria um arquivo .jar, como se vê na Figura 30, que poderá ser visualizado com qualquer descompilador jar.

Figura 29 – Extraindo uma APK com dois métodos diferentes.



Fonte: O autor.

Figura 30 – Arquivo .jar descompilado com o dex2jar



Fonte: O autor.

4.2 Aplicativo PalcoMP3

O aplicativo PalcoMP3 é, segundo o ranking do AppBrain, o app grátis número um da Google Play, com mais de cem milhões de instalações e serve para ouvir e baixar música de artistas e bandas independentes do Brasil, com playlists, opções de baixar offline, ranqueamento, entre outras funcionalidades. A versão testada foi a 3.6.23.

4.2.1 PalcoMP3 - Relatório de Vulnerabilidades

Seguinte ao processo de extração visto na seção anterior, podemos começar com várias abordagens para identificar potenciais vulnerabilidades. Pode ser utilizado tanto o Androbugs quanto o Androwarn. Utilizou-se o Androbugs pois há APKs os quais o Androwarn não funcionou por ser uma ferramenta mais antiga. O Androbugs gera um relatório de análise de código no qual podem ser encontradas informações como nível de criticidade, explicações, documentos com referência e recomendações. Ao todo existem 4 categorias de classificações de análise no relatório gerado: *info* (apenas informações sobre o app), *notice* (problemas com baixa prioridade), *warning* (vulnerabilidade não confirmada), *critical* (vulnerabilidade de segurança quase confirmada).

No AppenBox utiliza-se o comando `androbugs -f [apk]` onde `[apk]` é o arquivo .apk a ser testado. Opcionalmente pode-se escolher o diretório para gerar o report com o argumento `-o [local]`, mas por padrão, o Androbugs gera uma pasta chamada Reports. A Figura 31 demonstra o processo de geração de relatório.

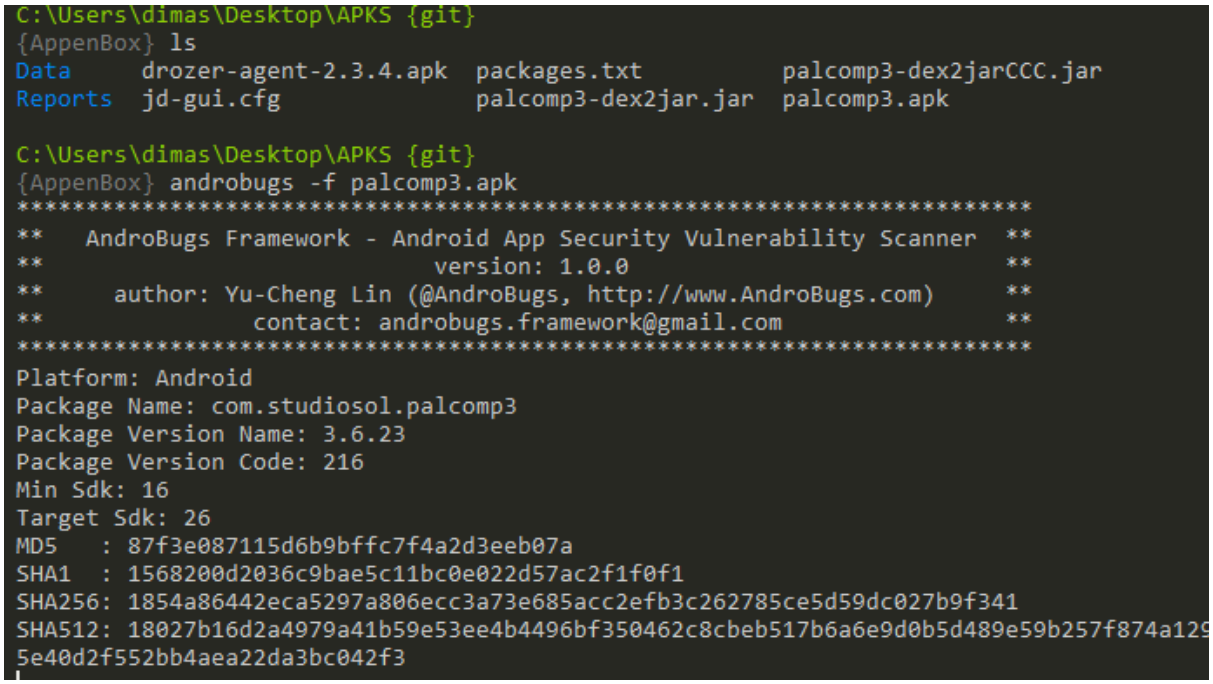
A partir do relatório gerado, precisou-se confirmar manualmente onde estariam as vulnerabilidades críticas e se os *warnings* realmente são vulnerabilidades. Ao todo foram encontradas três vulnerabilidades possivelmente confirmadas (*critical*):

1. **Uso de Intent Implícito:** quando se registra um *intent* e não se especifica que aplicação vai receber os dados, sendo assim, qualquer aplicação pode registrar um filtro e manipular estes dados. Geralmente estes chamados são falsos positivos pois aplicativos tendem a usar este tipo de *intent* para realizar uma ação que seu aplicativo não é capaz, como por exemplo um aplicativo que não foi feito para mandar e-mail, SMS ou compartilhar

imagem pode usar um intent implícito para enviar a ação para outro aplicativo se encarregar;

2. **Checagem SSL:** algumas URLs utilizadas no código não fazem checagem de certificado SSL assim permitindo alguns ataques como *man-in-the-middle*[6], onde a conexão entre dois hosts é interceptada por um terceiro, que acessa a informação dos dois. É um ataque que acontece de diversas formas, mas geralmente no meio Android ocorre pelo uso de comunicação HTTP, o qual deve-se proteger a conexão por meio de certificados SSL/TLS além de recursos do próprio Android como KeyStores com chaves públicas para identificar a conexão;
3. **Execução de código remoto em WebView**: a aplicação utiliza de uma WebView com JavaScript ativo, o que significa que é possível injetar código JavaScript e utilizar de ataques como XSS (*Cross-site Scripting*). No Android 4.2, como reportado na documentação da Google[7], também pode-se utilizar desta falha já conhecida para alterar o código java em tempo de execução pelo uso de Reflection[8].

Figura 31 – Processo de geração de relatórios com o Androbugs



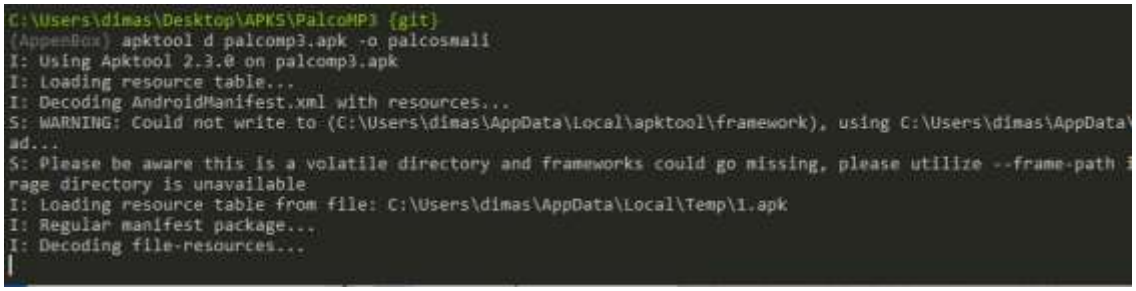
Fonte: O autor.

Também é necessário checar o arquivo AndroidManifest.xml, para ver as informações acerca das permissões do app e também dos provedores de conteúdo e activities. A maneira mais fácil é extraindo o apk com o APKTool. No diretório do

[6] Disponível em: <https://pt.wikipedia.org/wiki/Ataque_man-in-the-middle>
[7] Disponível em: <https://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface(java.lang.Object, java.lang.String)>
[8] Disponível em: <https://blog.trustlook.com/2013/09/04/alert-android-webview-addjavascriptinterface-code-execution-vulnerability/>

apk basta digitar `apktool d [apk] -o [dir]` onde `[apk]` é o nome do arquivo apk e `[dir]` o diretório de saída dos arquivos. A saída do APKTool é mostrada na Figura 32. Após o processo ser completado, basta ir na pasta do APK e verificar o arquivo AndroidManifest.xml com qualquer editor de texto. O Quadro 2 mostra as devidas permissões juntamente à suas descrições.

Figura 32 – Saída do aplicativo APKTool



Fonte: O autor.

Quadro 2 – Permissões do aplicativo PalcoMP3

Permissão	Descrição	Necessária?
INTERNET	Permite aplicações acessarem à internet.	Sim
ACCESS_NETWORK_STATE	Permite acessar informações sobre a rede.	Sim
ACCESS_WIFI_STATE	Permite acessar informações sobre a rede Wi-Fi.	Sim
CHANGE_WIFI_STATE	Permite o aplicativo mudar o status da conexão de Wi-Fi.	Talvez
GET_ACCOUNTS	Permite o acesso à lista de contas no serviço de contas do Google.	Não
ACCESS_COARSE_LOCATION	Permite o app acessar a localização com precisão aproximada.	Sim
ACCESS_FINE_LOCATION	Permite o app acessar a localização com precisão bastante precisa.	Talvez
WRITE_SETTINGS	Permite a aplicação escrever ou ler configurações do sistema.	Não
READ_EXTERNAL_STORAGE	Permite o aplicativo ler do armazenamento externo, como o cartão SD.	Sim
WRITE_EXTERNAL_STORAGE	Permite o aplicativo escrever no armazenamento externo, como o cartão SD.	Sim

Fonte: O autor.

Na tabela das permissões, podemos ver que há permissões que não batem com a descrição do aplicativo. Como o aplicativo utiliza internet, as três primeiras permissões já bastariam, mas o app utiliza a permissão CHANGE_WIFI_STATE provavelmente para lembrar o usuário de ligar a internet caso esteja desligada. A permissão GET_ACCOUNTS é perigosa e não deveria estar listada no aplicativo, já que não se tem sistema de login nem faz o uso da conta Google, então pode estar sendo listada porque o desenvolvedor incluiu o pacote do Google Play services completo, o que não é necessário já que os guias do Google Developers falam em modularizar este pacote. A permissão de localização é utilizada provavelmente para seus provedores de anúncios, que são diversos, porém bastava a ACCESS_COARSE_LOCATION já que o aplicativo não faz uso de mapas ou tem alguma funcionalidade que necessita ter uma maior precisão. No entanto, examinando melhor o código se vê que a permissão ACCESS_FINE_LOCATION é utilizada nos anúncios da API do InLocoMedia, que utiliza da localização para fazer anúncios direcionados. O WRITE_SETTINGS é desnecessário visto que o aplicativo não precisa modificar nada no sistema, como brilho, volume, luzes, entre outros. As últimas duas são necessárias para caso queira salvar dados não sensíveis como playlists. As outras permissões encontradas são internas utilizadas para notificação do FireBase.

4.2.2 PalcoMP3 – Componentes Explícitos

Foi utilizado o Drozer para verificar se havia componentes exportados (Figura 33). A superfície de ataque estava bem promissora, porém a medida que foi se descobrindo os componentes exportados, viu-se que nenhum dado sensível ou janela importante poderia ser atacada.

Figura 33 – Superfície de ataque do app PalcoMP3

```
dz> run app.package.attacksurface com.studiosol.palcomp3
Attack Surface:
  4 activities exported
  5 broadcast receivers exported
  0 content providers exported
  2 services exported
dz>
```

Fonte: O autor.

Comecando dos *services*, rodando o módulo app.service.info, foi possível descobrir que os serviços exportados eram do Firebase, que faz checagens de segurança no próprio servidor da Google para garantir a integridade dos dados, logo não seria um problema (Figura 34).

Figura 34 – Services exportados no PalcoMP3

```
dz> run app.service.info -a com.studiosol.palcomp3
Package: com.studiosol.palcomp3
  com.google.firebase.messaging.FirebaseMessagingService
    Permission: null
  com.google.firebase.iid.FirebaseInstanceIdService
    Permission: null
```

Fonte: O autor.

Logo depois, foram testados as *activities* exportadas (Figura 35), as quais visualizadas código fonte, que estava muito bem ofuscado, não pareceram danosas, até porque o aplicativo não trabalha com dados sensíveis e não tem sistema de login nem de sincronização, é tudo feito localmente.

Figura 35 – Activities exportadas no PalcoMP3

```
dz> run app.activity.info -a com.studiosol.palcomp3
Package: com.studiosol.palcomp3
  com.studiosol.palcomp3.Activities.MainActivity
    Permission: null
  com.studiosol.palcomp3.Activities.PlayerActivity
    Permission: null
  com.studiosol.palcomp3.Activities.ProxyActivity
    Permission: null
  com.studiosol.palcomp3.Activities.ProxyFileActivity
    Permission: null
```

Fonte: O autor.

Por fim, testou-se os *broadcast receivers* (Figura 36) e constatou-se que os três últimos eram *broadcasts* eram referentes aos módulos FireBase e o InMobi (anúncios). Os dois primeiros *receivers* servem para ações dentro do aplicativo, quando o usuário faz o download de uma música (DownloadReceiver), para fazer em background e também para responder os comandos do controle do player de música. Por exemplo, quando o usuário clica play em alguma música, a barra de notificações exibe a música que está sendo tocada no momento (MusicIntentReceiver). Como o código está bastante ofuscado, não houve tanta possibilidade de explorar mais estes *receivers*, mas pela exploração deles no código

Smali (Figura 37) foi possível descobrir que os downloads são armazenados no banco de dados local ao invés da pasta downloads, sendo possível extrair os áudios mesmo que as músicas possuam direitos autorais, por meio do SQLite. A pasta databases fica localizada em /data/data/com.studiosol.palcomp3/databases/ e o banco de dados aberto foi o *palcomp3_smartcache.sqlite*. Os dados são mostrados na Figura 38. Porém, isso não pode ser considerada uma vulnerabilidade, visto que não se sabe quais são as políticas do aplicativo quanto as músicas nele inseridas.

Figura 36 – Broadcasts exportados no PalcoMP3

```
dz> run app.broadcast.info -a com.studiosol.palcomp3
Package: com.studiosol.palcomp3
  com.studiosol.palcomp3.Services.MusicIntentReceiver
    Permission: null
  com.studiosol.palcomp3.Backend.Downloads.DownloadReceiver
    Permission: null
  com.inmobi.commons.core.utilities.uid.ImIdShareBroadCastReceiver
    Permission: null
  com.google.android.gms.measurement.AppMeasurementInstallReferrerReceiver
    Permission: android.permission.INSTALL_PACKAGES
  com.google.firebase.iid.FirebaseInstanceIdReceiver
    Permission: com.google.android.c2dm.permission.SEND
```

Fonte: O autor.

Figura 37 – Trecho de código Smali DownloadReceiver

```
aput-wide p2, v2, v3

invoke-virtual {v1, v2}, Landroid/app/DownloadManager$Query;->setFilterById([J)Landroid/app/DownloadManager$Query;

.line 25
invoke-virtual {v0, v1}, Landroid/app/DownloadManager;->query(Landroid/app/DownloadManager$Query;)Landroid/database/Cursor;

move-result-object v1

.line 27
const/4 v0, 0x0

.line 29
if-eqz v1, :cond_1
```

Fonte: O autor.

Figura 38 – Query SQLite de músicas em mp3

```
sqlite> .tables
android_metadata  playlist_entries  smart_cache_v2
downloads         playlists         smart_songs
sqlite> SELECT * FROM downloads;
5095221|382619|amor-bandido-vakeira-funk|vai-vakeira-vai-funkeira|VakeiraFunk|V
eiraFunk-vai-vakeira-vai-funkeira.mp3
5095222|382619|amor-bandido-vakeira-funk|vakeira-funk|VakeiraFunk|Vakeira funk|

sqlite>
```

Fonte: O autor.

4.2.3 PalcoMP3 – Intent Implícito

O Androbugs detectou o uso de Intent implícito nos serviços `com.google.firebase.messaging.FirebaseMessagingService` além do `com.google.firebase.iid.FirebaseInstanceIdService`. No caso deste aplicativo isso não é um problema, pois segundo orientações do Google Developers (2017), o “FirebaseInstanceIdService executa verificações de segurança em tempo de execução, sem necessidade de permissões explícitas”.

4.2.4 PalcoMP3 - Checagem SSL

Na segunda possível vulnerabilidade, foram detectadas algumas URLs utilizadas no app que não faziam checagem segura com SSL. Segundo a FireEye (2014), dos 1000 aplicativos mais baixados da Google Play até julho de 2014, 68% tinham pelo menos três vulnerabilidades SSL. O mesmo estudo aponta que algumas bibliotecas de anúncios transmitiam informação sensível sobre o usuário para poder criar anúncios direcionados, logo é essencial o uso de HTTPS para evitar este tipo de vazamento.

Como esperado, haviam vulnerabilidades de SSL no aplicativo, que utiliza de diversos servidores de anúncios como Twitter, Facebook, Google, InLocoMedia e MoPub além de conexão com seu próprio servidor na Web. Uma requisição POST no servidor do palcomp3.com é mostrado na Figura 39, expondo o e-mail de forma clara quando o usuário faz o download de alguma música. Também houve a exposição de dados sensíveis no envio do e-mail para algum artista, banda ou até para a empresa, como se pode ver nas Figuras 40 e 41, onde além do e-mail, foram expostos o nome e também a mensagem que se mandaria juntamente com o captcha. Apesar desta mensagem não poder ser utilizada para causar dano algum, um usuário que estivesse escrevendo para um cantor dentro do aplicativo poderia muito bem colocar informações sensíveis como telefone, cachê e outras que poderiam comprometer caso fossem vazadas.

Figura 39 – Requisição HTTP POST para baixar música revelando e-mail.

```
POST /ajax/musica/hit/?plataforma=android&id_mobile=dam4@cin.ufpe.br HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 48
Host: www.palcomp3.com
Connection: close
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.0

[{"m":2,"dt":15130434062221,"id_musica":4590738}]
```

Fonte: O autor.

Figura 40 – Formulário de e-mail do PalcoMP3

Fonte: O autor.

Figura 41 – Requisição HTTP Post com dados sensíveis

Raw | Params | Headers | Hex

```
POST /v2/artistas/mclivinhogr6/contatos/ HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 140
Host: api.palcomp3.com
Connection: close
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.9.0
Authorization: StudioSol UbUHEJk9ZlZsig3mCkURkA
X-Expires: 1513047392893

{"nome":"nome ficticio","email":"email@email-ficticio.com","mensagem":"Olá, este é o corpo da
mensagem","uss":"a7cfd8be","captcha":"3837"}
```

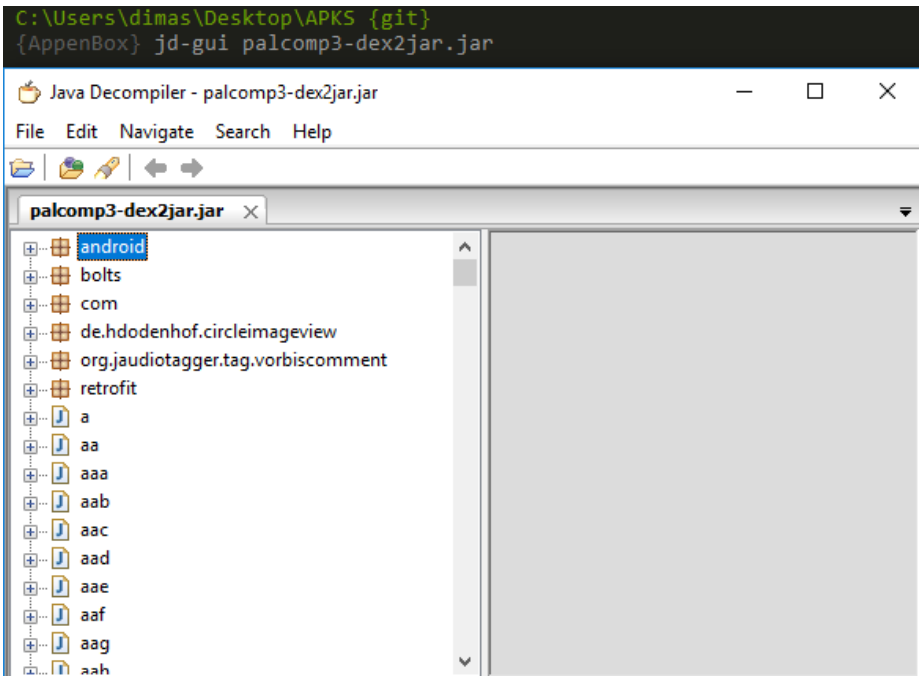
Fonte: O autor.

4.2.5 PalcoMP3 - JavaScript

Para a terceira vulnerabilidade, deve-se fazer análise estática e/ou dinâmica do código. Para análise estática, abriu-se o arquivo .jar na ferramenta JD-GUI para visualização utilizando o comando `jd-gui [jar]`, onde jar é o nome do arquivo .jar gerado na Seção 4.1 (Figura 42). Ao procurar por “addJavascriptInterface”, que é utilizado para injetar código javascript em uma WebView, nos deparamos com trechos de código que confirmam o achado, conforme o seguinte trecho de código: `addJavascriptInterface(new bfv(this), "googleAdsJsInterface").` Este código permite acessar código Java nativo por meio de *Reflection*, que permite criar chamadas em tempo de execução.

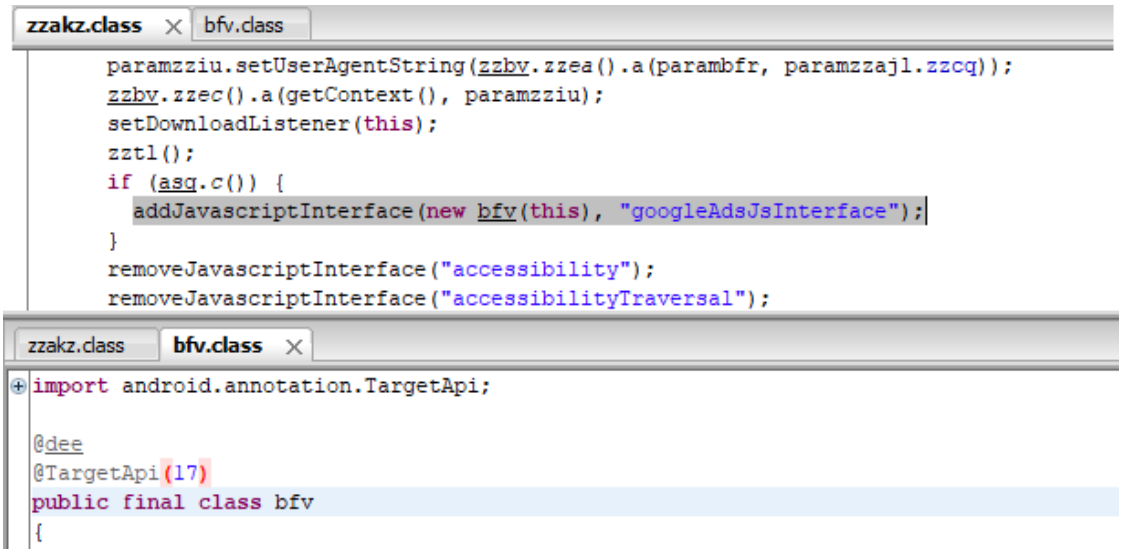
Somado à isso nos deparamos com uma classe ofuscada que foi apelidada de `bfv.class` pelo descompilador, que possui um trecho de código `@TargetApi(17)`, o que confirma que na API 17 (JellyBean) é utilizado o JavaScript, e a mesma contém o *exploit* já conhecido. Na Figura 43, é possível observar alguns destes trechos de código.

Figura 42 – Interface visual do JD-GUI



Fonte: O autor.

Figura 43 – Código parcialmente ofuscado.



Fonte: O autor.

4.2.6 PalcoMP3 - Conclusão

O aplicativo também foi testado em outros pontos como banco de dados sem encriptação, *activities* sem proteção e provedores de conteúdo. Porém, por não ter sistema de login, armazenamento de informações pois apesar de utilizar o SQLite, os downloads são armazenados no próprio telefone, e as *activities* que podem ser abertas a partir de outros apps não tinham relevância. O que impressiona neste aplicativo também é a quantidade de provedores de anúncio, o que acaba por exigir bastante bateria e dados de rede do *smartphone* durante o uso do app. O aplicativo é considerado vulnerável, pois tinha duas vulnerabilidades de nível crítico, uma pode ser realizada por meio do Android 4.2 (JellyBean), já que o requerimento mínimo é esta API.

4.3 Aplicativo Pessoa Física

O aplicativo Pessoa Física, da Receita Federal do Brasil, alguns lugares abaixo do PalcoMP3 junto com o aplicativo da Caixa. O Pessoa física tem mais de um milhão de downloads e serve para consulta de informações cadastrais sobre o CPF, restituição do IRPF, cálculo mensal e anual do imposto, informações gerais além de outras funcionalidades. A versão testada foi a 5.0.0.

4.3.1 Pessoa Física - Relatório de Vulnerabilidades

Utilizando o mesmo padrão do aplicativo passado, utilizou-se o Androbugs para criar um relatório sobre o aplicativo e também o APKtool para verificar as permissões do aplicativo. Do relatório temos algumas possíveis vulnerabilidades críticas:

1. **Fragment Injection**: esta vulnerabilidade só foi corrigida no Android 4.4, e permite que qualquer aplicativo possa carregar código arbitrário explorando o processo dinâmico de criação de um fragment[9];
2. **Proteção de KeyStore:** a aplicação utiliza o KeyStore para manusear certificados. É possível que exista informação sobre o certificado em forma de texto ou cadeias de bytes no código;
3. **Permissão em SharedPreferences:** o app utiliza as SharedPreferences para armazenar informações. Caso elas tenham dados sensíveis e se a permissão estiver em MODE_WORLD_READABLE ou MODE_WORLD_WRITEABLE, qualquer aplicativo pode ler a informação (MUELLER; SCHLEIER, 2017);
4. **Checagem de SSL:** a mesma vulnerabilidade do aplicativo anterior, porém há mais possíveis endereços URL.

Partindo para as permissões do app, de fato este aplicativo apenas aplica o mínimo de permissões necessárias. A única permissão que deveria ser retrabalhada é a GET_ACCOUNTS, que cai no mesmo caso do aplicativo passado, de envolver o uso de todo o Google Play Services. Temos na Tabela 4 que:

Quadro 3 – Permissões do aplicativo Pessoa Física

Permissão	Descrição	Necessária?
INTERNET	Permite aplicações acessarem à internet.	Sim
ACCESS_NETWORK_STATE	Permite acessar informações sobre a rede.	Sim
GET_ACCOUNTS	Permite o acesso à lista de contas no serviço de contas do Google.	Não
READ_EXTERNAL_STORAGE	Permite o aplicativo ler do armazenamento externo, como o cartão SD.	Sim
WRITE_EXTERNAL_STORAGE	Permite o aplicativo escrever no armazenamento externo, como o cartão SD.	Sim
WAKE_LOCK	Permissão para manter a tela do aparelho	SIM

[9] Disponível em: <http://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf>

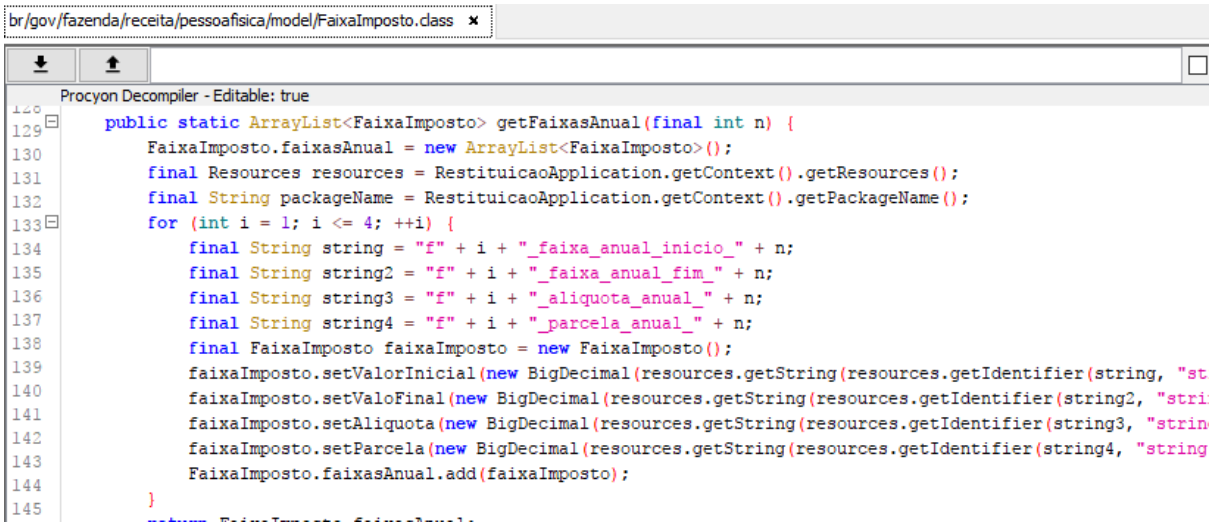
Permissão	Descrição	Necessária?
	ligada quando necessário.	
VIBRATE	Permissão para fazer o aparelho vibrar quando necessário.	SIM

Fonte: O autor.

4.3.2 Pessoa Física - Ofuscação de Código e Testes Rápidos

Com o arquivo .jar já criado com o d2j-dex2jar, desta vez foi utilizado o Bytecode Viewer para descompilação do .jar. Basta digitar o comando `bytecode` no console do AppenBox, e abrir o arquivo desejado. Viu-se que a ofuscação de código do aplicativo Pessoa física é fraca, de modo que conseguiu-se recuperar diversas classes inteiras como se vê na Figura 44.

Figura 44 – Exemplo de código não ofuscado.



Fonte: O autor.

4.3.3 Pessoa Física - Fragment Injection

Para verificar se a vulnerabilidade é passível de uso por hackers maliciosos, utilizando o Drozer, foram procurados Activities exportados utilizando o comando `run app.activity.info -a [package]` onde [package] é o nome do pacote. Como o resultado visto na Figura 45 mostra que não havia nenhuma Activity vulnerável a este tipo de ataque, assim foi descartada a possibilidade deste ataque.

Figura 45 – Informações acerca de Activities exportadas, no Drozer.

```
drozer Console (v2.4.4)
dz> run app.activity.info -a br.gov.fazenda.receita.pessoafisica
Package: br.gov.fazenda.receita.pessoafisica
  br.gov.fazenda.receita.pessoafisica.ui.activity.SplashActivity
    Permission: null

dz>
```

Fonte: O autor.

Mas para saber de onde veio esta hipótese de vulnerabilidade no relatório, voltando ao Bytecode Viewer, foi procurado pela String “PreferenceActivity” e foi achado a causa do problema. Um módulo chamado roboguice, que serve para injeção de dependências no código e continha uma activity que era passível de *fragment injection*, que foi corrigido na versão 19 (KitKat). Ainda assim, o roboguice encontra-se disponível no código do aplicativo apesar de aparentemente não ser utilizado. O módulo foi aposentado em 2016 e felizmente não pode ser utilizado para explorar a vulnerabilidade, entretanto ocupa um espaço inútil dentro do APK.

Figura 46 – Módulo Roboguice ainda ativo

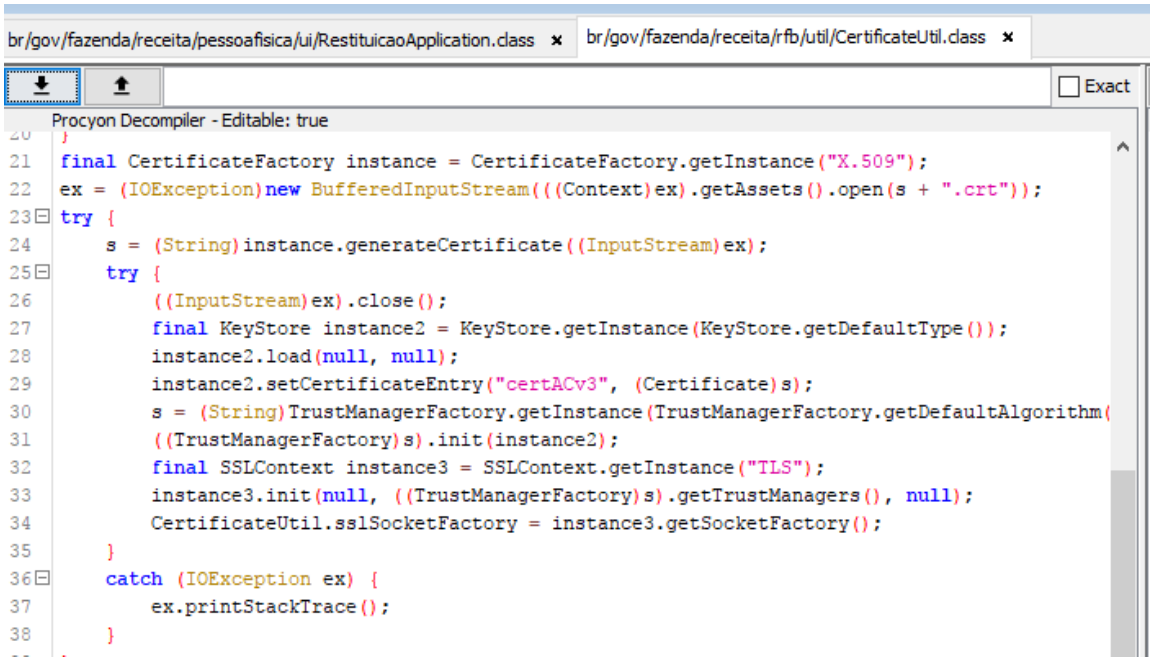
```
import android.preference.*;
import roboguice.inject.*;

public abstract class RoboPreferenceActivity extends PreferenceActivity implements RoboContext
{
    protected EventManager eventManager;
    @Inject
    ContentViewListener ignored;
    protected PreferenceListener preferenceListener;
    protected HashMap<Key<?>, Object> scopedObjects;
```

Fonte: O autor.

4.3.4 Pessoa Física - KeyStore

Os aplicativos usam o KeyStore para armazenagem chaves públicas e assim combater técnicas de *Man-In-The-Middle*, já citadas anteriormente. O processo é chamado de *Public Key Pinning* e além do cliente verificar informações sobre o certificado que foi enviado pelo servidor, também se verifica no servidor se a chave pública existente no certificado confere com a esperada para esse domínio (SOUZA, 2015).

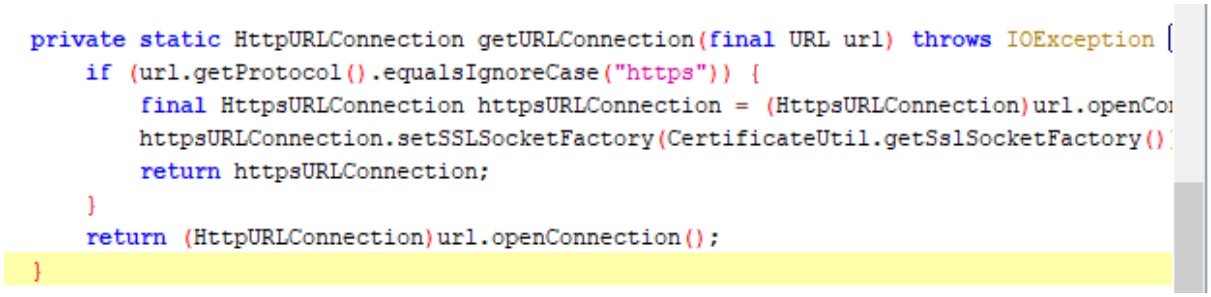
Figura 47 – Geração de certificados no aplicativo Pessoa Física



Fonte: O autor.

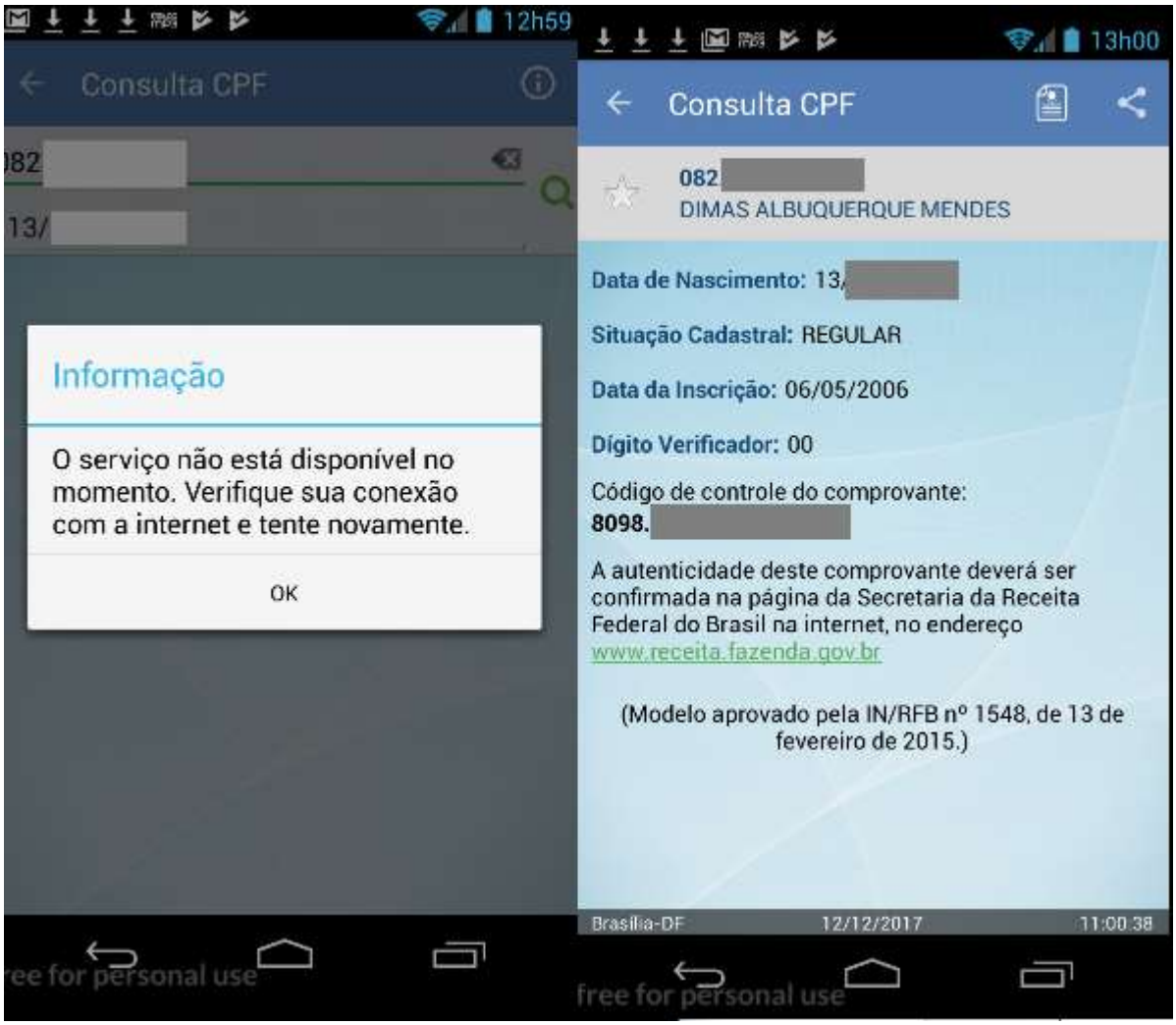
Procurando pela String “loadCertificates” no Bytecode Viewer, encontrou-se a classe `RestituicaoApplication` pertencente à `CertificateUtil` (Figura 41). O código apresentou corretamente a utilização do certificado TLS (versão atual do SSL) na conexão e utiliza sempre HTTPS no requerimento segundo a classe `URLConnectionUtil` (Figura 48). Quando se tenta consultar informação como por exemplo um CPF utilizando o Burp Suite como Proxy, o aplicativo retorna uma mensagem de erro de servidor indisponível. Já quando se tenta abrir a mesma consulta no smartphone sem o proxy, funciona normalmente, pois o servidor só aceita os certificados que foram gerados para o aplicativo, como demonstrado pela Figura 49.

Figura 48 – Trecho de código que demonstra o uso do HTTPS no Pessoa Física



Fonte: O autor.

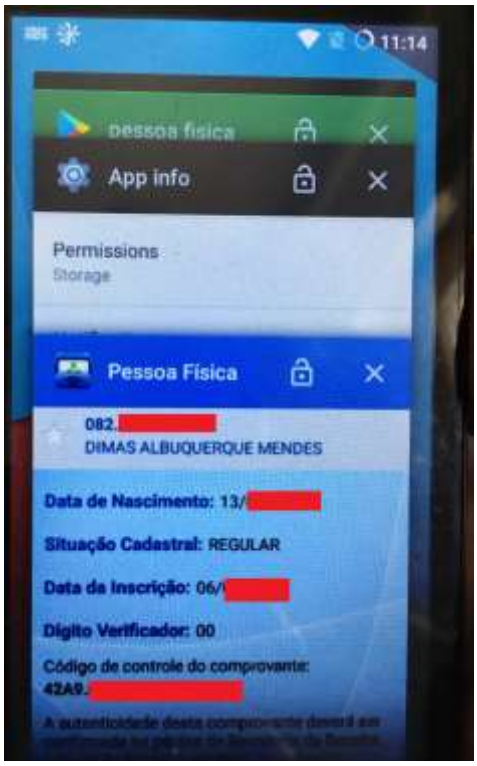
Figura 49 – Consulta com Burp Proxy, à esquerda, e sem o Burp proxy, à direita.



Fonte: O autor.

Entretanto, apesar da proteção SSL não se usa a proteção da Activity que está armazenando a informação, assim não impede que outro app tire uma foto da tela de maneira que os dados sensíveis seriam expostos (Figura 50). Para evitar isso, deve-se adicionar a FLAG_SECURE na Activity proposta, assim ela ficará ofuscada e não poderá ser fotografada.

Figura 50 – Activity não protegida mostrando dados sensíveis com o app em background.



Fonte: O autor.

4.3.5 Pessoa Física - SharedPreferences e SSL Connection

Para verificação do SSL *connection*, bastou ver se as principais requisições do aplicativo estavam utilizando HTTPS, pois assim o certificado visto na seção anterior irá funcionar. Assim, não existem brechas de SSL para dados sensíveis.

Como há o armazenamento de dados em SharedPreferences, bastou verificar se estes estavam utilizando o MODE_WORLD_PRIVATE (de valor 0) de armazenamento, assim livrando usuários comuns de ler o arquivo de preferências da aplicação. Tanto pelo código Java, como pela pasta do aplicativo com o *shell* do ADB, pudemos confirmar que a leitura do arquivo não é permitida por usuários comuns.

Figura 51 – Verificação de permissões de acesso no diretório onde ficam as SharedPreferences.

```
55  public static SharedPreferences getGCMPreferences(final Context context) {
56      return context.getSharedPreferences(GCM.TAG, 0);
57  }
```

```
-rw-rw---- u0_a39    u0_a39        412 2017-12-12 09:49 RestituicaoApplication.xml
-rw-rw---- u0_a39    u0_a39        432 2017-12-12 10:44 com.google.android.gms.analytics.prefs.xml
root@android:/data/data/br.gov.fazenda.receita.pessoafisica/shared_prefs #
```

Fonte: O autor.

4.3.6 Pessoa Física - Conclusão

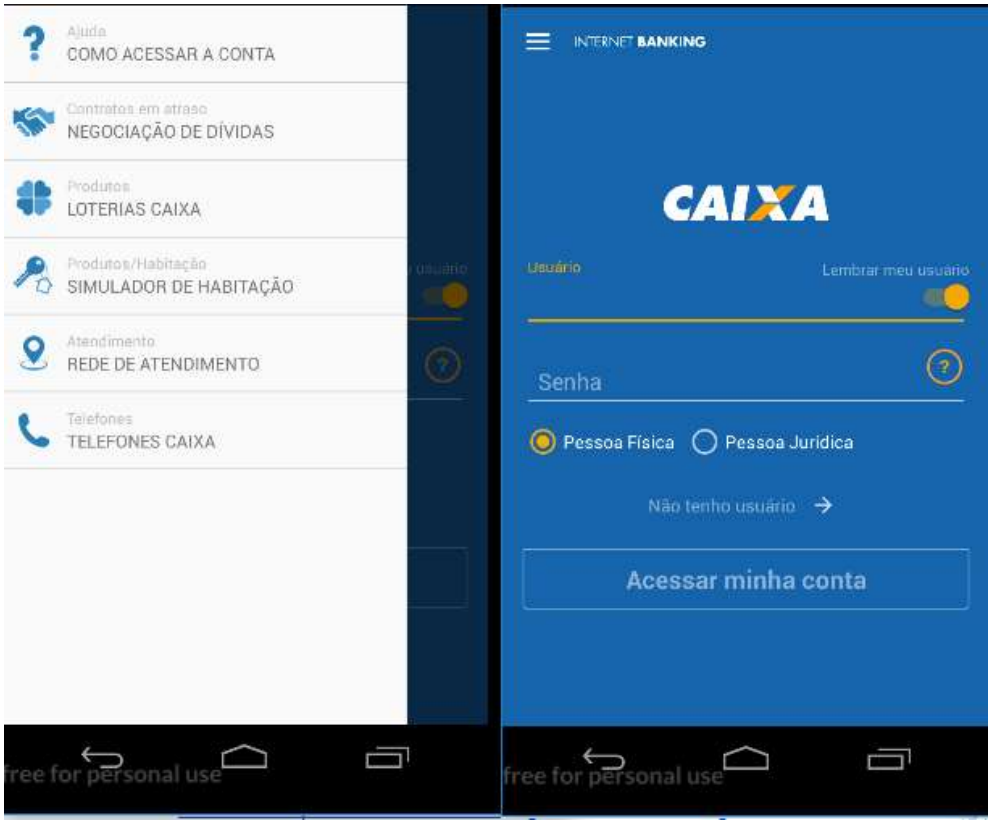
O aplicativo Pessoa Física, apesar de ser um código relativamente antigo, mas em atualização, demonstrou diversos meios para assegurar o não vazamento de dados sensíveis e pode-se dizer que o aplicativo não é vulnerável, apesar da questão da fotografia de tela, que é um problema resolvido em uma linha de código.

4.5 Aplicativo Caixa Econômica Federal

O aplicativo Caixa, da Caixa Econômica Federal possui mais de dez milhões de downloads para clientes do banco poderem acessar sua conta sem sair de casa, contanto com assinatura digital para transações financeiras além pagamentos, investimentos e todo o necessário de um banco. A versão testada foi a 3.2.3.

Como desta vez não se teve total acesso ao aplicativo, o teste foi limitado as janelas correspondentes aos menus e tentativas de burlar a segurança. A tela inicial e o menu são mostrados na figura abaixo:

Figura 52 – Tela inicial e menu gaveta do app Caixa.



Fonte: O autor.

4.5.1 Caixa - Relatório de Vulnerabilidades

Ao rodar o Androbugs para se ter o relatório geral, o APKTool para se pegar as permissões, activities e mais detalhes, e o d2j-dex2jar, das possíveis vulnerabilidades encontradas, foi descartada uma das vulnerabilidades críticas, a Runtime.getRuntime().exec("..."), pois quando procurou-se pelo comando no

Bytecode Viewer os códigos não eram prejudiciais. Logo as possíveis vulnerabilidades foram:

1. **KeyStore e SSL Checking**: estão juntas pois uma depende da outra e caso não houver geração de certificado próprio, seria bem passível de falhas no SSL;
2. **Execução de código remoto em WebView:** outro aplicativo com esta vulnerabilidade, gerada pelo uso de JavaScript em uma WebView para controlar a aplicação.

As permissões que foram listadas podem ser encontradas no quadro abaixo:

Quadro 4 – Permissões do aplicativo Caixa.

Permissão	Descrição	Necessária?
INTERNET	Permite aplicações acessarem à internet.	Sim
ACCESS_NETWORK_STATE	Permite acessar informações sobre a rede.	Sim
ACCESS_WIFI_STATE	Permite acessar informações sobre a rede Wi-Fi.	Sim
CAMERA	Permite o acesso à camera.	Sim
ACTION_MANAGE_OVERLAY_PERMISSION	Permite o controle de que apps irão ser desenhados por cima de outros.	Talvez
READ_PHONE_STATE	Permite acesso de leitura à situação do smartphone, incluindo número de telefone, informação de rede atual, chamadas em andamento e outros.	Talvez
SYSTEM_ALERT_WINDOW	Permite o aplicativo desenhar janelas sobre outros apps.	Não
USE_FINGERPRINT	Permite a leitura biométrica.	Sim
VIBRATE	Permite o aparelho vibrar.	Sim
WRITE_EXTERNAL_STORAGE	Permite o aplicativo escrever no armazenamento externo, como no cartão SD.	Sim

Fonte: O autor.

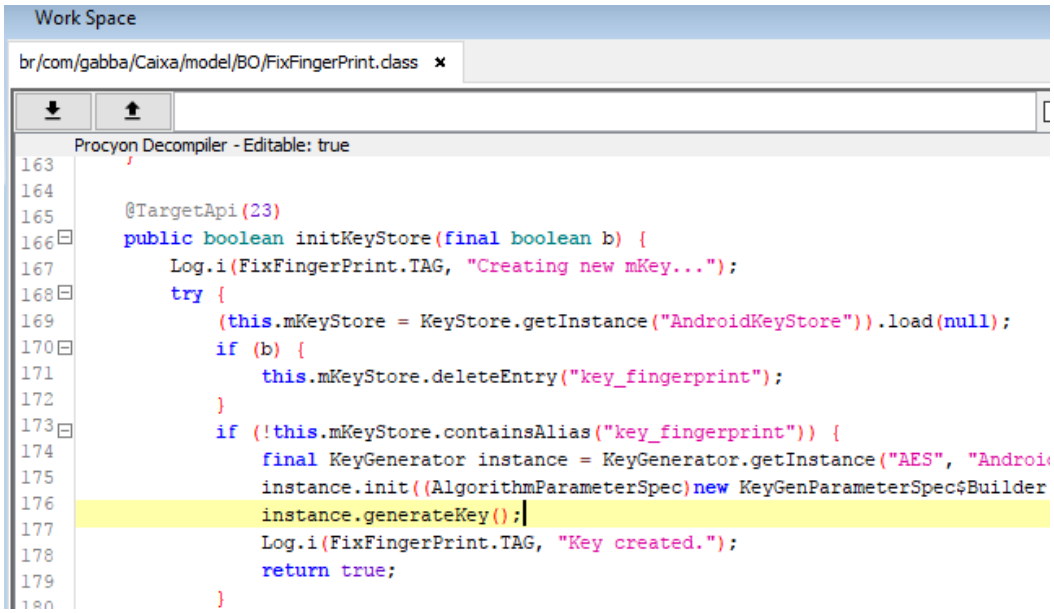
Sobre as permissões, o Caixa não chega a fugir das funcionalidades a que foi proposto, exceto pela permissão SYSTEM_ALERT_WINDOW, que até a versão do Android 6.0 era informada ao usuário no momento da instalação do App. Já a

permissão READ_PHONE_STATE pode ser utilizada para identificar o telefone através do IMEI, como uma espécie de segurança.

4.5.2 Caixa - KeyStore e SSL Checking

Com o Bytecode Viewer procurou-se pela String KeyStore, e pôde-se observar que ela é usada para validar o login por biometria e não apresenta problemas. A Figura 53 demonstra este uso.

Figura 53 – Classe FingerPrint utilizando KeyStore



Fonte: O autor.

Também não há nenhuma ou quase nenhuma ofuscação no código, pois recuperou-se praticamente o código original. Com isso fica mais fácil de encontrar vulnerabilidades. Somado a isso partes da aplicação utilizaram código HTTP para acessar o servidor, assim tornando-os vulnerável a possíveis ataques pela falta do SSL ou quaisquer travas de segurança.

Figura 54 – Protocolo HTTP sendo utilizado.

```
lic class BannerBO

private static final String BANNER_SERVICE_URL = "http://www.caixa.gov.br/_api/lists/GetByTitle('App_C
@Pref
CaixaPreferences_ caixaPreferences;
@RootContext
Context context;

private String getServerBanners() {
    final StringBuilder sb = new StringBuilder();
    try {
        final HttpURLConnection httpURLConnection = (HttpURLConnection)new URL("http://www.caixa.gov.b
        httpURLConnection.setRequestProperty("Accept", "application/json;odata=verbose");
        httpURLConnection.setRequestProperty("cookie", "security=true");
        httpURLConnection.setRequestMethod("GET");
        httpURLConnection.setDoInput(true);
        httpURLConnection.connect();
        final BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(httpURLConnecti
        while (true) {
            final String line = bufferedReader.readLine();
            if (line == null) {
                break;
            }
            sb.append(line);
```

Fonte: O autor.

4.5.3 Caixa - Vulnerabilidade JavaScript

Assim como o aplicativo PalcoMP3, este app também é vulnerável ao ataque, mas de forma mais explícita por ter mais vetores de ataque (mais WebViews espalhados), o que aumenta a superfície de ataque. Utilizou-se primeiramente o Pidcat, o registrador de *logs* do sistema com interface colorida, para ver se alguma informação seria extraída da navegação dentro do app. Para lançar o Pidcat, basta o comando `pidcat [package]` onde [package] é o nome do pacote a ser lido. Figura 55 mostra um destes logs do sistema demonstrando o uso do JavaScript.

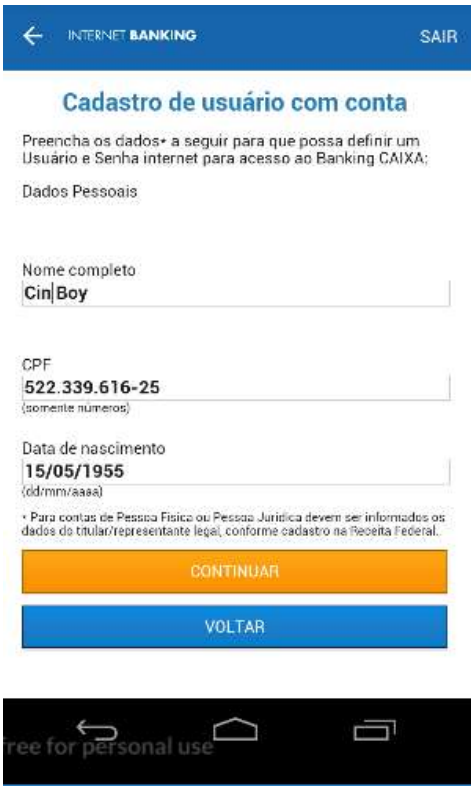
Figura 55 – Erro demonstrando o uso do JavaScript, no Pidcat.

```
            webviewglue  D  nativeDestroy view: 0xb9dce940
            Web Console  E  Unsafe JavaScript attempt to ac
cess frame with URL http://www.caixa.gov.br/comoacessar/Pag
inas/default.aspx from frame with URL https://www.youtube.c
om/embed/oCai4L94dp0. Domains, protocols and ports must mat
ch.
                            E  at null:1
```

Fonte: O autor.

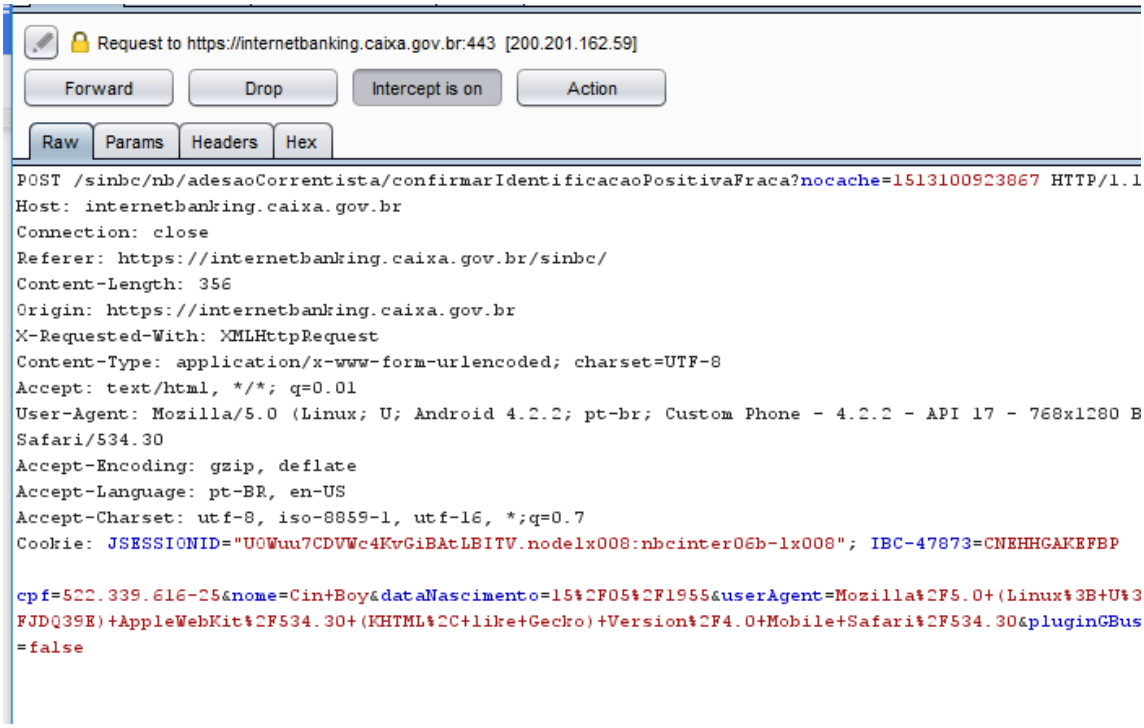
A partir deste ponto, já é uma prova suficiente para não precisar vasculhar mais o código. Utilizando o Burp Suite como proxy com interceptação ligada foi possível navegar e testar diversas tentativas de *exploit*. Seguindo para página de cadastro do aplicativo, foi possível utilizar a página de erro para injetar um código JavaScript que foi executado dentro do app. Foram utilizadas informações de cadastro falsas e um gerador de CPF válido para validar a entrada. As Figuras 56-59 demonstram o exploit ocorrendo por meio da injeção do código `<script>alert('CInjected! :D);</script>`.

Figura 56 – Tela de cadastro do aplicativo Caixa.



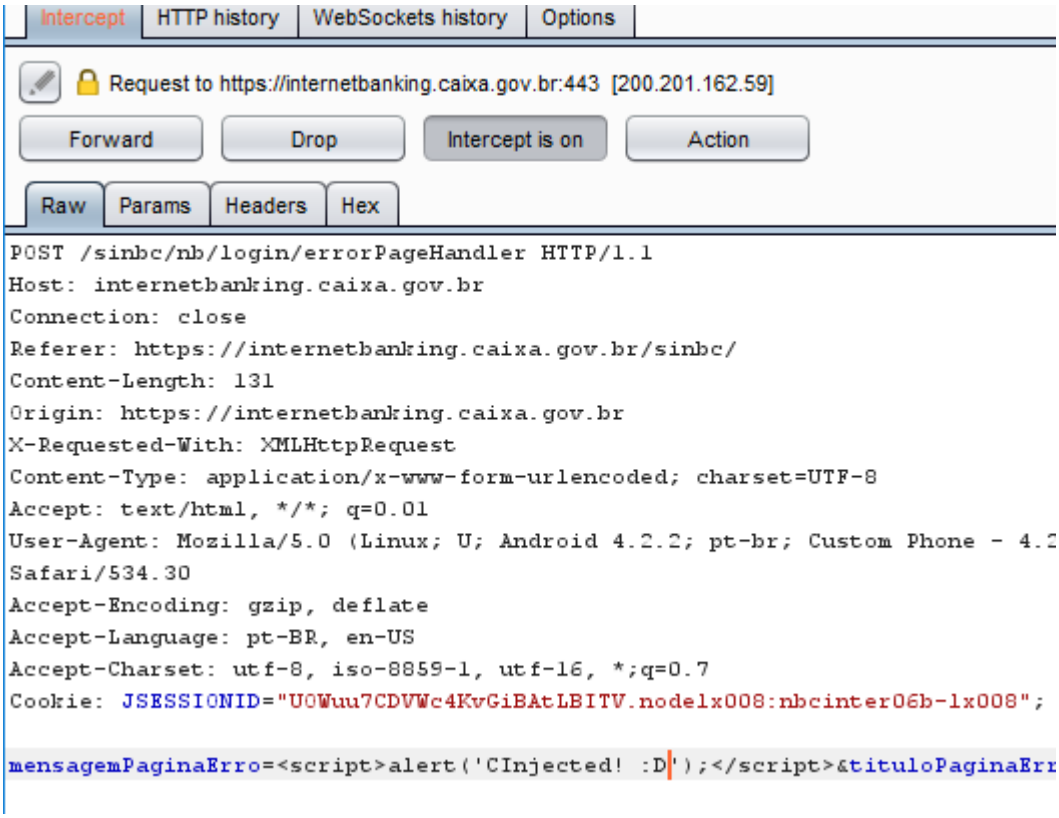
Fonte: O autor.

Figura 57 – Requisição POST referente à Figura 56.



Fonte: O autor.

Figura 58 – Erro no cadastro, conta inexistente no Banco, injetando JavaScript.



Fonte: O autor.

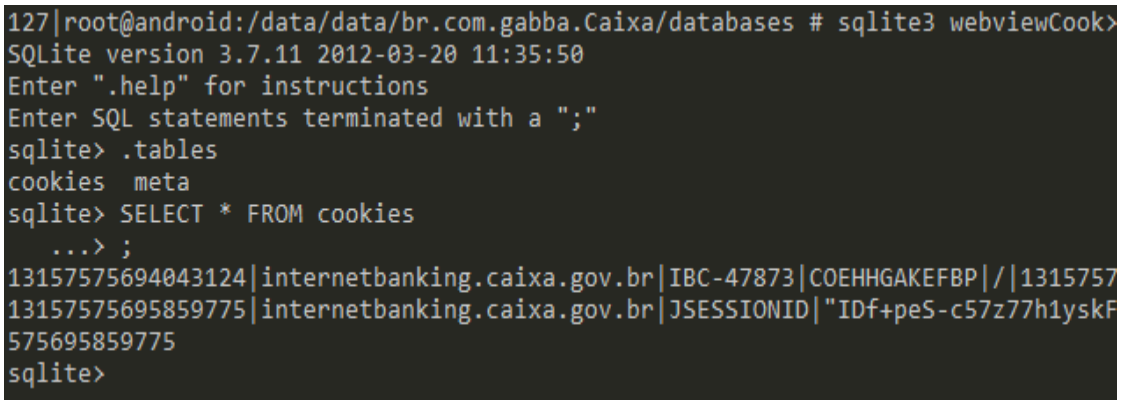
Figura 59 – JavaScript Injection no aplicativo Caixa.

Fonte: O autor.

4.5.4 Caixa - Outras Vulnerabilidades

O aplicativo Caixa também conta com outra vulnerabilidade que é de nível *low* (baixo, segundo critérios da OWASP) por ser difícil de ser explorada por alguém que não tem acesso ao smartphone do usuário, ou seja, sua superfície de ataque é diminuída. No diretório **/data/data/br.com.gabba.Caixa/databases** é salvo o número de identificação da sessão em um banco de dados não criptografado e que não possui senha alguma. Supondo que este número, na lógica da aplicação, fosse utilizado para definir se um usuário estiver autenticado ou não, comparando ao número da sessão no servidor, um atacante poderia roubar a sessão, contanto que ainda estivesse válida no momento em que ele obtivesse o acesso ao ID e inserisse em seu próprio banco de dados ou fizesse uma requisição ao servidor do app.

Figura 60 – Dados de sessão salvos em banco de dados sem nenhuma proteção.



Fonte: O autor.

4.5.5 Caixa - Conclusão

Dentre os aplicativos anteriores o Caixa é o que tinha mais vetores de ataque disponíveis para o hacker mal-intencionado. Isso porque só foi testado a parte inicial da aplicação até o login. Este aplicativo pode ser considerado como muito vulnerável.

5 CONSIDERAÇÕES FINAIS

Com o presente trabalho foi possível identificar e analisar vulnerabilidades existentes em aplicativos do sistema operacional Android, conforme os padrões OWASP e o auxílio de ferramentas de *pentesting*. A identificação de vulnerabilidades é um processo longo, trabalhoso e contínuo, o que requer uma atenção especial principalmente no processo de desenvolvimento, no qual o desenvolvedor tende a negligenciar os padrões de segurança para um desenvolvimento mais rápido e barato.

A criação do *software* implementado para este projeto, o AppenBox, foi essencial neste sentido, facilitando e otimizando o tempo de testes de cada aplicativo por ser completo, objetivo e independente, permitindo que o desenvolvedor tenha um ambiente completo de testagem de vulnerabilidades que pode ser carregado facilmente em um pendrive.

Foi possível observar por meio dos resultados utilizando a metodologia de testagem desenvolvida para este trabalho, que mesmo em aplicativos criados por empresas de grande porte e baixados por milhões de pessoas ainda há problemas, sejam eles mínimos até críticos, o que pode causar desde um vazamento de dados sensíveis até um acesso remoto ao servidor da aplicação e assim prejudicar os usuários que querem usufruir de todos os recursos com plena confiança de que seus dados estejam seguros. Todos estes problemas poderiam ser evitados caso fossem seguidos os padrões de desenvolvimento.

Apesar de contratempos na implementação, adaptação da metodologia e o curto espaço para demonstrar o uso de todas as ferramentas possíveis, foi possível realizar uma análise bastante compreensível de aplicações reais com uma metodologia de qualidade. Com isso, ficou evidente que todos os objetivos propostos foram realmente alcançados.

As explicações teóricas arquiteturais e conceituais dos livros lidos em conjunto com as metodologias adotadas e adaptadas foram essenciais na maneira de criar um referencial teórico de qualidade, lidar e categorizar as ameaças, criar uma ferramenta ideal e detectar os problemas.

Dada à importância do tema que vem crescendo a cada ano, é necessário o investimento contínuo em ferramentas *open source* e projetos visem a segurança de

smartphones, que são meios de comunicação individual que adentram todos os ambientes, desde casas até empresas.

Nesse sentido, a utilização de ferramentas e *softwares* dentro do ambiente de desenvolvimento Android permite aos desenvolvedores tornarem suas aplicações mais seguras, levando o profissional a seguir mais padrões de segurança e zelar mais por seus futuros usuários.

Para trabalhos futuros, sugere-se a criação de novas ferramentas de descompilação e análise completa de aplicações, visto que as que já existem estão desatualizadas em relação as novidades que o sistema trouxe com as novas versões do Android e principalmente com a chegada do Kotlin como linguagem oficial do sistema, o que vai necessitar de um esforço por parte dos desenvolvedores até se adaptarem a produzir códigos com bastante qualidade. Outras metodologias podem surgir ou ser adaptadas de outra afim de melhorar o sistema e enriquecer os resultados na relação. Por fim, deve-se repetir estes testes com os mesmos aplicativos a fim de verificar se no futuro estes problemas já foram corrigidos, e com outros aplicativos para construir uma comunidade de desenvolvimento cada vez mais saudável.

REFERÊNCIAS

AGRAWAL, A. Manifest Security. **Android Application Security Series**, 2015. Disponivel em: <https://manifestsecurity.com/android-application-security/>. Acesso em: 15 out. 2017.

APACHE FOUNDATION. Apache. **Apache Software Foundation**, 2017. Disponivel em: <http://www.apache.org>. Acesso em: 20 out. 2017.

BRAHLER, S. Karlsruhe institute for technology. **Analysis of the android architecture**, jun. 2010. Disponivel em: <https://os.itec.kit.edu/downloads/sa_2010_braehler-stefan_android-architecture.pdf>. Acesso em: 25 out. 2017.

CONSTANTIN, L. PC World. **VPN bypass vulnerability affects Android Jelly Bean and KitKat, researchers say**, 2014. Disponivel em: <https://www.pcworld.idg.com.au/article/536919/vpn_bypass_vulnerability_affects_android_jelly_bean_kitkat_researchers_say/>. Acesso em: 13 set. 2017.

CURPHEY, M.; GROVES, D. OWASP. **About The Open Web Application Security Project**, 2006. Disponivel em: <https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project>. Acesso em: 17 set. 2017.

DUBEY, A.; MISRA, A. **Android security:** attacks and defenses. [S.l.]: CRC Press, 2013.

FIRE EYE. FireEye Blog. **SSL Vulnerabilities:** Who listens when Android applications talk?, 2014. Disponivel em: <https://www.fireeye.com/blog/threat-research/2014/08/ssl-vulnerabilities-who-listens-when-android-applications-talk.html>. Acesso em: 15 nov. 2017.

GEORGIEV, A. B.; SILLITTI, A.; SUCCI, G. **Open Source Mobile Virtual Machines:** An Energy Assessment of Dalvik vs. ART. IFIP International Conference on Open Source Systems. Heidelberg, Berlim: Springer. 2014. p. 93-102.

GOOGLE. Android Open Source Project. **About the Android Open Source Project**, 2017. Disponivel em: <https://source.android.com/>. Acesso em: 01 out. 2017.

GOOGLE DEVELOPERS. Developers. **Google Developers**, 2017. Disponivel em: <https://developer.android.com/index.html>. Acesso em: 01 out. 2017.

GUPTA, A. **Learning Pentesting for Android Devices**. [S.l.]: [s.n.], 2014.

HERZOG, P. Institute For Security And Open Techonologies - Isecom. **The Open Source Security Testing Methodology Manual:** Contemporary Security Testing and Analysis, 2010. Disponivel em: <http://www.isecom.org/mirror/OSSTMM.3.pdf>. Acesso em: 15 out. 2017.

IDC. IDC. **Smartphone OS Market Share, 2017 Q1**, 2017. Disponivel em: <http://www.idc.com/promo/smartphone-market-share/os>. Acesso em: 13 set. 2017.

IETF. IETF. **Internet Engineering Task Force RFC 2828 Internet Security Glossary**, maio 2000. Disponivel em: <https://tools.ietf.org/html/rfc2828>. Acesso em: 20 set. 2017.

INFOSEC INSTITUTE. Infosec Institute. **Android Hacking Security**, 2014. Disponivel em: <http://resources.infosecinstitute.com/android-hacking-security-part-1-exploiting-securing-application-components/#gref>.

LECHETA, R. R. **Google Android:** Aprenda a criar aplicações para dispositivos móveis com o Android SDK. 5ª Edição. ed. [S.l.]: Novatec Editora, 2015.

MUELLER, B.; SCHLEIER, S. **OWASP Mobile Security Testing Guide**. Early Access. ed. [S.l.]: OWASP Foundation, 2017. Disponivel em: <https://github.com/OWASP/owasp-mstg>.

NORTH, Z. Office Space Software. **How Office Managers Can Make the Most of Smartphone Habits in the Workplace**, 2017. Disponivel em: <https://www.officespacesoftware.com/blog/how-office-managers-can-make-the-most-of-smartphone-habits-in-the-workplace>. Acesso em: 10 set. 2017.

OHA. OHA. **Open Handset Alliance**, 2017. Disponivel em: <https://www.openhandsetalliance.com/>. Acesso em: 20 set. 2017.

PELES, O.; HAY, R. SecurityIntelligence. **One Class to Rule Them All:** New Android Serialization Vulnerability Gives Underprivileged Apps Super Status, 2015. Disponivel em: <http://securityintelligence.com/one-class-to-rule-them-all-new-android-serialization-vulnerability-gives-underprivileged-apps-super-status/>. Acesso em: 20 set. 2017.

PEREZ, S. TechCrunch. **Google Play installs reached 65 billion last year**, 2016. Disponivel em: <https://techcrunch.com/2016/05/18/google-play-installs-reached-65-billion-last-year/>. Acesso em: 23 set. 2017.

PISCITELLO, D. ICANN. **Threats, Vulnerabilities and Exploits – oh my!**, 10 ago. 2015. Disponivel em: <https://www.icann.org/news/blog/threats-vulnerabilities-and-exploits-oh-my>. Acesso em: 20 set. 2017.

QUERINO FILHO, L. C. **Desenvolvendo seu Primeiro Aplicativo Android:** Entre de cabeça no mundo dos aplicativos móveis, criando e publicando seu próprio programa para o sistema líder do mercado! 2ª. ed. [S.l.]: Novatec Editora, 2013.

ROUSE, M. WhatIs. **What is sensitive information?**, 2014. Disponivel em: <http://whatis.techtarget.com/definition/sensitive-information>. Acesso em: 20 nov. 2017.

SOUZA, A. P. Cryptoid. **O que é o Certificate ou Public Key Pinning e qual a sua importância**, 2015. Disponivel em: <https://cryptoid.com.br/banco-de-noticias/o-que-e-o-certificate-ou-public-key-pinning-e-qual-a-sua-importancia/>. Acesso em: 18 nov. 2017.

TAG. Threat Analysis Group. **Threat, vulnerability, risk – commonly mixed up terms**, 2015. Disponivel em: <https://www.threatanalysis.com/2010/05/03/threat-vulnerability-risk-commonly-mixed-up-terms/>. Acesso em: 20 set. 2017.

TUNG, L. ZDNet. **Android security a 'market for lemons' that leaves 87 percent vulnerable**, 2015. Disponivel em: <http://www.zdnet.com/article/android-security-a-market-for-lemons-that-leaves-87-percent-insecure/>. Acesso em: 13 set. 2017.

ZAHARIA, A. Heimdal Security. **The Most Vulnerable Software in 2016 (and Why Updates Are Fundamental)**, 2017. Disponivel em: <http://heimdalsecurity.com/blog/most-vulnerable-software-2016/>. Acesso em: 13 set. 2017.