MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

RICARDO LUIS DIAS MARTINS FERREIRA

UM MÉTODO PROGNÓSTICO PARA DESENVOLVIMENTO SEGURO DE APLICATIVO

Rio de Janeiro
2016

INSTITUTO MILITAR DE ENGENHARIA

RICARDO LUIS DIAS MARTINS FERREIRA

UM MÉTODO PROGNÓSTICO PARA DESENVOLVIMENTO SEGURO DE APLICATIVO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. Anderson Fernandes P. dos Santos - D.Sc.
Co-Orientador: Prof. Ricardo Choren Noya - D.Sc.

Rio de Janeiro
2016

c2016

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

004.69 Ferreira, Ricardo Luis Dias Martins
F383m Um método prognóstico para desenvolvimento seguro de aplicativo / Ricardo Luis Dias Martins Ferreira, orientado por Anderson Fernandes P. dos Santos e Ricardo Choren Noya - Rio de Janeiro: Instituto Militar de Engenharia, 2016.

72p.: il.

Dissertação (mestrado) - Instituto Militar de Engenharia, Rio de Janeiro, 2016.

1. Curso de Sistemas e Computação - teses e dissertações. 1. Android. 2. Aplicações Móveis. 3. Segurança da Informação. 4. Vulnerabilidade. 5. Classificação de Vulnerabilidade. I. dos Santos, Anderson Fernandes P. . II. Noya, Ricardo Choren. III. Título. IV. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

RICARDO LUIS DIAS MARTINS FERREIRA

UM MÉTODO PROGNÓSTICO PARA DESENVOLVIMENTO SEGURO DE APLICATIVO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. Anderson Fernandes P. dos Santos - D.Sc.

Co-Orientador: Prof. Ricardo Choren Noya - D.Sc.

Aprovada em 09 de Dezembro de 2016 pela seguinte Banca Examinadora:

Prof. Anderson Fernandes P. dos Santos - D.Sc. do IME - Presidente

---

Prof. Ricardo Choren Noya - D.Sc. do IME

Prof^{a}. Raquel Coelho Gomes Pinto - D.Sc. do IME

Prof. Fábio Borges de Oliveira - Dr.-Ing. do LNCC

Rio de Janeiro
2016

Dedico este trabalho aos meus pais Roberto (*in memorian*) e Maria Dolores pelo sacrifício empregado para que eu recebesse a melhor educação possível, e pelos princípios éticos que me ensinaram ao longo desta vida.

Este trabalho também é dedicado à minha esposa Anna Paola e aos meus filhos Anna Luisa e Henrique, por me apoiarem nesta jornada, como sempre fizeram em nosso convívio familiar, me suportando e amparando nos momentos mais difíceis.

AGRADECIMENTOS

Agradeço aos meus professores orientadores Dr. Anderson Fernandes Pereira dos Santos e Dr. Ricardo Choren Noya pelo incentivo e por mostrar-me o caminho correto a seguir, de forma admirável e exemplar para que o objetivo final deste trabalho fosse atingido.

Agradeço, da mesma forma, à professora Dr^a. Raquel Coelho Gomes Pinto e ao professor Dr. Fábio Borges de Oliveira pela honra em aceitar o convite para fazer parte da distinta Banca Examinadora desta dissertação.

Aos meus familiares, em especial às minhas primas Márcia Ramos e Ana Ridel, por revisar os textos produzidos em língua inglesa.

Por fim, ao Instituto Militar de Engenharia por proporcionar-me esta oportunidade e aos meus amigos e a todos os professores que fazem parte desta instituição por, de alguma forma, ajudar-me na formação de ideias e pensamentos para que este trabalho fosse concluído.

"Existem muitos modos de ir para a frente, mas apenas um modo de ficar parado."

FRANKLIN D. ROOSEVELT

SUMÁRIO

LISTA DE ILUSTRAÇÕES 9

LISTA DE TABELAS 10

LISTA DE ABREVIATURAS 11

1 INTRODUÇÃO 14
1.1 Descrição do Problema 15
1.2 Objetivo 17
1.3 Organização da Dissertação 18

2 CONCEITOS BÁSICOS 19
2.1 Android 19
2.2 Segurança da Informação 20
2.2.1 Aspectos de segurança 20
2.2.2 Vulnerabilidades 21
2.3 Técnicas para Detecção Antecipada e Tardia de Vulnerabilidades 21
2.4 Revisão de Código 23
2.4.1 Verificação Manual 23
2.4.2 Verificação Automatizada 24
2.5 Vulnerabilidades em aplicativos Móveis 25

3 UMA TÉCNICA PARA AVALIAÇÃO E DETECÇÃO DE CÓDIGO POTENCIALMENTE VULNERÁVEL EM APLICAÇÕES ANDROID 27
3.1 Classificação de Vulnerabilidades para Desenvolvimento Seguro em Android . 27
3.1.1 *M2 - Insecure Data Storage* (Armazenamento Inseguro de Dados) 29
3.1.2 *M4 - Unintended Data Leakage* (Vazamento Não-Intencional de Dados) 30
3.1.3 *M5 - Poor Authorization and Authentication* (Autenticação e Autorização Falha) 31
3.1.4 *M7 - Client Side Injection* (Injeção de Código) 32
3.1.5 *M8 - Security Decisions Via Untrusted Inputs* (Decisões de Segurança via Entradas Não-Confiáveis) 35
3.1.6 *M9 - Improper Session Handling* (Manipulação Imprópria de Sessão) 36

3.1.7 Classificação de Vulnerabilidades para Desenvolvimento Seguro em Android . 37
3.2 Uma abordagem antecipada, baseada em análise estática, para auxiliar a detecção de vulnerabilidades 39
3.3 appDroidAnalyzer 40
3.4 Análise inicial da appDroidAnalyzer 42

4 PROVA DE CONCEITO 46
4.1 Estudo de Caso 46
4.1.1 Entrada Inválida 47
4.1.2 Armazenamento Frágil 48
4.2 Seleção de projetos para análise 50
4.3 Apuração dos resultados 52

5 TRABALHOS RELACIONADOS 55
5.1 Abordagem de detecção antecipada 55
5.2 Abordagem de detecção tardia de vulnerabilidades 57
5.3 Análise dos trabalhos relacionados 60

6 CONSIDERAÇÕES FINAIS 63
6.1 Conclusão 63
6.2 Contribuições 63
6.2.1 Classificação de Vulnerabilidades para Desenvolvimento Seguro em Android . 64
6.2.2 appDroidAnalyzer 64
6.3 Trabalhos Futuros 64

7 REFERÊNCIAS BIBLIOGRÁFICAS 66

8 APÊNDICES 69
8.1 APÊNDICE 1: Relação de aplicativos identificados 70

LISTA DE ILUSTRAÇÕES

FIG.1.1 Dinâmica do ataque. 16

FIG.2.1 Camadas da Plataforma Android. 19

FIG.2.2 Detecção tardia de vulnerabilidade. Baseado em (SOUZA, 2015) 22

FIG.2.3 Detecção antecipada de vulnerabilidade. Baseado em (SOUZA, 2015) 23

FIG.3.1 Assinatura do método *query* 34

FIG.3.2 Estrutura do Lint 41

FIG.3.3 Uma parte da lista de verificações do Lint 42

FIG.3.4 Aplicativo Lista de Configurações 43

FIG.3.5 Aplicativo vulnerável 43

FIG.3.6 Aplicativo protegido 44

FIG.3.7 Escaneamento sem vulnerabilidades 44

FIG.3.8 Autenticação do aplicativo 44

FIG.3.9 Identificação de vulnerabilidade de armazenamento frágil 45

FIG.4.1 Análise do código do método *queryLastaApp* 47

FIG.4.2 Resultado da análise do *WAPPushManager* 48

FIG.4.3 Pontuação original do jogo *Stick Cricket* 49

FIG.4.4 Pontuação modificada no *Stick Cricket* 50

FIG.4.5 Trecho do resultado do *Stick Cricket* 50

FIG.4.6 Resultado apurado 53

LISTA DE TABELAS

TAB.1.1 Volume (em milhões) de Unidades Vendidas em 2015. 14

TAB.2.1 Exemplo checklist - Verificação Manual. Baseado em Meier et al. (2005) 23

TAB.3.1 Classificação de Vulnerabilidades - Baseada em Ferreira (2016) 38

TAB.4.1 Lista de endereços de projetos Android 51

TAB.4.2 Exemplos de páginas para *download* de aplicativos 52

TAB.5.1 Comparativo entre trabalhos. 61

TAB.5.2 Comparativo entre trabalhos. 62

LISTA DE ABREVIATURAS

ABREVIATURAS

ADT	-	Android Development Tools
API	-	Application Programming Interface
App	-	Application
AST	-	Abstract Syntax Tree
CVE	-	Common Vulnerabilities and Exposures
FOSS	-	Free and Open Source Software
GPS	-	Global Positioning System
HTTP	-	Hyper Text Transfer Protocol
IDE	-	Integrated Development Environment
OWASP	-	Open Web Application Security Project
PDU	-	Protocol Description Unit
SGBD	-	Sistema Gerenciador de Banco de Dados
SQL	-	Structured Query Language
SO	-	Sistema Operacional
SOAP	-	Simple Object Access Protocol
SMS	-	Short Message Service
SSL	-	Secure Sockets Layer
URI	-	Uniform Resource Identifier
URL	-	Uniform Resource Locator
Web	-	World Wide Web

RESUMO

Atualmente a plataforma Android domina o mercado de *smartphones*. Uma pesquisa realizada pela HP indicou que a maioria dos aplicativos para Android possuem vulnerabilidades. Isto ocorre porque os desenvolvedores, inconscientemente, introduzem códigos vulneráveis no aplicativo, e isto acarreta o vazamento de informações sensíveis e privadas do usuário quando uma vulnerabilidade é explorada. Vulnerabilidades inseridas em um código-fonte colocam as informações dos usuários em risco, uma vez que estas podem ocasionar o comprometimento de importantes aspectos de segurança, como a confidencialidade, a integridade e a disponibilidade.

Contudo, a procura por vulnerabilidades em aplicativos tem sido objeto de estudo de diversos trabalhos que utilizam uma abordagem para detectar estas vulnerabilidades tardiamente, baseada no *dex bytecode* do aplicativo, quando este está pronto e disponível para *download* em lojas, o que pode ter afetado milhares de dispositivos.

Este trabalho possui dois objetivos. Inicialmente será apresentada uma classificação das vulnerabilidades enumeradas em *Top Ten Mobile Risks*, relacionando-as com os aspectos de segurança da informação definidos pela ABNT NBR ISO/IEC 27.002:2013 e com os métodos Java que introduzem estas vulnerabilidades no sistema.

Fundamentado nesta classificação o segundo objetivo apresentado é um método, baseado na técnica de análise estática com casamento de padrões, para identificar, analisar e avaliar códigos potencialmente vulneráveis, de forma antecipada, durante o desenvolvimento de aplicativos para Android. Este método foi utilizado para a criação da ferramenta appDroidAnalyzer, que foi usada para avaliar esta metodologia. Esta ferramenta permitirá ao desenvolvedor aprimorar o código-fonte do seu programa e evitar que brechas vulneráveis sejam inseridas neste.

Esta técnica foi avaliada através de uma prova de conceito experimental em 859 aplicativos obtidos do catálogo de projetos *open source* F-Droid. Estes foram avaliados pela ferramenta appDroidAnalyzer que identificou 65 aplicações com potenciais vulnerabilidades em seu código-fonte.

ABSTRACT

Android platform dominates the smartphone marketshare. A survey performed by HP indicated that the most of Android applications have vulnerabilities. This occurs because the developers unknowingly introduce vulnerable code into the application, which leads to leakage of sensitive and private user information when a vulnerability is exploited. User's information are put at risk when vulnerabilities are inserted into a source code as it can lead to compromising important security aspects such as confidentiality, integrity, and availability.

However, the search for vulnerabilities in applications has been the subject of many studies that use the approach of late detection of vulnerabilities, based on the application's dex bytecode, when it is ready and available for download in stores, which may have affected thousands of devices.

This study has two objectives. Initially, a classification of the vulnerabilities listed in Top Ten Mobile Risks will be presented, relating them to the information security aspects defined by ABNT NBR ISO/IEC 27.002: 2013 and the Java methods that introduce these vulnerabilities into the system.

Based on this classification, the second objective presented is a method, based on the technique of static analysis with pattern matching, to identify, analyze and evaluate potentially vulnerable codes, in advance, during the development of Android applications. This method was used to create the appDroidAnalyzer tool, which was used to evaluate this methodology. This tool will allow the developer to enhance the source code of the program and prevent vulnerable loopholes from being inserted into it.

This technique was evaluated through an experimental concept proof in 859 applications obtained from the F-Droid open source project catalog. They were evaluated by the appDroidAnalyzer tool, which identified 65 applications with potential vulnerabilities in their source code.

1 INTRODUÇÃO

O avanço tecnológico dos últimos anos permitiu a criação e o barateamento dos componentes de *hardware*, tornando-os cada vez menores, porém com uma capacidade de processamento superior. Tal evolução, prevista por Gordon Earl Moore em 1965, deu origem à Lei de Moore, que segundo Schaller (1997), enuncia que o poder de processamento dos computadores dobraria a sua capacidade ao final de cada período de 18 meses. Atualmente, são encontrados no mercado de informática microprocessadores e memórias, em tamanho reduzido, com um poder de processamento similar à alguns computadores pessoais. Em paralelo a isto, a popularização e evolução da internet e os investimentos em redes de telefonia móvel permitiram que a comunicação ficasse cada vez mais rápida e confiável.

Aliado a esta evolução outros fatores, como por exemplo, o surgimento de telas sensíveis ao toque, contribuíram para que os fabricantes de telefones móveis inovassem, investindo na criação de aparelhos sofisticados com capacidade de armazenamento e processamento similar aos computadores encontrados em casas e escritórios. Estes dispositivos passaram a realizar outras tarefas além de, simplesmente, fazer chamadas e enviar mensagens *SMS (Short Message Service)*. Hoje já é possível utilizar um telefone móvel para acessar contas bancárias, realizar compras em grandes redes varejistas, registrar fotografias, jogar, dentre outras utilidades.

Com o surgimento destes novos aparelhos, diversos sistemas operacionais para suportar este ambiente foram criados, destacando-se Android, iOS e Windows Phone. Android é, atualmente, a plataforma móvel mais utilizada, onde dispositivos com este sistema operacional (SO) dominam o mercado de *smartphones*, no qual a sua representatividade chega a 81,5% (SCARSELLA et al., 2015), conforme tabela 1.1.

TAB. 1.1: Volume (em milhões) de Unidades Vendidas em 2015.

Sistema Operacional	Volume	Market Share
Android	1161,1	81,2%
iOS	226	15,8%
Windows Phone	31,3	2,2%
Outros	11,3	0,8%
Total	1429,7	100%

Neste cenário, um mercado para desenvolvimento de *software* aderente a esta plataforma surgiu, fazendo com que empresas e desenvolvedores passassem a vislumbrar novos caminhos para o seu negócio. Estes novos programas, conhecidos como aplicativos (app), são distribuídos em lojas *online*, como a loja oficial do Android (Google Play) que possui 2,43 milhões de apps disponíveis para *download* (APPBRAIN, 2016).

1.1 DESCRIÇÃO DO PROBLEMA

Segundo Rawlinson (2013), em uma pesquisa realizada pela HP que avaliou 2.107 aplicativos para Android, 90% destes apresentam vulnerabilidades. Já o relatório técnico publicado pela empresa Arxan, que considerou em sua pesquisa aplicativos para a plataforma Android, 97% dos gratuitos e 80% dos pagos possuem alguma vulnerabilidade (TECHNOLOGIES, 2014).

Crespo e Chóez (2012) definem vulnerabilidade como sendo toda falha de segurança em sistemas de informática que faz com que estes funcionem de maneira diferente do que foi programado, afetando desta forma a segurança e podendo levar a perda ou roubo de informações sensíveis. Isto ocorre, por exemplo, quando uma aplicação armazena no sistema informações confidenciais, que somente usuários autorizados podem acessá-las, sem utilizar um algoritmo de criptografia para codificar estes dados. Esta situação permite que qualquer pessoa que esteja de posse do aparelho acesse estas informações, configurando, desta forma, a perda ou roubo destas.

As vulnerabilidades podem ocorrer tanto em *hardware* quanto em *software*, no entanto, normalmente, é neste último onde ocorrem as falhas mais exploradas (MARTÍNEZ; ROJAS, 2015). Vulnerabilidades inseridas em um código-fonte podem colocar as informações dos usuários em risco, uma vez que estas brechas podem levar ao comprometimento de importantes aspectos de segurança, como a confidencialidade, a integridade e a disponibilidade. Um exemplo ilustrativo deste problema é o módulo *WAPPushManager* da plataforma Android, que foi identificado como vulnerável à Injeção de SQL. O *Common Vulnerabilities and Exposures* (CVE), que é um dicionário público de vulnerabilidades e riscos de segurança da informação e que possui como objetivo padronizar uma lista de nomes de vulnerabilidades para facilitar a procura e o compartilhamento de informações de segurança, catalogou esta vulnerabilidade na CVE-2014-8507[1] e a classificou como crítica e de fácil exploração, cujo impacto compromete a confidencialidade, integridade e disponibilidade do sistema.

[1]https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-8507

A figura 1.1 ilustra a dinâmica deste problema. Um desenvolvedor introduz, de forma não-intencional, um defeito no código-fonte que desenvolveu e, desta forma, gera uma vulnerabilidade no aplicativo (app) que foi construído. Após o aplicativo ser disponibilizado em uma loja para *download*, um atacante instala o app, identifica esta brecha e inicia o ataque para explorar esta vulnerabilidade. Obtendo sucesso nesta investida contra o app e conseguindo aproveitar-se da fragilidade, os aspectos de segurança de um determinado ativo são violados, tornando este vulnerável.

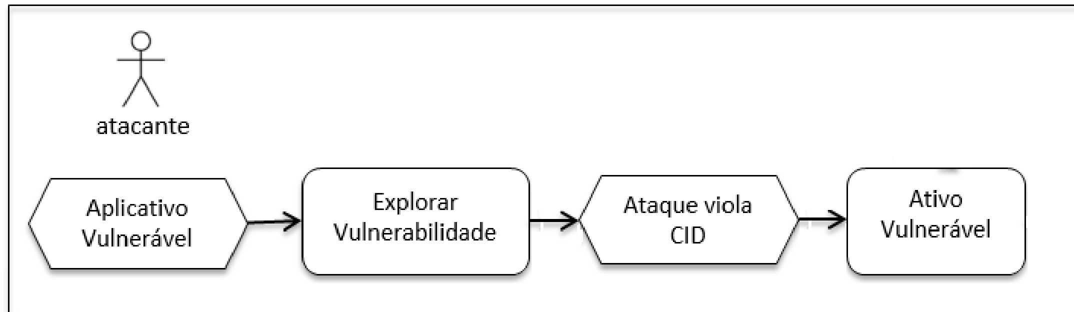

FIG. 1.1: Dinâmica do ataque.

A busca para encontrar vulnerabilidades em aplicativos tem sido objeto de estudo de diversos trabalhos. Em Wu et al. (2014), a partir da engenharia reversa do *dex bytecode*, que é o arquivo que contém as instruções de máquina de um aplicativo (abreviação de *Dalvik Executable*[2]), é realizada a procura por vulnerabilidades em componentes expostos (*Exposed Component Vulnerability*) e em *Application Programming Interfaces* (APIs) do Android. Ao encontrar um aplicativo vulnerável, os autores analisam e classificam esta vulnerabilidade.

No trabalho de Cheng et al. (2013), ainda através de uma abordagem que utiliza o *dex bytecode* do aplicativo, é proposto um método para detectar comportamentos perigosos em aplicações Android a partir da simulação da execução dos aplicativos. Este processo simulado tem como objetivo pesquisar comportamentos suspeitos do aplicativo em tempo de execução. Já no trabalho de Kim et al. (2012), é apresentada uma metodologia para procurar por vulnerabilidades que permitam que informações sensíveis do dispositivo móvel sejam expostas.

No entanto, a pesquisa por vulnerabilidades, como um passo anterior, durante a fase de desenvolvimento do aplicativo, tem sido pouco explorada. As abordagens baseadas na verificação a partir do *dex bytecode* se limitam a descobrir as vulnerabilidades após a aplicação estar concluída e publicada para *download*. Neste momento, milhares de

[2] https://source.android.com/devices/tech/dalvik/index.html

potenciais usuários já podem estar vulneráveis.

O trabalho de Sağlam (2014) procura identificar más práticas em codificação que possam ocasionar problemas de desempenho relacionados com o gerenciamento da memória. Ao identificar um aplicativo com problema, o autor relaciona-o com a sua baixa avaliação na loja Google Play. Apesar de tratar o problema durante a fase de desenvolvimento do aplicativo, esse trabalho não se preocupou com problemas relacionados a vulnerabilidades nos aplicativos.

1.2 OBJETIVO

Uma vez que os desenvolvedores de aplicativos podem introduzir defeitos, de forma não-intencional, em trechos do código que estão desenvolvendo e que estes defeitos podem gerar vulnerabilidades no aplicativo, este trabalho aborda os seguintes desafios de pesquisa:

- Identificar os tipos de código e métodos das classes em Java utilizados para o desenvolvimento de aplicativos para a plataforma Android, relacioná-los com as vulnerabilidades classificadas pela lista *Top Ten Mobile Risks* e correlacioná-las com os aspectos de segurança, definidos pela norma ABNT NBR ISO/IEC 27.002:2013, violados; e
- Avaliar o código-fonte gerado nas etapas de desenvolvimento do aplicativo à procura de potenciais vulnerabilidades.

Estes desafios tentam auxiliar na detecção de vulnerabilidades em um momento anterior ao que ocorre nas abordagens usuais de detecção de vulnerabilidades. No entanto, este trabalho não aborda as vulnerabilidades que são exploradas após o processo de desenvolvimento do aplicativo, dentre as quais, em uma arquitetura cliente-servidor, as recomendações e validações que devem ser aplicadas no servidor, a interceptação dos dados trafegados pela rede ou a criptoanálise do algoritmo de criptografia utilizada para trafegar ou armazenar informações do aplicativo. Também não serão abordadas as vulnerabilidades relacionadas com a engenharia reversa do app com a finalidade de se obter o código-fonte do mesmo. Para buscar possíveis soluções para estes itens, este trabalho tem os seguintes objetivos: gerar uma lista de métodos da linguagem Java relacionados às vulnerabilidades de aplicativos móveis, voltados para a plataforma Android, e identificar quais os aspetos de segurança estas vulnerabilidades violam; e desenvolver um método para auxiliar na detecção antecipada destes tipos de código. Desta forma, as contribuições esperadas para este trabalho são:

- Uma classificação de vulnerabilidades para desenvolvimento seguro em Android, que ajude o desenvolvedor a identificar os riscos de potenciais vulnerabilidades inseridas em seu código-fonte a partir dos métodos em Java utilizados. Esta classificação é baseada na lista proposta em Ferreira (2016); e

- Uma ferramenta de apoio computacional, baseada na técnica de análise estática com casamento de padrões, que implemente esta classificação, para testar, analisar e comunicar possíveis pontos vulneráveis em um código-fonte para aplicativos Android.

1.3 ORGANIZAÇÃO DA DISSERTAÇÃO

Além do primeiro capítulo de introdução, este trabalho está dividido em mais cinco capítulos, organizados da seguinte forma. O capítulo 2 apresenta a fundamentação teórica da dissertação e cita os conceitos relativos à plataforma Android, à segurança da informação, às técnicas de detecção, às formas de revisão de código e à lista de vulnerabilidades da OWASP. O capítulo 3 apresenta a metodologia utilizada por este trabalho. Já o capítulo 4 detalha os resultados obtidos através da realização da prova de conceito. O capitulo 5 apresenta os trabalhos relacionados com o tema abordado nesta pesquisa. Finalmente, o capítulo 6 conclui este trabalho apresentando as contribuições alcançadas e os trabalhos futuros.

2 CONCEITOS BÁSICOS

2.1 ANDROID

Esta plataforma possui uma arquitetura baseada em camadas, onde a camada inferior disponibiliza serviços para as camadas superiores.

O núcleo da plataforma é o sistema operacional Linux (Linux *Kernel*). A camada *Hardware Abstraction Layer* (HAL) é responsável pelos serviços de baixo nível e permite acesso aos componentes do sistema. Na camada *Native* C/C++ *Libraries* estão as bibliotecas nativas da plataforma, como as APIs para a camada de apresentação, para gerenciamento de banco de dados, dentre outras. Android *Runtime* permite que as aplicações sejam executadas nos dispositivos móveis. Já na camada Java API *Frameworks* estão as APIs que permitem que os aplicativos utilizem os recursos da plataforma, como telefonia, localização e notificações. Por fim, na camada *System Apps* estão as aplicações que são executadas sobre a plataforma. Podem ser aplicações nativas, como agenda de contatos, ou desenvolvida por terceiros, conforme figura 2.1 (DEVELOPERS, 2016).

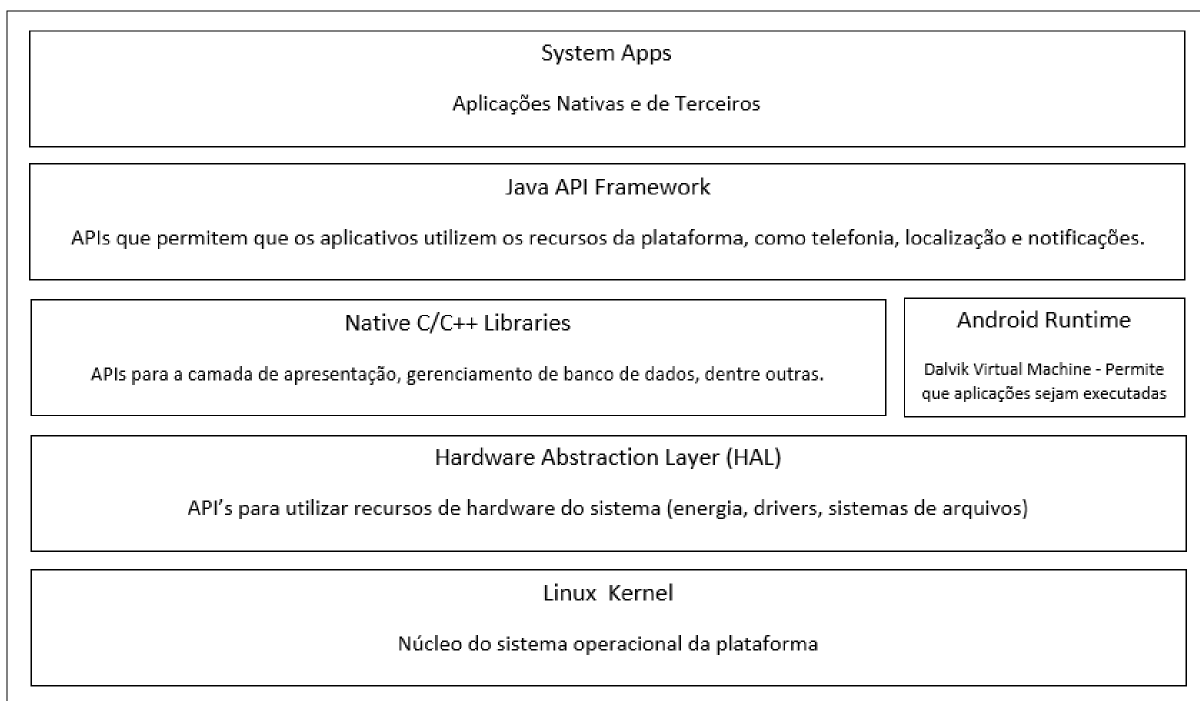

FIG. 2.1: Camadas da Plataforma Android.

Este trabalho aborda o problema apresentado no capítulo 1 em ralação aos problemas de segurança que ocorrem na camada de aplicação (*System Apps*), pois é nesta camada que ficam instalados os aplicativos desenvolvidos por terceiros.

2.2 SEGURANÇA DA INFORMAÇÃO

2.2.1 ASPECTOS DE SEGURANÇA

Segurança da Informação é a proteção das informações e ativos de uma organização contra diversas ameaças, com a finalidade de preservar a confidencialidade, a integridade e a disponibilidade (ABNT, 2013). Está diretamente relacionada com a proteção das informações de ativos com alto valor para uma organização.

Confidencialidade é a propriedade que limitará o acesso à informação somente aos usuários com permissão para visualizá-la. É a capacidade que um sistema possui de impedir que usuários não-autorizados acessem determinada informação.

Já o conceito de Integridade é a propriedade que garante que a informação será alterada somente por processos autorizados. É a capacidade que o sistema possui de impedir que a informação seja alterada por pessoas não autorizadas. Em último caso, detectar que houve uma alteração sem que exista a devida permissão para isto. Segundo ABNT (2013) é a propriedade que garante a exatidão e completeza de ativos.

Por fim, a Disponibilidade é a propriedade que assegura que uma informação estará sempre disponível para os usuários autorizados, enquanto estará indisponível para quem não possua autorização para acessá-la. Segundo Albuquerque e Ribeiro (2002), indica a quantidade de vezes que um sistema executou uma tarefa solicitada sem erros sobre o número de vezes em que a tarefa foi solicitada.

Outros conceitos importantes acerca de segurança da informação, segundo Albuquerque e Ribeiro (2002), são ativo, ameaça e ataque. Um ativo é algo de valor que é garantido pelo sistema, normalmente relacionado com algum aspecto de segurança (confidencialidade, integridade ou disponibilidade); uma ameaça está relacionada com um conjunto de três elementos: um atacante (ou agente de ameaça); uma vulnerabilidade; e um ativo com valor. Uma ameaça é um ataque em potencial; já um ataque ocorre quando o atacante consegue explorar uma vulnerabilidade e obtém o retorno desejado, atingindo o ativo com alto valor. Finalmente, atacante ou agente de ameaça é o responsável pela execução de um ataque.

2.2.2 VULNERABILIDADES

De acordo com Crespo e Chóez (2012), vulnerabilidades são brechas ou falhas em um sistema que permitem que usuários mal-intencionados violem os aspectos de segurança de uma aplicação. É toda falha de segurança em sistemas de informática que faz com que estes funcionem de maneira diferente do que foi programado, afetando a segurança e levando a perda ou roubo de informações sensíveis. É uma fragilidade de um ativo ou grupo de ativos que pode ser explorada por uma ou mais ameaças (ABNT, 2013).

Segundo Dantas (2011), as vulnerabilidades possuem origens naturais, organizacionais, físicas, de *hardware*, de *software*, nos meios de armazenamento, humanas e nas comunicações.

No contexto deste trabalho será abordada a vulnerabilidade de *software*, que são ocasionadas por falhas no desenvolvimento. Normalmente ocorrem por falta de conhecimento do desenvolvedor, tempo reduzido para entrega do produto ou por má-fé (quando uma falha é introduzida propositalmente no código). Vazamento de código em pequenas mídias também pode ocasionar invasões no sistema, uma vez que o atacante possui acesso ao código-fonte da aplicação.

2.3 TÉCNICAS PARA DETECÇÃO ANTECIPADA E TARDIA DE VULNERABILIDADES

A detecção tardia é uma técnica caracterizada por encontrar vulnerabilidades em aplicações após a criação e compilação de um programa. Segundo Souza (2015), a detecção tardia de uma vulnerabilidade de segurança implica em um aumento de custo de um projeto, em média, de 17%.

Esta abordagem segue um fluxo de trabalho onde, primeiro, o desenvolvedor implementa o código de uma aplicação, o que pode levar semanas. Quando o programa está finalizado, seja uma funcionalidade ou uma versão completa, ele utiliza alguma ferramenta para detectar vulnerabilidades no seu projeto. Finalmente, após receber o relatório de vulnerabilidades, ele corrige as vulnerabilidades encontradas e gera uma nova versão do programa, conforme figura 2.2.

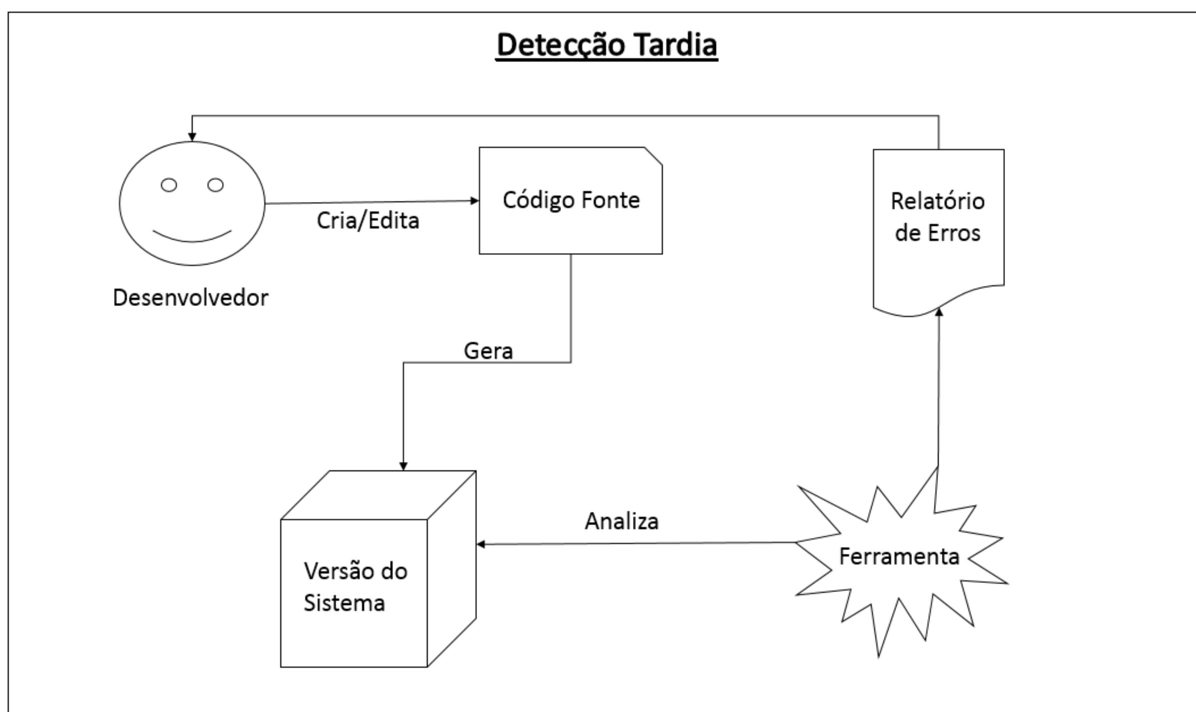

FIG. 2.2: Detecção tardia de vulnerabilidade. Baseado em (SOUZA, 2015)

A detecção tardia, em geral, não oferece uma prevenção para que as vulnerabilidades estejam presentes no *software*, é um modelo corretivo, onde a identificação e correção do problema ocorre após a utilização da aplicação.

Em contrapartida, a detecção antecipada é uma técnica utilizada para encontrar vulnerabilidades em aplicações durante o processo de codificação de um programa, enquanto o programador está criando ou editando seu código.

Neste aspecto, os desenvolvedores têm a possibilidade de remover potenciais vulnerabilidades do código fonte enquanto o contexto do problema a ser resolvido ainda está nas suas memórias, o que facilita o entendimento e correção do código (SOUZA, 2015). Deste modo, a detecção antecipada promove suporte constante ao desenvolvedor, uma vez que a todo momento o mesmo é alertado de possíveis problemas identificados em seu código.

Conforme figura 2.3, nesta abordagem o desenvolvedor é alertado de possíveis vulnerabilidades enquanto o programa é codificado.

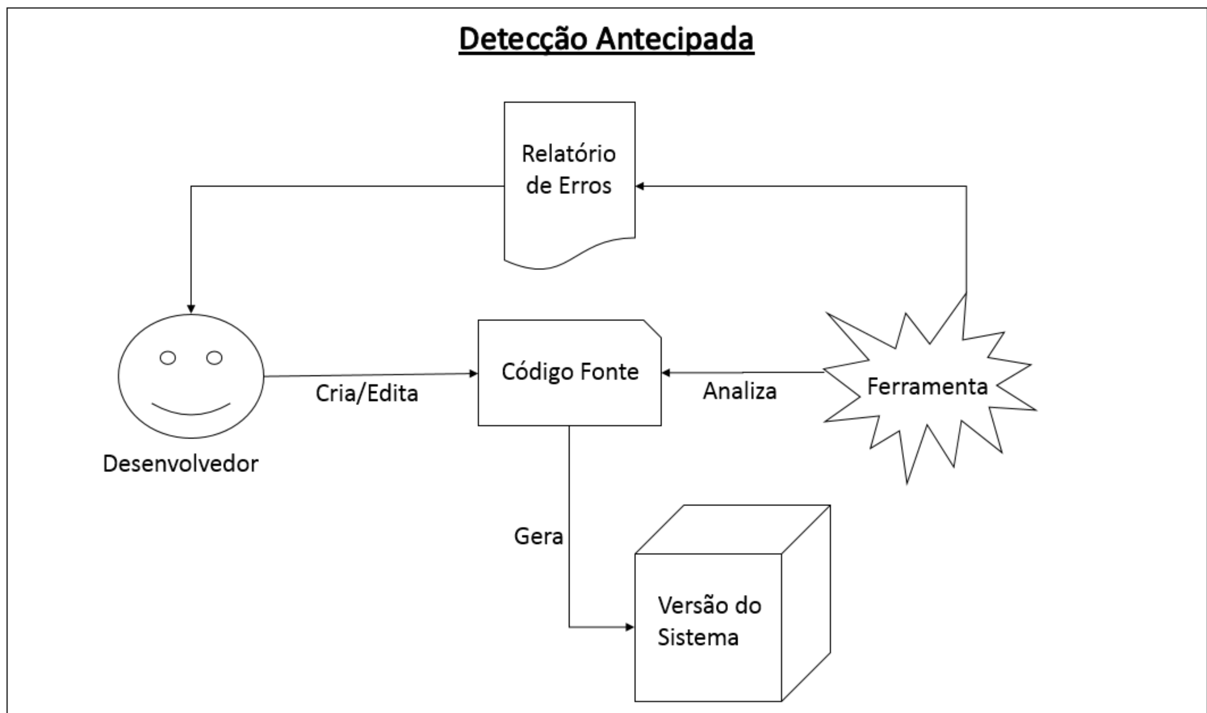

FIG. 2.3: Detecção antecipada de vulnerabilidade. Baseado em (SOUZA, 2015)

2.4 REVISÃO DE CÓDIGO

2.4.1 VERIFICAÇÃO MANUAL

A verificação manual de código é o processo pelo qual o desenvolvedor examina o código-fonte de um programa com a finalidade de identificar, analisar e corrigir um ponto vulnerável do *software*.

De acordo com Souza (2015), normalmente, é utilizado um *checklist* com o nome de uma vulnerabilidade e a explicação do que deve ser procurado no código-fonte que sinalize um possível problema de segurança. Um exemplo deste *checklist* pode ser visto na tabela 2.1.

TAB. 2.1: Exemplo checklist - Verificação Manual. Baseado em Meier et al. (2005)

Item	Procurar por
Acesso a Dados	Procure por armazenamento impróprio de strings de conexão e autenticação para o banco de dados
Autenticação	Procure por senhas fracas, credenciais em texto simples e outros problemas comuns de autenticação
Autorização	Procure pela separação inadequada de privilégios e outros problemas de autorização comum.

Este sistema de revisão apresenta vantagens e desvantagens. Segundo Souza (2015), a maior vantagem reside no fato de que a verificação manual é feita por especialista naquela linguagem de programação, que procura pelos erros apontados no *checklist*, contudo, este tem a capacidade intelectual para, durante a inspeção, encontrar outros problemas relacionados com o código.

Já a desvantagem deste método está no fato de que a revisão de um código que possua centenas de milhares de linhas de código torna-se muito difícil. Nesta situação o desenvolvedor fracassa ao não conseguir identificar todas as vulnerabilidades, permitindo que o produto fique vulnerável a ataques.

2.4.2 VERIFICAÇÃO AUTOMATIZADA

A verificação automatizada ocorre de duas formas: análise estática e análise dinâmica. A primeira forma, normalmente, envolve utilizar uma ferramenta para examinar a estrutura do código-fonte de uma aplicação, sem executá-lo, para procurar por erros ou vulnerabilidades durante o processo de codificação (LI et al., 2016). Quando um problema é encontrado, ele é informado para o desenvolvedor para que este o corrija.

A grande vantagem de se utilizar a análise estática de código é que esta pode revelar os erros e vulnerabilidades antes que estes aconteçam ou sejam explorados, até mesmo antes que uma versão do produto esteja concluída.

Segundo Souza (2015), a principal desvantagem em utilizar análise estática é que a maioria das ferramentas implementa esta técnica utilizando o casamento de padrões, o que não é suficiente para garantir que a vulnerabilidade realmente exista, uma vez que o contexto das variáveis e métodos não é considerado na análise.

Já a técnica de análise dinâmica para identificação de vulnerabilidades examina uma aplicação em tempo de execução (JOY; AJITH, 2016). Nesta técnica a aplicação é executada para que seja feita a procura por problemas, enviando entradas maliciosas para verificar se um resultado inesperado irá ocorrer.

A vantagem de se utilizar a análise dinâmica é que esta técnica identifica vulnerabilidades sem levar em consideração a forma de implementação do programa, a identificação da vulnerabilidade é baseada no retorno da requisição. Porém, como a validação é baseada na execução do programa, não há garantia de que todo o código seja inspecionado, uma vez que só o que passar pelos testes será verificado. Além desta desvantagem, é necessário que toda a aplicação esteja desenvolvida para que seja possível utilizar esta técnica.

2.5 VULNERABILIDADES EM APLICATIVOS MÓVEIS

Open Web Application Security Project (OWASP) é uma organização dedicada a capacitar as organizações a desenvolver, adquirir e manter aplicações confiáveis. Sua missão é fazer com que indivíduos e empresas obtenham conhecimento em relação aos riscos envolvidos no desenvolvimento de softwares seguros. OWASP *Mobile Security Project* é um projeto de segurança para aplicações móveis com foco em verificações de segurança. É destinado aos desenvolvedores e equipes de segurança e tem como finalidade prover os recursos para construir e manter aplicações móveis seguras, classificar os riscos e fornecer os controles necessários para minimizar os impactos ou a exploração de vulnerabilidades. Mantém a lista de vulnerabilidades *OWASP Top Ten Mobile Risks* (OWASP, 2014c), que apresenta aquelas que foram analisadas e classificadas por esta organização em 2014, da mais crítica para a menos crítica, considerando a facilidade de exploração, a detecção e o impacto no ativo explorado, conforme relação a seguir.

M1. *Weak Server Side Controls*: está relacionada com as medidas que devem ser empregadas para garantir que a segurança das informações seja mantida quando uma arquitetura cliente-servidor é utilizada. Estas medidas objetivam evitar que dados sensíveis sejam expostos a usuários não autorizados.

M2. *Insecure Data Storage*: é referente à exposição de informações confidenciais, como por exemplo *logs* e histórico de transações, que estão armazenadas no dispositivo móvel do usuário de forma insegura.

M3. *Insufficient Transport Layer Protection*: é pertinente às informações obtidas por um atacante, através da interceptação da comunicação, quando o tráfego de rede não oferece uma proteção adequada, como por exemplo, com o uso de *Secure Sockets Layer* (SSL).

M4. *Unintended Data Leakage*: vazamento não-intencional de dados é relativa às vulnerabilidades que ocorrem devido a falhas do Sistema Operacional ou do *hardware*. Em geral, informações são armazenadas em um local facilmente acessado por outros aplicativos sem que o desenvolvedor tenha prévio conhecimento destas fragilidades.

M5. *Poor Authorization and Authentication*: a exploração desta vulnerabilidade ocorre quando o atacante conhece como é realizado o processo de autenticação do aplicativo e, a partir deste momento, realiza uma solicitação de autenticação falsa ao sistema. Uma vez autenticado, ele realiza as operações disponíveis sem que a sua presença seja percebida.

M6. *Broken Cryptography*: está associada com a baixa complexidade utilizada para

realizar a criptoanálise do algoritmo de criptografia utilizado para cifrar as informações que são armazenadas ou que trafegam pelo aplicativo.

M7. *Client Side Injection*: esta vulnerabilidade se apresenta em aplicativos móveis sob as formas de Injeção de SQL, Inclusão em Arquivo Local, Cross-site scripting (XSS)[3]e Ataque ao código binário. Ocorre quando um código malformado é fornecido como entrada para um processo do sistema que o executa sem realizar uma prévia validação.

M8. *Security Decisions Via Untrusted Inputs*: esta vulnerabilidade ocorre através da troca de mensagens, com parâmetros não confiáveis, entre dois aplicativos e não há a validação das informações transmitidas entre eles. Esta comunicação ocorre por meio de um mecanismo chamado de Comunicação entre Processos (*Inter-Process Communication* - IPC).

M9. *Improper Session Handling*: esta vulnerabilidade permite que um atacante se passe por um usuário legítimo quando aquele consegue obter um *token* de sessão manipulado de forma imprópria.

M10. *Lack of Binary Protections*: está relacionado com um aplicativo que sofreu uma alteração maliciosa em seu código-fonte que foi obtido a partir da engenharia reversa do *dex bytecode*. Tem como objetivo fazer com que o código malicioso não seja percebido pelo usuário final, quando este instalar o app modificado pensando estar utilizando um app seguro.

[3]Injeção de *scripts* maliciosos em páginas web

3 UMA TÉCNICA PARA AVALIAÇÃO E DETECÇÃO DE CÓDIGO POTENCIALMENTE VULNERÁVEL EM APLICAÇÕES ANDROID

Este capítulo descreve a metodologia utilizada para atingir o objetivo geral deste trabalho, que é propor uma classificação de vulnerabilidades que ajude o desenvolvedor a evitar que potenciais vulnerabilidades sejam inseridas de forma não intencional no código-fonte, durante o processo de codificação do *software*, com a finalidade de preservar os aspectos de segurança das informações do aplicativo.

Esta classificação de vulnerabilidades tem como objetivo propor uma solução que relaciona os principais problemas apontados pela lista OWASP (2014c) com as recomendações de segurança sugeridas em ABNT (2013) violadas e com quais classes e métodos em programação para Android que as produzem. Além disto, a ferramenta appDroidAnalyzer, desenvolvida sob a ótica da análise estática com detecção de padrões, será utilizada para experimentar e validar esta classificação.

Nesta classificação serão consideradas as fragilidades que podem ser exploradas diretamente no dispositivo móvel. As vulnerabilidades que são exploradas em outros meios não foram analisadas como, por exemplo, as que ocorrem durante a transmissão de informações pela rede ou que se manifestam no lado servidor da aplicação.

3.1 CLASSIFICAÇÃO DE VULNERABILIDADES PARA DESENVOLVIMENTO SEGURO EM ANDROID

Existe um grande número de recomendações em programação que aliam a segurança de uma aplicação com boas práticas de programação, que de fato, se seguidas, eliminam muitos erros e falhas de segurança das aplicações (ALBUQUERQUE; RIBEIRO, 2002). Dentre estas recomendações, em relação à segurança, temos as normas Código de Prática para a Gestão da Segurança da Informação (ABNT NBR ISO/IEC 27.002:2013) e *Common Criteria for Information Technology Security Evaluation* (ISO/IEC 15408). Além destas normas, outras iniciativas contribuem para melhorar a segurança nas aplicações, como *Building Security In Maturity Model* (BSIMM-7), *Software Engineering Institute CERT Coding Standards* da universidade (*Carnegie Mellon*) e *OWASP Mobile Security Project*.

Para alinhar as recomendações com o escopo deste trabalho foram utilizadas as ori-

entações estabelecidas pela norma ABNT NBR ISO/IEC 27.002:2013 com a lista de vulnerabilidades da organização OWASP.

A norma ABNT NBR ISO/IEC 27.002:2013 (Código de Prática para Controles de Segurança da Informação) (ABNT, 2013) foi selecionada porque define os objetivos e diretrizes gerais para manter um sistema de gestão de segurança da informação. Contém recomendações de boas práticas para implementar este sistema definindo os pilares de segurança que serão utilizados: confidencialidade, integridade e disponibilidade.

A lista *Top Ten Mobile Risks* foi escolhida porque, fundamentada em sua metodologia (OWASP, 2014b), classifica e expõe os riscos encontrados com mais frequência dentre os dados coletados por diversas organizações que contribuem para este projeto (OWASP, 2014a). Seu objetivo é capacitar as empresas a desenvolver, adquirir e manter aplicações confiáveis. Além disto, esta seleção considerou que esta lista é referente a vulnerabilidades específicas para aplicações desenvolvidas para plataformas móveis.

Apoiada nesta lista foi realizada uma análise a partir da qual puderam ser identificadas as características de cada vulnerabilidade, alinhando aos riscos indicados pela OWASP que podem ser evitados, preventivamente, no decorrer do processo de construção do app e que estivessem diretamente relacionados com a sua ocorrência no dispositivo móvel (FERREIRA, 2016).

As vulnerabilidades M1, M3, M6 e M10, descritas na seção 2.5, foram excluídas do escopo pois estas ameaças são exploradas após o processo de desenvolvimento do aplicativo, quando o mesmo já está pronto e publicado em uma loja oficial. M1 não se aplica, pois, a prevenção está relacionada com as validações que devem ocorrer no lado servidor do sistema; M3 é referente a problemas de segurança na camada de rede, ou seja, o agente ataca e explora a comunicação entre o aplicativo e o destino; M6 está relacionado com a criptoanálise do algoritmo de criptografia utilizado para trafegar ou armazenar informações do aplicativo; e M10 está relacionada com fazer a engenharia reversa do app com o objetivo de alterá-lo para, travestido de um aplicativo original, obter algum ativo valioso.

Por outro lado, as vulnerabilidades *M2 - Insecure Data Storage, M4 - Unintended Data Leakage, M5 - Poor Authorization and Authentication, M7 - Client Side Injection, M8 - Security Decisions Via Untrusted Inputs* e *M9 - Improper Session Handling* foram abordados nesta pesquisa.

Após definir o objetivo do trabalho, foi realizado o alinhamento das vulnerabilidades com os Métodos da linguagem de programação Java que podem, potencialmente, permitir a exploração destas (FERREIRA, 2016). De forma semelhante, esta classificação

também forneceu quais os pilares de segurança definidos na norma ABNT NBR ISO/IEC 27.002:2013 seriam violados com a exposição de um ativo exposto.

3.1.1 *M2 - INSECURE DATA STORAGE* (ARMAZENAMENTO INSEGURO DE DADOS)

Esta vulnerabilidade está relacionada com o armazenamento de dados do aplicativo no dispositivo móvel e é caracterizada pela perda ou exposição destas informações. Pode ocorrer sob a forma da exposição de dados sensíveis, como histórico de transações, nomes de usuários, senhas, *tokens*, dados de localização, informações pessoais e dados do dispositivo.

Esta situação pode ser ilustrada quando um atacante consegue obter, de forma ilícita, as credenciais de acesso de um usuário que estavam armazenadas de forma insegura e assume a identidade deste para realizar atos ilegítimos. Isto posto, um ataque bem-sucedido ao sistema comprometerá a confidencialidade das informações. Como haverá acesso a informação, indevidamente, por um agente não autorizado, estas estarão comprometidas, e a integridade será perdida se estas forem alteradas pelo agente, uma vez que terão sido modificadas de forma ilegítima (FERREIRA, 2016).

A plataforma Android disponibiliza como recurso para este armazenamento de dados os arquivos de preferências, chamados de *SharedPreferences*. Este recurso permite a escrita e leitura, em arquivos específicos de uma aplicação ou Atividade, através de combinações de chave-valor, cuja chave identifica a informação armazenada e o valor corresponde ao dado que está gravado. Outro recurso disponível é a criação de arquivos localmente, que podem ser criados na memória interna do dispositivo ou armazenados em cartão de memória externa *Secure Digital* (SD), denominados, respectivamente, Armazenamento Interno e Armazenamento Externo.

Para criar ou acessar um arquivo de preferências através do recurso *SharedPreferences* o desenvolvedor pode utilizar dois métodos. O *getSharedPreferences* é utilizado quando existe a necessidade de criação de mais de um arquivo deste tipo pelo aplicativo, pois, esta forma permite que ele seja identificado por um nome, que será especificado no primeiro parâmetro do método. Já o método *getPreferences* é utilizado quando há a necessidade de criação do arquivo para uma Atividade específica. Neste caso, o arquivo é identificado pela própria Atividade, uma vez que não é fornecido o nome para o método. Em ambos os casos, o modo de acesso ao arquivo deve ser informado. O modo *MODE_ WORLD_ READABLE* permite que qualquer aplicativo que conheça o identificador do arquivo de preferências o

acesse com permissão de leitura, enquanto que *MODE_ WORLD_ WRITEABLE* permite, nestas condições, a escrita. *MODE_ PRIVATE* restringe o acesso somente ao aplicativo que criou as preferências, através do *user id* do app, que é um identificador único criado quando o mesmo é instalado.

Em relação a criação de arquivos, o método *openFileOutput* é utilizado para a criação nas memórias interna e externa. Semelhante ao recurso *SharedPreferences*, é necessário informar o nome que o identificará e o modo de acesso, que neste caso, além do modo privado, existe também o *MODE_ APPEND*, que adicionará as informações no final do arquivo caso o mesmo já exista. Para gerar o arquivo em um armazenamento interno, primeiramente o usuário deverá obter o caminho do sistema de arquivos, através do método *getFilesDir*. Se o armazenamento ocorrer na memória externa, o usuário deverá obter o caminho da mídia através do método *getExternalStorageDirectory*.

Para que um aplicativo não possua uma brecha para Armazenamento Inseguro de Dados, esta pesquisa concluiu que o desenvolvedor deverá ser alertado de que os recursos das classes *getSharedPreferences* e *getPreferences* deverão ser evitados e, caso seja necessário utilizá-los, deverá ser usado o modo de acesso privado (*MODE_ PRIVATE*) com o uso adicional de um algoritmo de criptografia. Esta criptografia poderá ser feita através do método *setStorageEncryption*, que solicita ao SO que o armazenamento seja criptografado, para o armazenamento na memória interna do dispositivo, ou a API *javax.crypto* para armazenamento em cartão de memória externa (SD) (FERREIRA, 2016). Apesar do modo privado trazer alguma segurança ao armazenamento das informações, não há a garantia de que estes dados não serão acessados. Um atacante que possua acesso ao dispositivo com acesso *root* poderá ler e escrever nestes arquivos. Isto reforça a importância de que os dados sejam mantidos criptografados (RAMOS; FEITOSA, 2015). Os modos *MODE_ WORLD_ READABLE* e *MODE_ WORLD_ WRITEABLE*, que estão obsoletos desde a API 17, não deverão ser utilizados, uma vez que o primeiro permite que qualquer aplicação acesse o arquivo para leitura e o segundo permite acesso de escrita.

3.1.2 *M4 - UNINTENDED DATA LEAKAGE* (VAZAMENTO NÃO-INTENCIONAL DE DADOS)

Vazamento Não-Intencional de Dados está relacionado com vulnerabilidades do sistema operacional, do *hardware* e dos *frameworks* utilizados. Ocorre quando um desenvolvedor, sem conhecimento prévio destas vulnerabilidades, processa ou armazena informações sensíveis em áreas de fácil acesso do dispositivo.

Em Android, um recurso disponível que possui a característica de ser facilmente acessado, é o armazenamento de informações em cache dos dados de requisições *Hyper Text Transfer Protocol* (HTTP). Assim como ocorre em Armazenamento Inseguro de Dados, o cache destas requisições de um aplicativo pode ser armazenado tanto na memória interna do dispositivo quanto em um cartão de memória externa.

A criação de cache nesta plataforma realiza-se a partir do método *HttpResponse-Cache.install*, que recebe como entrada o diretório para armazenamento dos dados e o tamanho máximo, em *bytes*, que este arquivo poderá ter. Ao optar por guardar estas informações internamente, a criação do cache deverá ser precedida pelo método getCacheDir, que é responsável por obter o diretório do sistema de arquivos onde o cache ficará armazenado. Em contrapartida, ao escolher salvar externamente, o método utilizado deverá ser getExternalCacheDir, que devolve o local daquela memória, ficando limitado o cache ao tamanho máximo disponível. É importante salientar que a memória externa pode ser acessada livremente, fato que permite que esta possa ser reescrita ou apagada por qualquer entidade, inclusive, não estar mais acessível e disponível, caso tenha sido removida do dispositivo.

De acordo com Ramos e Feitosa (2015) a maioria das bibliotecas que implementam cache em Android não se empenham em garantir a segurança das informações, principalmente com o vazamento destes dados mantidos. Nestas situações, onde o desenvolvedor deseja obter uma melhoria em performance e decide guardar em cache os dados de requisições HTTP, um atacante poderá violar a confidencialidade destas informações, seja através de um *malware*, de um aplicativo modificado maliciosamente ou do acesso físico ao aparelho, o que irá configurar o vazamento não-intencional dos dados armazenados.

Consequentemente, para prevenir *Unintended Data Leakage* esta pesquisa demonstra que o desenvolvedor deverá ser alertado quando utilizar o código *HttpResponseCache.install* para armazenar cache de requisições web. Caso ele entenda ser necessário manter em cache informações sensíveis ou confidenciais, segundo Ramos e Feitosa (2015), estas devem estar criptografadas, para que uma camada adicional de segurança seja acrescentada nestas informações.

3.1.3 *M5 - POOR AUTHORIZATION AND AUTHENTICATION* (AUTENTICAÇÃO E AUTORIZAÇÃO FALHA)

Poor Authorization and Authentication é caracterizada por um processo falho de autenticação e autorização que permite que um agente mal-intencionado assuma a condição de

um usuário legítimo e roube informações sensíveis, cause danos à reputação do usuário ou cometa fraudes em nome deste. A partir do momento em que aquele agente obtém controle sobre o dispositivo e conhece o processo de autenticação da aplicação, o sistema é atacado, através de uma falsa requisição de autenticação e, em seguida, são solicitados os serviços que deseja burlar.

Esta vulnerabilidade se manifesta na plataforma Android através do armazenamento inseguro dos dados de autenticação, quando estes são guardados com o objetivo de se manter a sessão de um usuário ativa ou através da utilização de um campo do tipo senha (*inputType="textPassword"*) em uma Atividade. Consequentemente, os métodos para manipulação de arquivos descritos na vulnerabilidade *Insecure Data Storage* são adotados para este armazenamento.

A partir do momento em que um atacante obtém controle sobre o dispositivo e conhece o processo de autenticação da aplicação, o ataque é realizado fazendo uma falsa requisição de autenticação e, em seguida, os serviços que deseja burlar são solicitados. Neste cenário, segundo Ferreira (2016), o aspecto da confidencialidade será violado, uma vez que o atacante agora está de posse das informações do sistema. De forma semelhante, a integridade das informações será perdida caso algum dado seja alterado, pois o atacante não deveria conseguir alterar dados se não estivesse em posse das credenciais de acesso de um usuário legítimo. Do mesmo modo, a disponibilidade do sistema poderá ser afetada caso o ataque tenha este objetivo, enviando requisições aquém da capacidade de processamento do sistema.

Logo, quando o desenvolvedor criar uma Atividade que possuir um campo do tipo *inputType="textPassword"*, ele deverá ser alertado para utilizar um método de validação que não permita senhas consideradas fracas, como por exemplo, com oito ou menos bytes. Ademais, são válidas as mesmas considerações realizadas para a vulnerabilidade *Insecure Data Storage* caso o desenvolvedor armazene informações de autenticação localmente, o que não seria uma boa-prática de programação.

3.1.4 *M7 - CLIENT SIDE INJECTION* (INJEÇÃO DE CÓDIGO)

A vulnerabilidade *Client Side Injection*, na forma de Injeção de SQL, é relativa a introdução de entrada de dados maliciosos e malformados em campos de uma Atividade para que sejam obtidas informações confidenciais ou fraudes sejam cometidas.

A Injeção de SQL é caracterizada pela introdução de códigos maliciosos que, ao serem processados pelo sistema, atuarão em conjunto com a linguagem utilizada para

manipulação de dados, formando novos comandos que serão executados pelo Sistema de Gerenciamento de Banco de Dados (SGBD). A finalidade desta operação é expor informações confidenciais ou manipular os dados do sistema. Neste contexto, considere que um atacante insira entradas maliciosas em uma Atividade de um aplicativo e que este não valide as entradas de dados recebidas antes de utilizá-las para processamento de uma consulta SQL. Quando o app processar a requisição feita pela atacante, ele poderá obter ou alterar informações que ele não deveria acessar naquele contexto.

Para manipular os dados através de consultas SQL que serão realizadas no SGBD SQLite, a plataforma Android disponibiliza uma série de métodos para manipulação de dados, através da classe *SQLiteDatabase*. Para inserir dados em uma tabela existem os métodos *insert, insertOrThrow* e *insertWithOnConflict*. Os métodos *insert* e *insertOrThrow* possuem três parâmetros. O primeiro, *table*, recebe o nome da tabela onde o registro será inserido. O segundo, *nullColumnHack*, que é opcional, especifica o que deverá ser inserido, explicitamente, o valor nulo em uma coluna da tabela caso o último parâmetro esteja vazio. Por último, o parâmetro *values* relaciona cada coluna com o valor que será inserido. A diferença entre estes dois métodos reside no fato que *insertOrThrow* retornará uma exceção, caso a inserção apresente erro, enquanto que *insert* retornará o valor -1. Já o método *insertWithOnConflict*, além dos três parâmetros descritos, possui um adicional. O parâmetro *conflictAlgorithm* indica o que o SGBD deverá fazer caso exista um conflito com dados que já estão cadastrados. Se o valor for a constante CONFLICT_IGNORE, o sistema continuará a execução dos próximos comandos, senão, será retornada uma exceção indicando que os dados já existem.

Em relação a atualização de dados, são fornecidos os métodos *update, updateWithOnConflict, replace* e *replaceOrThrow*. A diferença entre os métodos *update* e *updateWithOnConflict* reside no parâmetro conflictAlgorithm, que tem a mesma função descrita para *insertWithOnConflict*. Os demais parâmetros, comuns aos dois métodos, são *table*, que indica o nome da tabela, *values*, que representa o nome das colunas e os valores que serão atualizados, *whereClause*, que poderá ser composta pela sequência de texto com as restrições para a atualização ou pela concatenação do texto de restrição com os critérios da clausula *where*, representados pelo símbolo "?" e por último o parâmetro *whereArgs*, que são os valores que serão substitutos do símbolo "?" do item anterior. Os métodos *replace* e *replaceOrThrow* possuem a particularidade de, ao tentar atualizar uma informação, caso ela não exista, os dados serão criados. Os parâmetros destes métodos são *table, nullColumnHack* e *initialValues*. Os dois primeiros cumprem a mesma função descrita para os métodos de inserção e *initialValues* é similar a *values*.

Para excluir uma informação do SGBD, o método *delete* é utilizado. Os parâmetros recebidos são *table*, *whereClause*, *where* e *whereArgs*, todos com as mesmas utilidades descritas para a atualização de dados.

Relativamente a seleção e consulta de informação, a plataforma oferece os métodos *rawQuery* e *query*. *rawQuery* é composta por dois parâmetros de entrada, *sql* que é a consulta propriamente dita e *selectionArgs* que são os dados que substituirão os símbolos "?" da consulta, caso estes existam em *sql*. Já *query* recebe o nome da tabela onde a consulta será realizada (*table*), as colunas desta tabela que serão retornadas (*columns*), a restrição aplicada (*selection*), os valores utilizados para restringir o retorno, através do símbolo "?" (*selectionArgs*), o agrupamento de dados(*groupBy*, a restrição do agrupamento (*having*) e a ordem de apresentação dos dados (*orderBy*).

A vulnerabilidade *Client Side Injection* se manifesta em um aplicativo quando os métodos para manipulação de dados em um SGBD SQLite são usados sem a utilização de parâmetros, com a concatenação de *strings* na cláusula *where*. A figura 3.1 apresenta a assinatura do método *query* com um exemplo que utiliza a concatenação de *strings*.

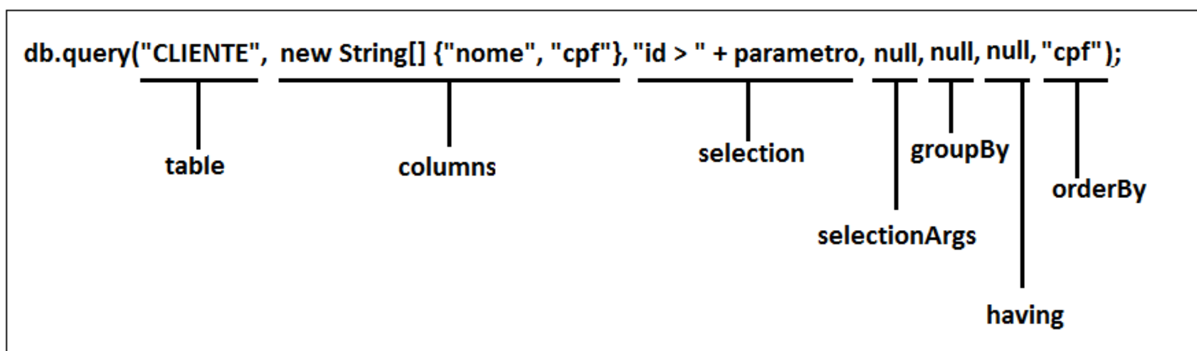

FIG. 3.1: Assinatura do método *query*

Para prevenir *Client Side Injection*, especificamente *SQL Injection* em banco de dados SQLite, o presente trabalho apontou que o desenvolvedor deverá ser alertado para sempre validar os parâmetros de entrada. Para os métodos de inserção (*insert, insertOrThrow e insertWithOnConflict*) isto é necessário para que não sejam armazenadas informações com código malicioso no SGBD. No que se refere aos métodos para consulta, quando o método *rawQuery* for identificado, o desenvolvedor deverá receber um alerta para não o utilizar, pois este não permite realizar consultas sem a concatenação de *strings*. Caso ele use o método *query* e seja identificado que os argumentos não estão sendo utilizados através do parâmetro *selectiontArgs*, que é responsável por não permitir que códigos maliciosos sejam inseridos, o desenvolvedor deverá ser alertado para o risco de que poderá sofrer um ataque deste tipo. No que diz respeito a *delete*, *update* e *updateWithOnCon-*

flict são válidas as mesmas recomendações para os métodos de consulta. Em relação a *replace* e *replaceOrThrow* as propostas para a inserção deverão ser seguidas. Estas orientações preservam a confidencialidade, a integridade e a disponibilidade das informações do sistema.

Em outras palavras, segundo Fang et al. (2017), dados externos e de fontes não confiáveis que não sofram uma validação estarão suscetíveis a sofrer injeção de código e assim expor informações sensíveis.

3.1.5 *M8 - SECURITY DECISIONS VIA UNTRUSTED INPUTS* (DECISÕES DE SEGURANÇA VIA ENTRADAS NÃO-CONFIÁVEIS)

Esta vulnerabilidade, em Android, está relacionada com a comunicação que ocorre entre dois aplicativos com a finalidade de que um utilize processos públicos do outro. Um processo público é uma funcionalidade disponibilizada por um aplicativo para que outros possam utilizá-la, como por exemplo, um app que possua um processo para visualizar um arquivo do tipo PDF pode tornar este público para que outros apps não precisem reescrever esta funcionalidade. Esta comunicação ocorre através de chamadas entre processos (*Inter Process Communication* - IPC), que são mecanismos que permitem aos processos trocarem informações entre si. A plataforma permite que esta comunicação seja realizada através da classe *Intent* (Intenção), que é uma descrição abstrata de uma operação que deverá ser executada pelo SO, como por exemplo, abrir uma Atividade.

Uma vez que existe este processo de comunicação, um aplicativo malicioso que não possua permissão explícita para leitura ou escrita no arquivo de contatos do aparelho, poderia através de uma chamada IPC para um app que possuísse tal permissão, solicitar a lista de contatos ou modificar um contato existente. Caso esta aplicação que recebeu a solicitação não valide se a outra possui permissão para tal, a confidencialidade ou integridade do sistema será violada (FERREIRA, 2016).

Security Decisions Via Untrusted Inputs ocorre quando uma aplicação recebe solicitações através de chamadas IPC e utiliza estas informações sem realizar uma validação antecipada. O atacante, que neste caso pode ser um malware ou um aplicativo malicioso, poderá burlar o mecanismo de segurança da plataforma Android e obter privilégios a que não tem direito, como acessar um recurso do sistema que não esteja explicitamente configurado no seu arquivo de configurações (AndroidManifest.xml).

Para que isto aconteça o desenvolvedor deverá ter utilizado o método *checkCallingOrSelfPermission*, que é usado para verificar se o aplicativo que está recebendo a chamada

IPC ou o app que solicitou o processo possui a permissão para acessar àquele recurso. Recebe como parâmetro a permissão que deve ser verificada. Por exemplo, para verificar se um dos aplicativos possui permissão para acessar a internet, o método deverá validar *context.checkCallingOrSelfPermission(android.permission.INTERNET)*.

Como resultado desta pesquisa é recomendado que o desenvolvedor seja alertado ao utilizar este método que ele permite que um aplicativo malicioso, sem permissão explícita para acessar um determinado recurso, pode se beneficiar desta brecha. Para prevenir esta vulnerabilidade o método a ser usado deve ser *checkCallingPermission*. Este método, que recebe o nome da permissão como parâmetro, verifica e valida se somente o processo de origem da chamada IPC possui a permissão desejada, o que impossibilita o acesso ao recurso caso aquela não exista.

3.1.6 *M9 - IMPROPER SESSION HANDLING* (MANIPULAÇÃO IMPRÓPRIA DE SESSÃO)

M9 - Improper Session Handling (Manipulação Imprópria de Sessão) é uma vulnerabilidade que ocorre quando os dados da sessão do usuário não estão protegidos corretamente, permitindo que um atacante se passe por um usuário legítimo do sistema e execute transações em nome deste. Para manter o estado da aplicação quando protocolos *stateless* são utilizados (HTTP ou SOAP) e facilitar as transações entre o aplicativo e o servidor, *session tokens* são utilizados. Inicialmente, o aplicativo autentica o usuário no servidor (*backend*) e em resposta à esta autenticação, o servidor emite um *session cookie* para que o aplicativo utilize este *cookie* nas suas próximas transações. Isto permitirá que o servidor, quando necessário, imponha autenticação e autorização para as próximas requisições de serviços emitidas pelo aplicativo.

Um agente de ameaça que obtenha as credenciais de um usuário (de forma ilegítima, através de um *malware* ou acesso físico ao dispositivo), que estejam armazenadas nestes *cookies* de sessão, poderá executar ações em nome deste sem levantar suspeitas. Isto caracteriza a violação da confidencialidade das informações do sistema.

Esta pesquisa identificou que, nesta plataforma, a manipulação imprópria de sessão é realizada através da remoção incorreta de *cookies* da aplicação. Android disponibiliza para este fim os métodos *removeAllCookies* da classe *CookieManager* e os métodos *remove* e *removeAll* da classe *CookieStore*. O método *removeAllCookies* é utilizado para remover todos os *cookies* da aplicação enquanto que o método *removeAll* remove um arquivo de *cookie* específico. Já *remove*, que recebe como parâmetro uma URI[4] e o nome do arquivo

de *cookie*, é utilizada para remover uma URI de um arquivo.

Como resultado do trabalho, para evitar que esta vulnerabilidade aconteça, o desenvolvedor precisa ser alertado que ao usar estes métodos, que ele deverá invalidar estes *cookies* no lado servidor da aplicação. Isto é necessário para evitar que estes *cookies* continuem válidos, abrindo uma brecha para que estas informações sejam utilizadas por terceiros.

3.1.7 CLASSIFICAÇÃO DE VULNERABILIDADES PARA DESENVOLVIMENTO SEGURO EM ANDROID

A partir da identificação das vulnerabilidades, dos aspectos de segurança violados e da identificação das classes e métodos que podem permitir que estas vulnerabilidades sejam exploradas, estas foram agrupadas e classificadas quanto a exposição das informações e às formas de validação empregadas para preveni-las, conforme a tabela 3.1.

A classificação "Armazenamento Frágil" inclui todas as vulnerabilidades que possuem como característica a precariedade em relação ao armazenamento das informações em áreas específicas do dispositivo. Aqui, estas possuem em comum o fato de que, uma vez que a fragilidade tenha sido explorada, as informações que foram armazenadas em texto claro estão expostas, como por exemplo, dados armazenados em cache sem criptografia que sofrem um ataque. Neste exemplo as informações já estarão em posse do atacante assim que o ataque for consumado.

Por outro lado, "Entrada Inválida" consiste na lista de vulnerabilidades que estão relacionadas a uma fragilidade relativa aos dados inseridos como entrada para processos internos do sistema. Nesta situação, o local ou a forma como os dados estão armazenados não é o fator que os torna vulneráveis, mas o que caracteriza este agrupamento e que irá expor uma informação é o vínculo entre um dado de entrada para o sistema com a sua posterior utilização sem que exista a sua validação.

[4] Cadeia de caracteres utilizada para identificar um recurso na internet.

TAB. 3.1: Classificação de Vulnerabilidades - Baseada em Ferreira (2016)

Classificação	OWASP	Tipos de Código		Aspecto Violado	Tipo	Agente de Ameaça
Armazenamento Frágil	M2-Insecure Data Storage	SharedPreferences	getSharedPreferences / getPreferences	Confidencialidade / Integridade	Alerta	Acesso físico ao dispositivo / malware / aplicativo malicioso
		Armazenamento Interno	getFilesDir + openFileOutput			
		Armazenamento Externo	getExternalStorageDirectory + openFileOutput			
Armazenamento Frágil	M4- Unintended Data Leakage	Cache em memória interna de requisição HTTP	getCacheDir + HttpResponseCache.install	Confidencialidade	Alerta	Acesso físico ao dispositivo / malware / aplicativo malicioso
		Cache em memória externa de requisição HTTP	getExternalCacheDir + HttpResponseCache.install			
Armazenamento Frágil	M9- Improper Session Handling	Remoção de cookies com a classe CookieManager	removeAllCookies	Confidencialidade	Alerta	Acesso físico ao dispositivo / malware
		Remoção de cookies com a classe CookieStore	remove / removeAll			
Armazenamento Frágil	M5- Poor Authorization and Authentication	SharedPreferences	getSharedPreferences / getPreferences	Confidencialidade / Integridade / Disponibilidade	Alerta	Malware / botnets
		Armazenamento Interno	getFilesDir + openFileOutput			
		Armazenamento Externo	getExternalStorageDirectory + openFileOutput			
Entrada Inváida		Campo de senha	inputType="textPassword"			
Entrada Inváida	M7- Client Side Injection	Execução de SQL com a classe SQLiteDatabase	Insert / insertOrThrow / insertWithOnConflict / delete / update / updateWithOnConflict / query / rawQuery / replace / replaceOrThrow	Confidencialidade / Integridade / Disponibilidade	Erro	Próprio usuário / aplicativo malicioso
Entrada Inváida	M8- Security Decisions Via Untrusted Inputs	Verificar permissões de aplicativos via chamada IPC	checkCallingOrSelfPermission	Confidencialidade / Integridade	Alerta	Próprio usuário / aplicativo malicioso / malware

3.2 UMA ABORDAGEM ANTECIPADA, BASEADA EM ANÁLISE ESTÁTICA, PARA AUXILIAR A DETECÇÃO DE VULNERABILIDADES

Uma vez que as vulnerabilidades foram classificadas e o escopo deste estudo foi definido, é necessário definir a abordagem que será utilizada para analisar, identificar e avaliar os códigos potencialmente vulneráveis do aplicativo. Como o objetivo deste estudo é ajudar o desenvolvedor a descobrir as possíveis fragilidades do seu aplicativo, durante o desenvolvimento, esta abordagem foca nos problemas de segurança identificados neste trabalho e que sejam identificados no nível do código-fonte da aplicação. Para isto, as técnicas de detecção antecipada de vulnerabilidades e de análise estática foram selecionadas.

A técnica de detecção antecipada de vulnerabilidades do código-fonte foi escolhida porque possui algumas vantagens em relação à detecção tardia. Dentre estas, podemos destacar o fato de que, ao abordar a identificação da fragilidade antecipadamente, não é necessário empregar nenhum método adicional para se obter o código-fonte a partir do *bytecode* do aplicativo, como por exemplo, a engenharia reversa do código. Ademais, outras técnicas podem ser empregadas para dificultar a obtenção e análise do código-fonte, como a ofuscação do código. Outra vantagem reside no fato de que a detecção antecipada da vulnerabilidade conduz ao desenvolvimento do *software* seguro, haja vista que estas são identificadas, analisadas e corrigidas antes do produto ser finalizado e publicado para uso. Já a detecção tardia proporciona uma análise retrospectiva de vulnerabilidades, quando estas já estão presentes no aplicativo e podem ter afetado os usuários do sistema. Por fim, o fato de o desenvolver receber notificações de vulnerabilidades durante o processo de codificação facilita a correção destas, devido ao fato de que o mesmo está com os requisitos e a lógica do processo guardados em sua memória recente, fato que não ocorre posteriormente.

De forma semelhante, a opção pela técnica de análise estática de código deve-se ao fato de que esta é, normalmente, empregada para analisar o código-fonte de um sistema sem a necessidade de executá-lo, enquanto que na análise dinâmica é necessário executar a aplicação e percorrer as funcionalidades que serão verificadas. Na maioria das vezes, as soluções que optam pela técnica de análise estática envolvem o uso de uma ferramenta automatizada que recebe como entrada o código de um programa e o examina, verificando a sua estrutura e chamadas de funções, dentre outras verificações. Outra vantagem de se utilizar esta técnica é que ela pode revelar erros ou vulnerabilidades antes que estes se manifestem ou sejam explorados. Já com a técnica de análise dinâmica os problemas são descobertos, algumas vezes, muito tempo depois do *software* ter sido liberado para o seu

público alvo. Finalmente, a análise dinâmica possui a desvantagem de somente validar os pedaços do código do sistema que forem executados durante a validação, à medida que na naquela, todo o código-fonte é analisado.

3.3 APPDROIDANALYZER

Com base nos estudos realizados, desenvolvi uma ferramenta que automatiza as verificações de vulnerabilidades descritas nesta classificação. Para desenvolver a ferramenta appDroidAnalyzer foi utilizado um computador Dell XPS13, com o sistema operacional Windows 7 Service Pack 1, processador Intel Core i5-2467M CPU 1.60 GHz e 4 Gb de memória RAM. Neste, foi instalada a IDE para desenvolvimento Java Eclipse, versão Mars.2 Release (4.5.2). Além da Eclipse, também foi necessário instalar o *plug-in* do Android Development Tools (ADT), versão 23.0.7.

Este é responsável por fornecer um ambiente integrado com as APIs da plataforma Android para a construção de aplicações. A ferramenta appDroidAnalyzer foi codificada a partir da criação de uma extensão da ferramenta Lint, portanto, ela é compatível tanto com o ambiente de desenvolvimento Android Studio (IntelliJ IDEA) quanto com o Eclipse (LINT PROJECT).

Lint é baseada na técnica de análise estática de código com casamento de padrões e utiliza a árvore sintática da própria linguagem de programação, nesse caso Java, possibilitando identificar as partes vulneráveis no código fonte do aplicativo, e é distribuída juntamente com o ADT, a partir da versão 16, fornecido pela Google. Conforme figura 3.2, Lint percorre todos os arquivos que compõe o projeto do aplicativo (arquivos Java, XML, ícones e configurações) e identifica as oportunidades de otimização do código no que se refere especificamente à exatidão, segurança, desempenho, usabilidade, acessibilidade e internacionalização. A verificação pode ser executada via linha de comando, no Eclipse ou IntelliJ (DEVELOPERS).

Posto que appDroidAnalyzer é uma extensão do Lint, esta ferramenta foi criada pelo autor através da implementação de classes Java que foram registradas na biblioteca do Lint, para que esta passasse a utilizar estas verificações em sua rotina de análise do código-fonte.

As vulnerabilidades identificadas em Ferreira et al. (2016) e classificadas neste trabalho foram utilizadas para balizar o limite de atuação desta ferramenta. Para cada tipo de código identificado neste trabalho foi implementada uma classe para detectar o problema. Estas classes, que implementam Detector.JavaScanner do Lint, percorrem o código fonte

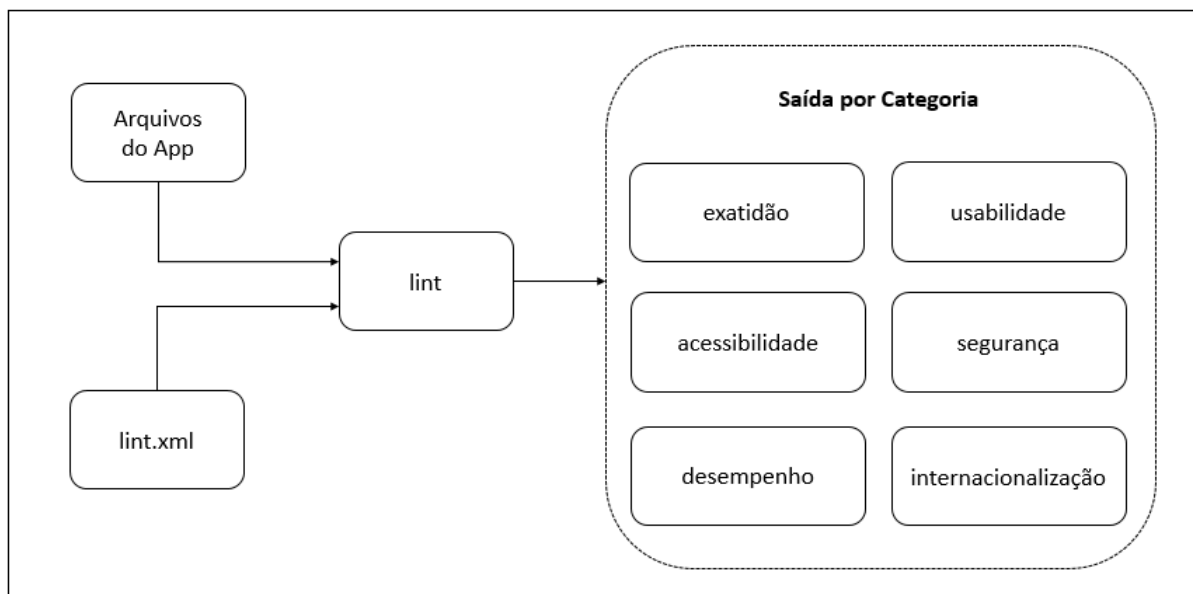

FIG. 3.2: Estrutura do Lint

do aplicativo a procura de um padrão de código que pode introduzir a vulnerabilidade.

Ao encontrar um possível problema, o mesmo é inserido no relatório de *logs* do Lint, e um alerta é emitido para o desenvolvedor com uma explicação do problema, o que pode ser feito para que o mesmo seja evitado, a sua prioridade e sua categoria. Caso nenhum problema seja encontrado o detector termina a verificação.

Para que o Lint reconheça estas verificações e passe a utiliza-las, é necessário registrar as novas classes que detectam as vulnerabilidades em sua biblioteca de verificação. Este registro é feito através do desenvolvimento de outras classes Java, uma para cada classe detectora, que estende a classe IssueRegistry do Lint. Um exemplo deste registro pode ser visto na figura 3.3, que lista um trecho de algumas verificações registradas na base de pesquisa do Lint.

A ferramenta appDroidAnalyzer oferece aos desenvolvedores a oportunidade de se obter um rápido *feedback* em relação ao código que ele está escrevendo, uma vez que ela identifica, em tempo de codificação, os problemas de segurança utilizados nesta classificação. Além disto, appDroidAnalyzer, possui um grande potencial para oferecer ao desenvolvedor a oportunidade de aprender a não cometer as mesmas falhas em termos de segurança, uma vez que ele estará sendo constantemente notificado dos problemas encontrados, assim que os mesmos forem identificados.

```
Available issues:
...
Security
========
...

GetSharedPreferencesDetector
----------------------------------------------------------------------------------------------------------------
Summary: SharedPreferences!               Certifique-se que usou criptografia para salvar informações ao utilizar o
                                          método getSharedPreferences. Este tipo de código permite que um usuário
Priority: 5 / 10                          mal-intencionado obtenha informações sensíveis da aplicação sem que o
Severity: Warning                         mesmo possua autorização para isto.
Category: Security                        More information:
                                          https://www.owasp.org/index.php/Mobile_Top_10_2014-M2

SQLInjectionQueryDetector
----------------------------------------------------------------------------------------------------------------
Summary: Injeção de SQL detectada!        Não utilize queries dinâmicas sem a utilização de parâmtros no SQLite.  Este
                                          tipo de código permite que um usuário mal-intencionado obtenha informações
Priority: 5 / 10                          sensíveis da aplicação sem que o mesmo possua autorização para isto.
Severity: Warning                         More information:
Category: Security                        https://www.owasp.org/index.php/Mobile_Top_10_2014-M7
...
```

FIG. 3.3: Uma parte da lista de verificações do Lint

3.4 ANÁLISE INICIAL DA APPDROIDANALYZER

Para avaliar a conformidade da ferramenta appDroidAnalyzer com esta metodologia, foram desenvolvidos, experimentalmente, dois aplicativos para testes iniciais desta ferramenta. O primeiro aplicativo, chamado de "Lista de Configurações", que valida a classificação "Entrada Inválida", simula uma aplicação com um cadastro de configurações que exibe os dados públicos para o usuário, omite os dados privados do sistema e utiliza o banco de dados SQLite para armazenar estas informações. O segundo aplicativo, denominado "Login", reproduz um sistema de autenticação que armazena as informações de acesso localmente, em um arquivo de preferências. Mantém os dados de usuário e senha armazenados para que o usuário seja autenticado sem necessidade de digitar novamente estas informações ao entrar no sistema. Este foi criado para validar a classificação "Armazenamento Frágil".

Para realizar esta avaliação, dois ambientes distintos foram utilizados. Para executar a ferramenta appDroidAnalyzer, foi empregado o mesmo computador utilizado para desenvolvê-la. O outro ambiente, utilizado para testar os aplicativos desenvolvidos, foram utilizados dois aparelhos com o sistema operacional Android: um MotoG 3ª Geração com Android 6.0, 1 Gb de memória RAM e processador Qualcomm Snapdragon 410 MSM8916 Cortex-A53; e um Samsung Galaxy S2 GT-i9100 com Android 4.0.4, 1 Gb de memória RAM e processador Samsung Exynos 4210 / ARM Cortex-A9 1200Mhz 32bits Dual-Core.

O aplicativo "Lista de Configurações" foi desenvolvido, propositalmente, suscetível ao ataque de Injeção de SQL, classificado como Entrada Inválida por este trabalho. Ao executá-lo, o sistema exibe uma Atividade com um campo para o usuário informar os

critérios de filtro que ele deseja utilizar para realizar a pesquisa de configurações. Quando nenhum parâmetro é informado ou uma sequência de caracteres válidos é utilizada, a lista de configurações públicas é exibida. Porém, como este aplicativo possui uma vulnerabilidade, se for informado um filtro com dados, intencionalmente, malformados, as configurações privadas (*HASH, mainWebService, password, senhaHash* e *user*) também serão exibidas, caracterizando desta forma a perda da confidencialidade de informações sensíveis do sistema, conforme a figura 3.4.

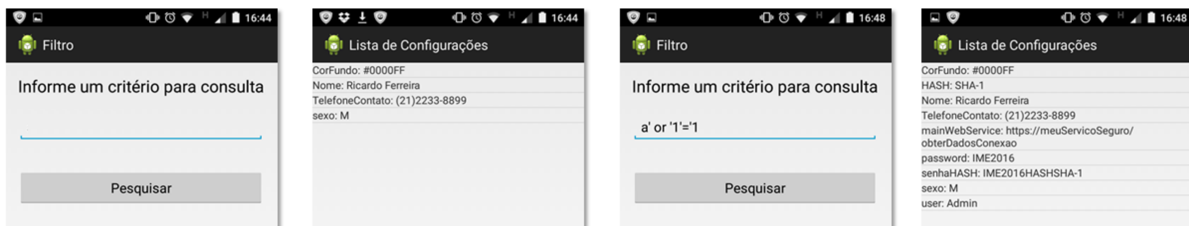

FIG. 3.4: Aplicativo Lista de Configurações

Após a confirmação de que o mesmo estava vulnerável, o código fonte deste aplicativo foi submetido à análise da appDroidAnalyzer. Esta ferramenta identificou que aquele aplicativo possui um possível ponto de exposição à vulnerabilidade de injeção de SQL, conforme a figura 3.5.

```
Scanning IME_SQLite: ..........
         - appcompat_v7:
src\br\eb\ime\sqlitevalidador\BD.java:49: Warning: Evite consultas dinamicas no
SQLite sem passar parametros para a query.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao,
obtenha dados privados, violando os aspectos de confidencialidade, integridade
 ou disponibilidade. [SQLiteInjectionDetector]
    cursor = bd.query("configuracao", colunas, "privado = 0 AND nome ='" + filtr
o + "'", null, null, null, "nome ASC");
               ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0 errors, 1 warnings
```

FIG. 3.5: Aplicativo vulnerável

Em seguida o aplicativo foi corrigido e submetido à uma nova tentativa de ataque, onde ele mostrou-se resistente ao problema. Ao inserir um parâmetro malicioso para a Atividade, o sistema executou a consulta, porém agora o sistema mostrou-se resiliente à tentativa de ataque, conforme figura 3.6. Novamente o código fonte deste aplicativo foi submetido à análise da appDroidAnalyzer e a vulnerabilidade não foi encontrada. Isto pode ser visto na figura 3.7, demonstrando que incorporar este processo de validação durante a fase de desenvolvimento poderia ter evitado que esta vulnerabilidade fosse inserida no aplicativo.

FIG. 3.6: Aplicativo protegido

FIG. 3.7: Escaneamento sem vulnerabilidades

No segundo experimento, o aplicativo desenvolvido armazena as informações de acesso do usuário em um arquivo de preferências, reproduzindo uma vulnerabilidade classificada como Armazenamento Frágil. Esta aplicação utiliza os recursos da plataforma da interface *SharedPreferences* para armazenar os dados de usuário e senha no dispositivo, conforme a figura 3.8. Os dados de acesso somente são guardados se o *checkbox* "Salvar login" estiver marcado. Após efetuar a autenticação e ter os dados armazenados no arquivo de preferências, estes foram expostos ao se obter e acessar este arquivo. Isto foi possível pois o dispositivo estava com acesso *root* e de superusuário, violando neste caso a confidencialidade das informações.

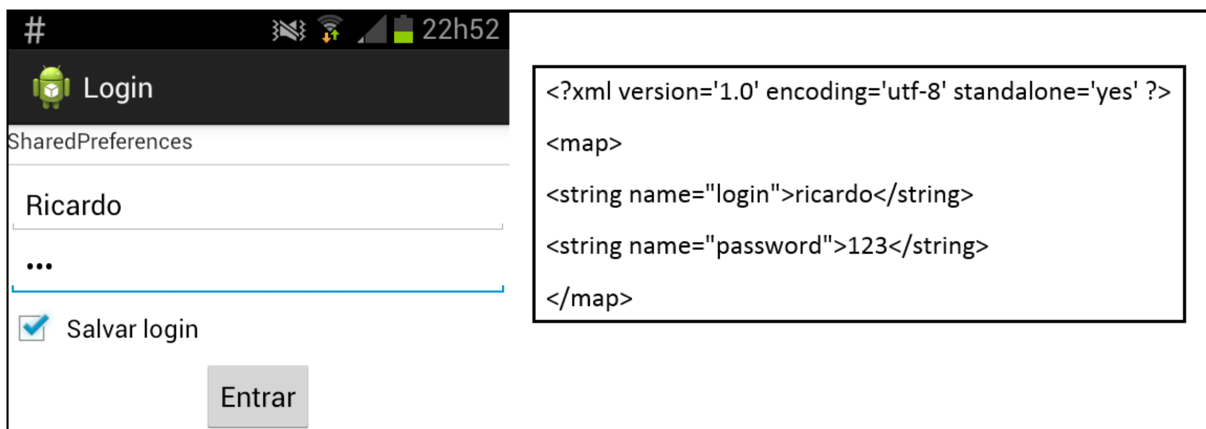

FIG. 3.8: Autenticação do aplicativo

Ao ser analisado pela appDroidAnalyzer, o código-fonte deste aplicativo foi considerado suscetível a esta vulnerabilidade caso o mesmo armazene informações sensíveis no arquivo de preferências, como visto na figura 3.9. Novamente, uma validação antecipada do código, durante o processo de desenvolvimento ajudaria o desenvolvedor a evitar que

esta situação ocorresse.

```
Scanning ExemploSharedPreferences: ......
src\br\exemplosharedpreferences\Login.java:21: Warning: Evite armazenar informações
utilizando getSharedPreferences sem criptografia.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao,
obtenha dados privados, violando os aspectos de confidencialidade, integridade
 ou disponibilidade. [GetSharedPreferencesDetector]
   SharedPreferences sp = getSharedPreferences(PREF_NAME, MODE_PRIVATE);
```

FIG. 3.9: Identificação de vulnerabilidade de armazenamento frágil

4 PROVA DE CONCEITO

Para validar a metodologia proposta no capítulo anterior, cujo objetivo é especificar uma classificação de vulnerabilidades que ajude os desenvolvedores para a plataforma Android a identificar possíveis vulnerabilidades em seu código-fonte, foram realizados testes com o código-fonte de aplicativos reais obtidos em um repositório de aplicativos *open source*[5].

Assim, este capítulo descreve esta prova de conceito e está organizado conforme a estrutura a seguir: a seção 4.1 demonstra um estudo de caso com dois aplicativos reais que motivaram este estudo; a seção 4.2 descreve como outros aplicativos foram selecionados para servirem de base para a verificação; já a seção 4.3 apresenta os resultados encontrados a partir da execução da ferramenta appDroidAnalyzer na base de aplicativos selecionado; e a seção 4.4 descreve a limitação deste trabalho.

4.1 ESTUDO DE CASO

Para compreender melhor as vulnerabilidades descritas nesta classificação e entender como elas se manifestam, serão analisados dois exemplos de aplicações que serviram para motivar este trabalho. O módulo nativo do Android *WappPushManager*, que é responsável por processar mensagens *WapPush*, permite que mensagens maliciosas sejam recebidas e processadas sem que sejam realizadas validações nestes dados de entrada. Ao utilizar o método *rawQuery* para executar a consulta ao banco de dados da aplicação, concatenando as cláusulas de restrição com os parâmetros recebidos, este módulo possibilita que ocorra Injeção de SQL, o que configura a vulnerabilidade Entrada Inválida desta classificação. Já o aplicativo Stick Cricket, um jogo com aproximadamente 200.000 *downloads* na loja Google Play[6], armazena as suas informações em arquivos de preferências, sem utilizar um algoritmo de criptografia para estes dados, como a pontuação máxima obtida pelo jogador. Isto permite que um atacante acesse este arquivo e altere a sua pontuação, o que irá configurar uma fragilidade classificada como Armazenamento Frágil.

[5] https://opensource.org/osd

[6] https://play.google.com/store/apps/details?id=com.sticksports.stickcricket

4.1.1 ENTRADA INVÁLIDA

WAPPushManager é o módulo da plataforma Android responsável por receber mensagens *WAPPush* e processá-las. Até a versão 5.0 do Android, este módulo possui a vulnerabilidade *SQL Injection* no banco de dados SQLite, que foi catalogada através da CVE-2014-8507[7], classificada pela *IBM X-Force Exchange* como crítica e de fácil exploração, impactando a confidencialidade, integridade e disponibilidade[8].

Para explorar esta vulnerabilidade, basicamente, o atacante envia uma mensagem *WAPPush* malformada para o telefone da vítima com a finalidade de abrir uma Atividade ou executar um serviço. Quando esta mensagem é recebida, ela é processada pelo método *dispatchWapPdu* da classe *com\android\internal\telephony\WapPushOverSms.java*. Este método recebe o identificador da aplicação e o conteúdo da mensagem e repassa estas informações para o método *queryLastApp*, que é responsável por executar uma consulta no banco de dados com estes parâmetros e retornar as informações necessárias para a classe devolver o serviço solicitado.

Este método é vulnerável porque utiliza a concatenação de *strings* para montar a consulta *SQL* que será executada pelo módulo, conforme a figura 4.1, permitindo que os parâmetros passados pela mensagem malformada façam acesso à um recurso do sistema que o atacante pode não possuir permissão. Esta vulnerabilidade foi corrigida a partir da versão 5.0 do Android, onde o método *queryLastApp* deixou de usar a concatenação de *strings* e passou a utilizar uma consulta parametrizada.

Código Vulnerável	**Código Corrigido e Seguro**
String sql = "select install_order, "	Cursor cur = db.query(APPID_TABLE_NAME,
+ " package_name, "	2 new String[] {"install_order", "package_name",
+ " class_name, app_type, "	"class_name", "app_type",
+ " need_signature, further_processing"	"need_signature",
+ " from " + APPID_TABLE_NAME	"further_processing"},
+ " where x_wap_application=\'" + app_id + "\'"	"x_wap_application=? and content_type=?",
+ " and content_type=\'" + content_type + "\'"	new String[] {app_id, content_type},
+ " order by install_order desc"; 1	null /* groupBy */, 1
Cursor cur = db.rawQuery(sql, null);	null /* having */,
2	"install_order desc" /* orderBy */);

FIG. 4.1: Análise do código do método *queryLastaApp*

[7]https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-8507
[8]https://exchange.xforce.ibmcloud.com/vulnerabilities/99008

No lado esquerdo da figura 4.1, podemos ver que o item identificado pelo número 1 concatena as informações recebidas pelos parâmetros *app_id* e *contente_type* com a cláusula *where* da consulta, confiando na informação recebida, e processa-a através do comando *rawQuery* identificado pelo número 2. O ataque será consumado caso o conteúdo recebido seja malicioso. Já no lado direito da figura vemos que o problema foi tratado e resolvido. O método *rawQuery* foi substituído pelo método *query*, e não existe mais a concatenação de *strings* dos parâmetros recebidos com a cláusula *where* da consulta. Esta situação foi substituída pela utilização dos parâmetros fornecidos pelo método *query*, onde as informações recebidas são repassadas para a consulta deste.

Quando submetido à análise da ferramenta appDroidAnalyzer, o código fonte deste módulo foi considerado suscetível a injeção de SQL, vulnerabilidade classificada como "Entrada Inválida", conforme visto na figura 4.2, onde aquela identificou e alertou ao desenvolvedor que utilizar consultas dinâmicas concatenando *strings* pode permitir que as informações sejam violadas, comprometendo a confidencialidade, a integridade ou a disponibilidade dos dados do sistema.

```
Scanning smspush: ...
src\com\android\smspush\WapPushManager.java:120: Warning: Evite consultas no SQLite sem passar parametros para a query.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao,
obtenha dados privados, violando os aspectos de confidencialidade, integridade
 ou disponibilidade. [SQLInjectionRawQueryDetector]
            Cursor cur = db.rawQuery(sql, null);
                         ~~~~~~~~~~~~~~~~~~~~~~
0 errors, 1 warnings
```

FIG. 4.2: Resultado da análise do *WAPPushManager*

4.1.2 ARMAZENAMENTO FRÁGIL

Stick Cricket é um popular jogo de *Cricket* que não necessita de uma conexão de rede com a internet para ser jogado, em outras palavras, pode ser executado *off line*. Esta característica faz com que seja necessário armazenar diversas informações do jogo no próprio dispositivo do usuário para que o jogador mantenha o seu desempenho e seu histórico guardados.

Uma das informações que este jogo mantém no dispositivo é a maior pontuação obtida pelo jogador. Para armazenar estas informações, este aplicativo utiliza os recursos nativos da classe *sharedPreferences* da plataforma, salvando estes dados no arquivo de preferências *Cocos2dxPrefsFile.xml* que fica localizado no diretório */data/data/com.sticksports.stickcricket/shared_prefs*. A figura 4.3 apresenta a tela inicial do aplicativo com a pontuação obtida pelo usuário em destaque e ao seu lado, o fragmento do arquivo de preferências

com a identificação do trecho que armazena esta pontuação.

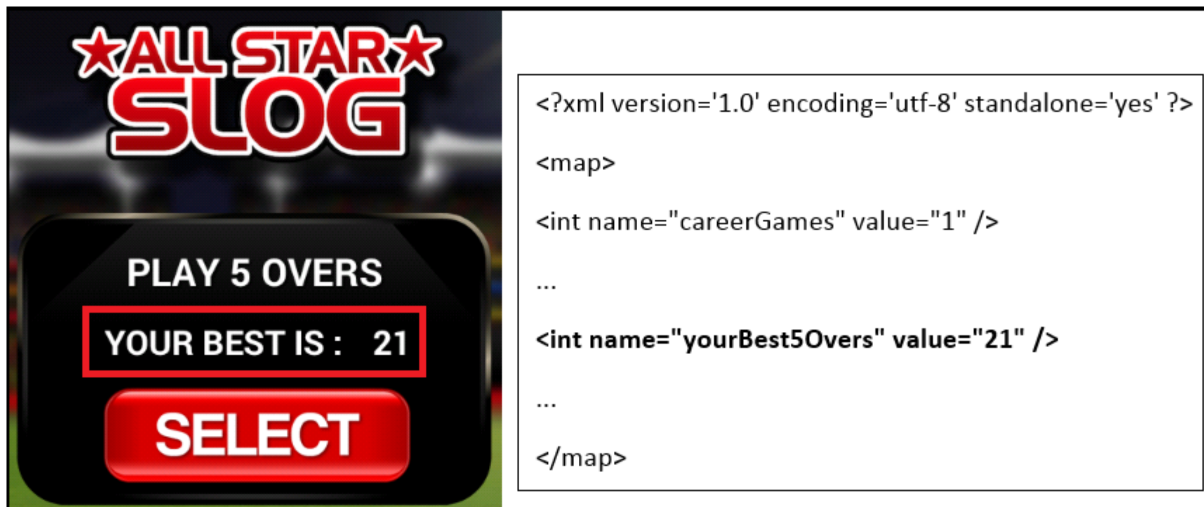

FIG. 4.3: Pontuação original do jogo *Stick Cricket*

Como pôde ser visto na figura 4.3, este aplicativo não utiliza nenhum método de criptografia para armazenar as informações de forma segura, todas os dados são mantidos em texto em claro. Esta situação permite que um usuário mal-intencionado possa alterar as informações deste arquivo, como por exemplo, para aumentar a sua pontuação e obter alguma vantagem a partir disto.

Para obter esta vantagem é necessário apenas que o usuário possua acesso *root* ao dispositivo e permissões de superusuário, que podem ser obtidas de forma simples, apenas com informações de tutorias na internet, e assim alterar o arquivo Cocos2dxPrefsFile.xml. A figura 4.4 exibe o resultado da tela inicial do aplicativo após o mesmo ter a sua pontuação original alterada de 21 para 190, e ao lado, o trecho do arquivo de preferências que foi modificado.

Esta situação configura a exploração da vulnerabilidade *Insecure Data Storage*, classificada como "Armazenamento Frágil", e poderia ser evitada se, durante o processo de desenvolvimento, o desenvolvedor estivesse consciente de que este problema poderia ocorrer, e que a confidencialidade e a integridade dos dados poderiam ser perdidas. Ao submeter o código-fonte deste aplicativo à análise da appDroidAnalyzer, a ferramenta identificou e alertou que usar arquivo de preferências sem criptografia pode permitir que as informações sejam reveladas, conforme pode ser visto na figura 4.5.

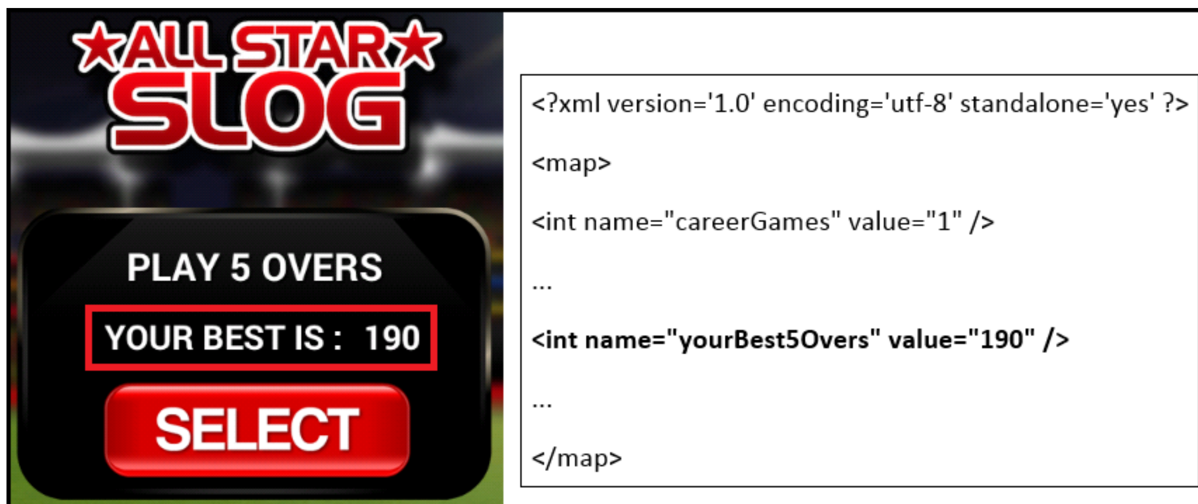

FIG. 4.4: Pontuação modificada no *Stick Cricket*

```
Scanning StickCricket: .....
        - appcompat_v7: ..
src\com\sticksports\stickcricket\Cocos2dxHelper.java:238: Warning: Evite armazenar informações utilizando getSharedPreferences sem criptografia.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao,
obtenha dados privados, violando os aspectos de confidencialidade, integridade
 ou disponibilidade. [GetSharedPreferencesDetector]
        return ((Activity) sContext).getSharedPreferences(PREFS_NAME, 0).getBoolean(key, defaultValue);
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
src\com\sticksports\stickcricket\Cocos2dxHelper.java:242: Warning: Evite armazenar informações utilizando getSharedPreferences sem criptografia.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao,
obtenha dados privados, violando os aspectos de confidencialidade, integridade
 ou disponibilidade. [GetSharedPreferencesDetector]
        return ((Activity) sContext).getSharedPreferences(PREFS_NAME, 0).getInt(key, defaultValue);
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
src\com\sticksports\stickcricket\Cocos2dxHelper.java:246: Warning: Evite armazenar informações utilizando getSharedPreferences sem criptografia.
```

FIG. 4.5: Trecho do resultado do *Stick Cricket*

4.2 SELEÇÃO DE PROJETOS PARA ANÁLISE

Para selecionar os projetos que farão parte da avaliação deste método, foi empregada uma seleção que não se baseou em características específicas dos aplicativos. O critério utilizado para que um projeto estivesse apto a ser selecionado, respeitando o paradigma deste trabalho, foi escolher aplicativos que sejam classificados como *software* livre e de código aberto. Além disto, estes deveriam estar catalogados em um repositório específico para aplicativos para Android. Respeitando estas premissas, o repositório F-Droid foi escolhido por atender a todos os requisitos definidos anteriormente. Este repositório é um banco de dados de aplicativos classificados como *Free and Open Source Software* (FOSS)[9]

Para realizar a validação e posterior apuração dos resultados deste trabalho, foram selecionados 1338 aplicativos catalogados no repositório F-Droid. Contudo este não armazena os códigos fonte dos projetos, apenas mantém uma página com as características do aplicativo e um *link* para a página que contém os arquivos do projeto, como por exem-

[9]https://f-droid.org/

plo, GitHub[10], GoogleGit[11], Sourceforge[12], dentre outros. Possui aproximadamente 2.000 aplicativos registrados.

A partir desta lista de projetos foi desenvolvida uma rotina para realizar o *download* de cada um dos aplicativos. Esta rotina é composta de três passos realizados em sequência. Inicialmente foi criado um código JavaScript para obter a URL da página principal de cada projeto. Este código percorreu todas as páginas do repositório F-Droid e exportou para um arquivo texto o endereço da página de cada aplicativo, como por ser visto na tabela 4.1 uma fração deste resultado.

TAB. 4.1: Lista de endereços de projetos Android

Endereço
https://f-droid.org/wiki/page/a2dp.Vol
https://f-droid.org/wiki/page/aarddict.android
https://f-droid.org/wiki/page/acr.browser.barebones
https://f-droid.org/wiki/page/acr.browser.lightning
https://f-droid.org/wiki/page/akk.astro.droid.moonphase
https://f-droid.org/wiki/page/am.ed.exportcontacts

De posse desta lista, foi necessário visitar cada um destes endereços para que fosse possível obter o código-fonte dos aplicativos. Para atender a este objetivo, devido ao grande volume de endereços, foi desenvolvida uma aplicação na plataforma Outsystems[13] para realizar o *download* dos códigos fonte. Esta plataforma foi escolhida porque, além de possuir uma versão gratuita, é muito fácil e rápido construir aplicações nesta tecnologia.

Assim, para cada aplicativo listado, esta aplicação enviou uma solicitação HTTP para o endereço relacionado e, ao obter a página de retorno, foi feito *parse* no resultado para identificar o local onde o projeto está hospedado. De acordo com este retorno foi decidido se o aplicativo faria parte ou não da validação desta pesquisa, pois muitos destes endereços possuem um *link* para uma página de hospedagem inválida. Desta forma, foram selecionados e baixados os códigos-fonte dos aplicativos que estão hospedados no GitHub ou GoogleGit e que possuem uma referência válida para *download*.

Esta decisão foi tomada porque estes repositórios concentram a maior parte dos projetos catalogados na base de dados do F-Droid e durante este processo foi identificado que muitos *links* para outros repositórios estavam inoperantes. Outro motivo desta escolha reside no fato de que estes repositórios mantêm um padrão para formação do endereço

[10] https://github.com/
[11] https://android.googlesource.com/
[12] https://sourceforge.net/
[13] https://www.outsystems.com/

para *download* do projeto. Por exemplo, no GitHub, o endereço para baixar o pacote *Master* de um projeto é composto pelo endereço do GitHub seguido pelo identificador do aplicativo mais a sequência de caracteres "/archive/master.zip". Já no GoogleGit, de forma similar, o endereço para *download* é formado pelo prefixo do repositório acrescido do nome do aplicativo e seguido pela *strings* "/+archive/master.tar.gz". Está especificação está representada na tabela 4.2.

TAB. 4.2: Exemplos de páginas para *download* de aplicativos

Endereço
https://github.com/jroal/a2dpvolume/archive/master.zip
https://github.com/aarddict/android/archive/master.zip
https://android.googlesource.com/platform/packages/apps/Music/+archive/ master.tar.gz

Tendo em vista que os arquivos dos projetos são baixados compactados, na forma de um único arquivo, uma terceira rotina foi executada para extrair o código-fonte destes aplicativos. Para esta etapa foi criado uma rotina em *Visual Basic Script* (VBScript) que, através de linha de comando, percorreu o diretório onde os arquivos compactados estavam armazenados e, para cada um, executou um comando que abriu o aplicativo Winrar e descompactou os projetos baixados.

4.3 APURAÇÃO DOS RESULTADOS

Conforme visto na seção 4.1, o módulo *WAPPushManager* e o aplicativo *Stick Cricket* foram submetidos à análise da ferramenta appDroidAnalyzer. Ficou evidenciado que a aplicação de um método que ajude, previamente, o desenvolvedor a manter aplicações mais seguras pode alcançar este resultado se ele estiver apoiado por uma ferramenta automatizada para a identificação de vulnerabilidades.

Desta forma, após esta primeira análise, os 1.338 aplicativos que foram selecionados e obtidos através da base de dados de aplicativos *open-source* F-Droid foram submetidos para serem verificados pela ferramenta appDroidAnalyzer. Esta validação foi realizada em duas fases. Na primeira etapa, buscou-se identificar quais aplicativos possuíam uma potencial vulnerabilidade em relação à classificação "Armazenamento Frágil". No segundo ciclo de verificação a ferramenta procurou identificar vulnerabilidades relacionadas com a classificação "Entrada Inválida".

Em consequência de haver um grande número de aplicativos para serem analisados, o que torna o processo manual inviável, foi desenvolvida, novamente em VBScript, uma

rotina que executou em cada um dos projetos, via linha de comando, as verificações utilizadas pela ferramenta appDroidAnalyzer a fim de validar a metodologia descrita neste trabalho. Esta rotina armazenou o resultado desta validação em um arquivo que serviu como base para apuração dos resultados.

Durante as duas fases de validação, diversos aplicativos não foram verificados porque a ferramenta Lint, que foi utilizada como base para desenvolvimento da appDroidAnalyzer, possui uma incompatibilidade para analisar alguns projetos Android desenvolvidos com base na versão do ADT 19 ou anterior[14]. Dentre os 1.338 projetos selecionados, 481 apresentaram este problema (*No .class files were found in Project*) e foram descartados da análise, restando 857 aplicativos verificados.

Na primeira etapa de validação, onde buscou-se identificar os aplicativos com uma vulnerabilidade classificada em “Armazenamento Frágil”, foram encontrados 51 aplicativos suscetíveis a esta fragilidade, todos relativos a vulnerabiliade *M2 - Insecure Data Storage*. Já na segunda etapa, 10 aplicativos foram identificados com a vulnerabilidade classificada como “Entrada Inválida”. Destes, 1 apresentou a vulnerabilidade *M8 - Security Decisions Via Untrusted Inputs* e 9 apresentaram *M7 - Client Side Injection*. Por fim, de todos os aplicativos que foram analisados, 2 apresentaram ser suscetíveis às duas classificações, onde foram encontradas as vulnerabilidades M2 e M7. Somando-se à esta validação os dois estudos de casos descritos na seção 4.1, 54 aplicativos possuem “Armazenamento Frágil” e 13 possuem a vulnerabilidade “Entrada Inválida”, conforme figura 4.6.

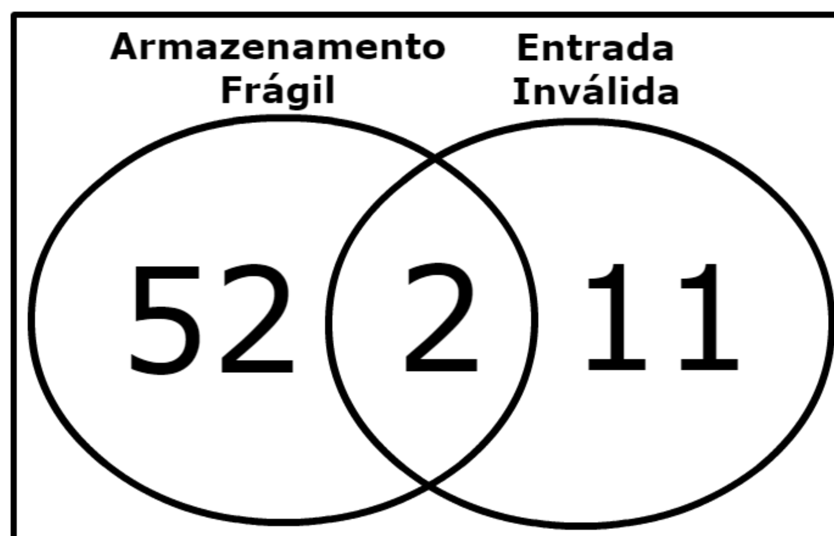

FIG. 4.6: Resultado apurado

Este estudo demonstrou que, no total, 65 aplicativos de todos os que foram sub-

[14]https://code.google.com/p/android/issues/detail?id=28278

metidos à verificação da ferramenta desenvolvida apresentaram pelo menos um tipo de vulnerabilidade das que foram classificadas nesta pesquisa, representando 7,6% do total. A relação destes aplicativos pode ser visualizada no Apêndice 1.

5 TRABALHOS RELACIONADOS

Este capítulo apresenta uma comparação entre os trabalhos que estão relacionados com este estudo. A seção 5.1 apresenta a descrição dos trabalhos que abordam o problema antes do aplicativo ter sido finalizado, chamada de "detecção antecipada". A seção 5.2 descreve os trabalhos que analisam as vulnerabilidades após a conclusão do sistema, aqui identificados como "detecção tardia". Na seção 5.3 é apesentado um comparativo entre os trabalhos relacionados com a metodologia utilizada neste estudo.

5.1 ABORDAGEM DE DETECÇÃO ANTECIPADA

Esta seção apresenta os trabalhos relacionados ao tema deste estudo e que abordam o problema utilizando a detecção antecipada de vulnerabilidades. Em Souza (2015) foram identificados e analisados dois problemas em relação a segurança de software: as ferramentas existentes para validação de vulnerabilidades atuam somente no final do processo de desenvolvimento de *software*, quando a aplicação está pronta; e a técnica de casamento de padrões de código gera um alto número de falsos positivos.

Segundo o autor, o fato desta análise ocorrer de forma tardia acarreta em uma manutenção custosa do programa, uma vez que o desenvolvedor, nesta fase, pode ter perdido o contexto de detalhes do *software* que foi desenvolvido há dias ou semanas atrás. Já em relação à técnica de casamento de padrões é que este mecanismo gera uma alta taxa de falsos positivos.

Inicialmente, a análise de Souza (2015) consistiu em validar se a detecção antecipada de vulnerabilidades (durante a etapa de desenvolvimento) ajudaria o desenvolvedor a melhorar a qualidade do seu código em termos de segurança. O segundo experimento serviu para validar se a técnica Análise de Fluxo de Dados (*Data Flow Analysis* - DFA), utilizada por compiladores, diminuiria a taxa de falsos positivos quando comparado com a utilização de casamento de padrões de código.

Para validar estas hipóteses, o autor desenvolveu, na linguagem de programação Java, um *plug-in* para a IDE Eclipse chamado ESVD (*Early Security Vulnerability Detector*). Esta ferramenta foi criada para identificar as vulnerabilidades classificadas pela lista *OWASP - Top Ten 2013* e foi baseada na técnica de Análise de Fluxo de Dados.

Em um primeiro experimento, que serviu para validar se a técnica utilizada reduz

a quantidade de falsos positivos, seis projetos de *software* foram analisados pela ESVD e por outras três ferramentas que implementam a técnica de casamento de padrões. O estudo concluiu que a análise de fluxo de dados reduz significativamente a taxa de falsos positivos quando comparada com a outra técnica.

Para validar se a detecção antecipada ajuda o desenvolvedor a corrigir as vulnerabilidades de um programa, o autor realizou um experimento com 27 participantes, que foram divididos em dois grupos denominados "*Early*" e "*Late*". O primeiro grupo foi composto por 14 desenvolvedores, enquanto que o segundo continha 13 pessoas. Para cada um destes participantes foi entregue uma lista com cinco tarefas a serem concluídas em 90 minutos, dentre as quais existiam, por exemplo, criar uma página de "*login*". Após finalizar esta atividade, Souza (2015) verificou que o grupo "*Early*" terminou as suas tarefas em um tempo menor que o outro grupo, e que os participantes do grupo "*Late*" introduziram muito mais defeitos em suas aplicações.

Baseado neste estudo chegou-se à conclusão que a detecção antecipada ajuda o desenvolvedor a prevenir ou remover as vulnerabilidades encontradas no código, fazendo com que o programador tenha consciência das vulnerabilidades existentes e os incentive a corrigi-las prontamente.

Outra abordagem para detectar problemas de forma antecipada foi utilizada em Sağlam (2014), que partiu da hipótese de que a causa de muitos aplicativos existentes na Google Play possuírem uma baixa avaliação e consequentemente reputação ruim é o fato de que muitos deles possuem problemas de lentidão ocasionados pelo mal gerenciamento da memória. Em contrapartida, os aplicativos com uma boa avaliação possuem algumas características em comum, como rapidez para iniciar e abrir atividades, estabilidade durante o uso (não congelar ou fechar inesperadamente) e apresentar rápida resposta a cada solicitação de um serviço pelo usuário. Para classificar um aplicativo com baixa reputação, foi utilizado como critério o mesmo ter recebido uma avaliação com uma ou duas estrelas na loja da Google, enquanto que para receber uma boa reputação o app deverá ter recebido quatro ou cinco estrelas.

Inicialmente, o autor identificou quais os métodos Java para a plataforma Android estão relacionados com os problemas referentes ao gerenciamento ruim de memória e em seguida, estes foram classificados em quatro tipos: *Using non-static inner classes*, *Not setting thread priorities*, *Not using a cancellation policy in a thread* e *Not reusing views in list view*. Para identificar estas más-práticas, foi desenvolvida a ferramenta *Bad-Practice Finder*, baseada em Lint, que percorre o código fonte de aplicativos Android. Esta ferramenta procura por códigos que possam ocasionar lentidão devido aos problemas de

gerenciamento de memória classificados em uma destas quarto categorias.

Foram submetidas a análise desta ferramenta 100 aplicativos baixados de um repositório *open-source* de aplicativos para Android. Para esta seleção o autor comparou as informações de 932 aplicativos baixados daquele repositório com a sua reputação na loja Google Play, e escolheu os que obtiveram as piores avaliações.

Após obter os resultados da análise anterior, Sağlam (2014) concluiu que uma boa reputação na Google Play cresce exponencialmente a partir das avaliações dos usuários e das suas percepções em relação à lentidão, estabilidade e baixo consumo de memória e que a melhora desta percepção está diretamente relacionada com a remoção de práticas ruins de codificação.

5.2 ABORDAGEM DE DETECÇÃO TARDIA DE VULNERABILIDADES

Procurar por vulnerabilidades de forma tardia significa utilizar um método para analisar uma aplicação após a mesma estar pronta e disponível para utilização. Neste sentido, em Cheng et al. (2013) foi proposto um método para detectar comportamentos perigosos em aplicações Android. Este método consistiu em analisar o *dex bytecode* dos aplicativos selecionados, através de um processo que simula a execução dos aplicativos, para identificar cinco tipos diferentes de vulnerabilidades:

a) *Service Provider services ordering*: compartilhamento de informações através de mensagens de texto para um número de um provedor de serviços;

b) *Privacy leak*: vazamento de informações confidenciais do usuário;

c) *Charges consumption*: consumo de recursos que causem danos financeiros, como por exemplo, envio de SMS;

d) *Native code*: execução de comandos do sistema operacional ou referências diretas às bibliotecas nativas do sistema operacional; e

e) *Constant URL connection*: utilização de URLs fixas no código para conexão.

Para realizar esta simulação, foi desenvolvido, nas linguagens de programação Python e Java, o protótipo *ApkRiskAnalyzer* que é composto de três fases. Na primeira fase, o pacote da aplicação (APK - *Android Package*) é desempacotado para que seja extraído o *dex bytecode* da aplicação. O *dex bytecode* é armazenado em um SGBD MySQL. Estes arquivos são armazenados para que sirvam como entrada para o processamento que ocorrerá na segunda fase do processo.

Na fase seguinte, é realizada a simulação dos processos de cada *dex bytecode* que foi armazenado no SGBD durante a fase anterior. Esta simulação é realizada a partir de dois

mecanismos. No mecanismo *Taint Analysis* é feita uma análise para identificar potenciais comportamentos perigosos a partir do uso incorreto de dados sensíveis, como por exemplo, a lista de contatos do usuário. No mecanismo *Constant Analysis* o simulador verifica como o aplicativo utiliza as informações que estão armazenadas em variáveis do tipo constante, como por exemplo, conteúdos de mensagens de texto e números de telefones para enviar estas mensagens para solicitar algum serviço. Desta forma, um aplicativo malicioso pode utilizar um outro app inseguro para enviar uma mensagem de texto para um número e conteúdo armazenados em constantes para requisitar um serviço a que não tenha permissão.

Finalmente, na terceira fase que é utilizada para detectar os riscos dos aplicativos, é realizada uma análise estática para comparar os comportamentos identificados na fase anterior com uma base de dados de regras vulneráveis previamente cadastradas pelos autores. Ao encontrar um casamento de padrões entre esta base de conhecimento e o comportamento inseguro encontrado, o risco é inserido em uma das cinco classificações sugeridas: *Service Provider services ordering*, *Privacy leak*, *Charges consumption*, *Native code* ou *Constant URL connection*.

Este método foi aplicado em 1260 aplicativos vulneráveis e o *ApkRiskAnalyzer* identificou 1246 apps com comportamentos suspeitos. Em seguida, 630 aplicações foram obtidas na loja Google Play, e deste total, em 91% também foram identificadas estas práticas, ou seja, em 575 apps.

Outra abordagem, ainda através do arquivo executável *dex bytecode*, foi utilizada por Wu et al. (2014), que inicia o seu trabalho definindo que a plataforma Android permite que aplicações exponham componentes para utilização, de forma colaborativa, de outros aplicativos. Como exemplo, os autores citaram o componente para captura de imagens através da câmera fotográfica, que fica exposto para utilização de outros aplicativos. Este mecanismo permite que uma aplicação envie uma requisição ou uma entrada maliciosa para um outro componente disponível no sistema ou para outra aplicação. Este conceito foi chamado pelos autores de *Exposed Component Vulnerability* (ECV).

Neste trabalho, Wu et al. (2014) apresenta uma abordagem mais efetiva para identificar e classificar as vulnerabilidades que compreendem vazamento de informações contidas em ECVs em Android. Inicialmente, foi realizada uma análise para criar uma classificação e um catálogo de regras de sintaxe baseadas em uma combinação de métricas (por exemplo, semântica de permissões, nomes de APIs). Este processo forneceu um conjunto de dados *VSinks*, que é uma abreviação para *Vulnerability-specific Sink* e especifica um conjunto de ataques relacionados com as ECVs.

Após esta classificação, uma abordagem apoiada em mais três etapas foi utilizada. Na primeira etapa é utilizado um algoritmo que realiza uma análise intra-procedural iterativa do aplicativo para que seja criado um mapeamento do controle de fluxo do aplicativo. Isto tem como objetivo criar a ordem com que será realizada a análise do código-fonte do aplicativo, de acordo com a estratégia que foi definida.

Em seguida, apoiado em uma análise de fluxo de dados criado no passo anterior, a vulnerabilidade é classifica em uma das quatro categorias descritas no trabalho. Em *VS_Direct* serão categorizadas as vulnerabilidades relacionadas com privilégios de acesso aos recursos do sistema, como por exemplo, se um app que envia uma mensagem SMS possui permissão para tal. Já *VS_DirectByPerm* é similar à *VS_Direct*, porém neste caso serão considerados os parâmetros enviados para um método sob ataque. Nesta situação, diferentes parâmetros causam diferentes ataques. *VS_Input* é a categoria que agrupa os *VSinks* que fazem mau uso de recursos privilegiados, com ou sem a entrada de parâmetros, como por exemplo, vazamentos relacionados com a internet (HttpClient.execute). Uma vez que as entradas maliciosas sejam utilizadas, este método pode ser usado indevidamente para acessar os recursos da internet. Por fim, na categoria *VS_Public* estão as vulnerabilidades que acarretam em vazamento de informações, mas que não estão relacionadas diretamente com os privilégios do aplicativo. Neste cenário os dados são coletados de outras formas, como por exemplo, coordenadas do GPS gravadas em arquivo de log.

Finalmente, para cada ECV classificada em *VS_DirectByPerm* e *VS_Input*, são removidos os falsos positivos a partir de uma análise semiguiada, que utiliza os parâmetros utilizados nos métodos considerados vulneráveis.

Para validar estes conceitos, 1.000 aplicativos foram submetidos à análise do ECVDetector, que foi uma ferramenta desenvolvida a partir da linguagem de programação Java e de *scripts* Python. Após o experimento, 49 aplicativos analisados foram considerados vulneráveis (4,9%).

Já em Kim et al. (2012) foi desenvolvida uma ferramenta, denominada ScanDal, que implementa a técnica de análise estática com a finalidade de encontrar vazamento de informações em aplicativos Android, em outras palavras, o autor procura detectar diversas possibilidades de falhas que irão permitir o compartilhamento de informações sensíveis.

Kim et al. (2012) classificou, para utilizar em sua análise, dois tipos de problemas que afetam as informações. APIs que retornam informações sensíveis de recursos do sistema, como a localização física do usuário, identificadores do telefone (IMEI, número de série dentre outros) e dados de áudio e vídeo; e as APIs que transferem dados pela rede, como o envio de um arquivo através da internet ou o envio de um SMS.

Para atingir o objetivo proposto, ScanDal analisa o aplicativo em três fases distintas que ocorrem em sequência. O processo começa com a fase *Front-end* que realiza a extração do *dex bytecode* obtido a partir do APK do aplicativo. Em seguida é realizada a fase *Translation* de conversão do arquivo executável, onde este é traduzido para uma linguagem intermediária, mais simples e eficiente, uma vez que o arquivo original possui mais de 220 instruções *Dalvik*. Esta linguagem foi criada pelo autor do trabalho e é chamada de *Dalvik Core*. Por último, na fase *Abstract Interpretation*, ocorre a interpretação desta linguagem para que sejam identificados os trechos vulneráveis do código-fonte.

ScanDal analisou 90 aplicativos gratuitos obtidos na loja Google Play e detectou o vazamento de informações em 12% deles. Destes 11 apps, 6 enviavam dados de localização do usuário para servidores de propagandas, 5 armazenavam estas informações em arquivos e 1 anviava o IMEI do dispositivo para um servidor remoto. Em seguida, 8 apps obtidos em loja de terceiros, conhecida por armazenar aplicativos modificados, foram submetidos a ScanDal. Em todos estes foi identificado o vazamento de informações de localização e dados do dispositivo, como o IMEI.

5.3 ANÁLISE DOS TRABALHOS RELACIONADOS

Esta seção apresenta uma análise comparativa entre esta pesquisa com os outros trabalhos relacionados. Primeiramente, o que diferencia o presente trabalhos dos demais está no fato de que nenhum define uma classificação que correlacione os problemas de segurança com os princípios definidos pela norma ABNT NBR ISO/IEC 27002:2013, confidencialidade, integridade e disponibilidade.

Em relação ao trabalho de Souza (2015) a principal diferença está caracterizada pela plataforma utilizada, onde este utilizou uma avaliação direcionada para aplicações Web, o que faz com que o seu conteúdo não seja aplicável à plataforma Android. A complexidade para validar o código para Android é diferente da validação realizada em um código Java para Web.

Outro fator reside no fato de que em Android existe um mapeamento da interface, criada em XML (*Extensible Markup Language* - Linguagem de Marcação Extensível), para um objeto, e a partir deste objeto a linguagem Java é utilizada para manipular os atributos e comportamentos deste, com classes e métodos específicos desta plataforma. Já na linguagem Java para desenvolvimento Web não existe este mapeamento anterior. Além desta particularidade, outra característica que diferencia o Android de Web é que a plataforma móvel permite que o desenvolvedor utilize recursos do dispositivo, como o

acesso de leitura e escrita a agenda de contatos, recurso que não existe na Web.

Apesar de Sağlam (2014) tratar os problemas de forma antecipada e criar uma extensão do Lint, este trabalho não se preocupou com a segurança dos apps. O autor identifica problemas de desempenho relacionados com o gerenciamento de memória e a sua relação com a reputação do aplicativo nas lojas oficiais.

Por último, em Cheng et al. (2013), Wu et al. (2014) e Kim et al. (2012), a principal diferença reside no fato de que nestes a análise do aplicativo não é feita através do código-fonte, mas é realizada a partir do *dex bytecode*, após este estar pronto e publicado em uma loja oficial. Nestes casos, não há a prevenção para evitar que potenciais vulnerabilidades sejam inseridas durante o processo de desenvolvimento do aplicativo, pois o código-fonte não é analisado durante esta fase, de forma antecipada.

Estas comparações podem ser visualizadas na tabela 5.1.

TAB. 5.1: Comparativo entre trabalhos.

	Plataforma Android	Detecção Antecipada	Evita vulnerabilidade	Correlata CID	Códido-Fonte	Relacionado à segurança
Cheng et al.	✓	✗	✗	✗	✗	✓
Kim et al.	✓	✗	✗	✗	✗	✓
Saglam	✓	✓	✓	✗	✓	✗
Souza	✗	✓	✓	✗	✓	✓
Wu et al.	✓	✗	✗	✗	✗	✓
Ferreira et al.	✓	✓	✓	✓	✓	✓

Em termos numéricos, Cheng et al. (2013) identificou 91% de comportamentos suspeitos nos aplicativos analisados, enquanto que Wu et al. (2014) identificou 4,9% apps vulneráveis, ao passo que Kim et al. (2012) encontrou 12%. Na prova de conceito realizada por este trabalho 7,6% dos aplicativos submetidos à análise da appDroidAnalyzer possuíam vulnerabilidades. Em relação ao trabalhos de Souza (2015) e Sağlam (2014) não é possível identificar o percentual devido à finalidade de cada uma destas pesquisas. A tabela 5.2 apresenta esta comparação.

TAB. 5.2: Comparativo entre trabalhos.

	Total	Encontrados	%
Cheng et al.	630	575	91%
Wu et al.	1000	49	4,9%
Kim et al.	90	11	12%
Saglam	-	-	-
Souza	-	-	-
Ferreira et al.	859	65	7,6%

6 CONSIDERAÇÕES FINAIS

6.1 CONCLUSÃO

A procura para encontrar vulnerabilidades em aplicativos para a plataforma Android, que afeta 90% destes, tem sido abordada tardiamente, após os mesmos estarem publicados para *download* em lojas, onde milhares de usuários já podem ter sido afetados. Por este motivo, é de grande importância que estas fragilidades sejam identificadas durante o processo de desenvolvimento, enquanto o desenvolvedor está codificando o app.

A classificação de vulnerabilidades proposta neste trabalho permite ao desenvolvedor conhecer as vulnerabilidades, os tipos de código que permitem a introdução destas e a sua relação com os atributos de segurança da informação confidencialidade, integridade e disponibilidade. Ao utilizar esta metodologia, o desenvolvedor terá a oportunidade de identificar, analisar e corrigir o código-fonte do aplicativo, antecipadamente, para evitar que as vulnerabilidades sejam inseridas no app, preservando aqueles atributos de segurança.

Através da prova de conceito realizada nesta pesquisa foi possível encontrar 65 aplicativos que possuem potenciais vulnerabilidades. Isto demonstra que identificar as fragilidades de um app com o apoio de uma ferramenta, enquanto o mesmo está sendo codificado pelo desenvolvedor, permite que estas sejam evitadas, uma vez que, naquela oportunidade, ele possui o contexto do que ele está desenvolvendo.

6.2 CONTRIBUIÇÕES

Este trabalho contribui com a definição de um método para fornecer suporte para que o desenvolvedor de aplicativos para a plataforma Android identifique, analise e corrija o código-fonte, de forma antecipada, produzindo aplicativos mais seguros. As contribuições específicas desta pesquisa foram divididas da seguinte forma:

6.2.1 CLASSIFICAÇÃO DE VULNERABILIDADES PARA DESENVOLVIMENTO SEGURO EM ANDROID

Este trabalho forneceu como uma de suas contribuições a definição de uma classificação de vulnerabilidades baseada em Ferreira (2016). Esta classificação identificou as vulnerabilidades que ocorrem em dispositivos móveis com o SO Android e as dividiu em "Entrada Inválida" e "Armazenamento Frágil".

"Entrada Inválida" está relacionada com a validação de dados recebidos pelo app e inclui *Client Side Injection*, *Security Decisions Via Untrusted Inputs* e com parte de *Poor Authorization and Authentication.* "Armazenamento Frágil" é referente a forma de armazenamento de informações do app e engloba *Insecure Data Storage*, *Unintended Data Leakage* e a outra parte de *Poor Authorization and Authentication.* Após esta divisão, foi identificado como estas vulnerabilidades ocorrem no Android e quais os métodos em Java que permitem que estas fragilidades sejam inseridas.

De posse destas informações, estas foram associadas com os aspectos de segurança definidos pela norma ABNT NBR ISO/IEC 27.002:2013 que serão violados caso uma destas vulnerabilidades seja explorada. Esta classificação foi descrita na seção 3.1.

6.2.2 APPDROIDANALYZER

Fundamentado nesta classificação de vulnerabilidades, outra contribuição produzida por este trabalho foi o desenvolvimento de uma ferramenta de apoio, chamada appDroidAnalyzer, que identificou as vulnerabilidades e alertou que o código possuía uma brecha. Esta ferramenta implementa a técnica de análise estática com casamento de padrões e foi desenvolvida a partir de uma extensão do Lint. appDroidAnalyzer foi especificada na seção 3.3.

6.3 TRABALHOS FUTUROS

Uma das sugestões de trabalhos futuros para continuar esta pesquisa é estudar outros métodos e classes Java que possam permitir que as vulnerabilidades descritas sejam exploradas, e incorporá-las no validador criado neste trabalho. Isto tornará a ferramenta mais completa e robusta, permitindo ao desenvolvedor, de aplicativos para a plataforma Android, criar e manter produtos mais seguros.

Outra proposta para dar continuidade à esta pesquisa seria incorporar a ferramenta appDroidAnalyzer ao Android Studio ou ao Eclipse. Como nesta pesquisa o método foi

avaliado utilizando uma grande quantidade de aplicativos, a automatização do processo de validação através de linha de comando foi necessária. Com isto, a execução do validador através de uma IDE não foi utilizada. Incorporá-la a estas IDEs traria, para o desenvolvedor, o benefício de verificar as vulnerabilidades do seu código através de uma interface visual, em tempo de desenvolvimento.

Uma outra sugestão seria adaptar este trabalho para fazer uso de uma outra técnica de análise de código diferente da análise estática com casamento de padrão, que foi utilizada nesta pesquisa. Esta modificação traria a possibilidade de se fazer uma avaliação em relação aos resultados falso-positivos, que não foram abordados neste estudo por já ser um assunto bastante difundido na literatura em relação ao método de análise utilizado (SOUZA, 2015) (CHENG et al., 2013) (NADEEM et al., 2012).

Por fim, outra proposta para dar continuidade a este trabalho seria obter uma quantidade maior de aplicativos, transferidos de outros repositórios que não o F-Droid, com a finalidade de se alcançar uma base de avaliação maior. Isto teria como objetivo conseguir um resultado mais expressivo em relação a quantidade total de aplicativos existentes no mercado, hoje, aproximadamente 2430 mil apps.

7 REFERÊNCIAS BIBLIOGRÁFICAS

ABNT, N. **ABNT ISO/IEC 27.002 Tecnologia da informação — Técnicas de segurança — Código de Prática para Controles de Segurança da Informação**. 1. ed. Rio de Janeiro: Associação Brasileira de Normas Técnicas, 2013.

ALBUQUERQUE, R.; RIBEIRO, B. **Segurança no Desenvolvimento de Software– Como desenvolver sistemas seguros e avaliar a segurança de aplicações desenvolvidas com base na ISO 15.408**. 1. ed. Rio de Janeiro: Editora Campus. Rio de Janeiro, 2002. 310 p.

APPBRAIN. Android Statistics - Number of Android applications. Disponível em: <http://www.appbrain.com/stats/number-of-android-apps>. Acesso em: 14 set. 2016.

CHENG, S.; LUO, S.; LI, Z.; WANG, W.; WU, Y. ; JIANG, F. Static detection of dangerous behaviors in android apps. **Lecture Notes in Computer Science**, v. 8300, p. 363–376, 2013. Disponível em: <http://link.springer.com/chapter/10.1007%2F978-3-319-03584-0_27>. Acesso em: 04 jun. de 2015.

CRESPO, M. A. C.; CHÓEZ, R. E. R. **Estudio del impacto financiero de las vulnerabilidades de las páginas Web de los bancos en Ecuador**. 2012. 193 f. Trabalho de Conclusão de Curso (Graduação em Ingenieria de Sistemas) – Universidad Politécnica Salesiana, Guayaquil, 2012.

DANTAS, M. L. **Segurança da Informação: uma abordagem focada em gestão de riscos**. 1. ed. Receife: Livro Rápido - Ecológica, 2011. 152 p.

DEVELOPERS, ANDROID. Improve Your Code with Lint. Disponível em: <https://developer.android.com/studio/write/lint.html>. Acesso em: 08 out. 2015.

ANDROID DEVELOPERS. Platform Architecture. Disponível em: <https://developer.android.com/guide/platform/index.html#art>. Acesso em: 10 set. 2016.

FANG, Z.; LIU, Q.; ZHANG, Y.; WANG, K.; WANG, Z. ; WU, Q. A static technique for detecting input validation vulnerabilities in android apps. **Science China Information Sciences**, v. 60, n. 5, p. 052111, 2017.

FERREIRA, RICARDO LUIS D. M.; SANTOS, A. F. P. D. C. R. Vulnerabilities classification for safe development on android. **Journal of Information Systems Engineering & Management**, v. 1:3, p. 187–190, 2016. Disponível em: <http://www.lectitojournals.com/Article/Detail/ZPK6H594>. Acesso em: 20 jun. de 2016.

JOY, R.; AJITH, A. Survey on android malware detection methods using static and dynamic analysis. **International Journal**, v. 5, n. 7, 2016. Disponível em: <https://goo.gl/0joZk8>. Acesso em: 02 set. 2016.

KIM, J.; YOON, Y.; YI, K.; SHIN, J. ; CENTER, S. Scandal: Static analyzer for detecting privacy leaks in android applications. **MoST**, v. 12, 2012. Disponível em: <http://www.mostconf.org/2012/papers/26.pdf>. Acesso em: 10 set. 2016.

LI, LI AND BISSYANDE, TEGAWENDÉ FRANÇOIS D ASSISE AND PAPADAKIS, MIKE AND RASTHOFER, SIEGFRIED AND BARTEL, ALEXANDRE AND OCTEAU, DAMIEN AND KLEIN, JACQUES AND LE TRAON, YVES. Static Analysis of Android Apps: A Systematic Literature Review. Disponível em: <http://orbilu.uni.lu/bitstream/10993/26879/1/tr_slr_article.pdf>. Acesso em: 18 jun. 2016.

LINT PROJECT, ANDROID STUDIO PROJECT SITE. Android Lin Tips. Disponível em: <http://tools.android.com/tips/lint>. Acesso em: 08 out. 2015.

MARTÍNEZ, J. I. E.; ROJAS, L. C. Q. Vulnerabilidad en dispositivos móviles con sistema operativo android.. **Cuaderno Activa**, v. 7, n. 7, p. 55–65, 2015.

MEIER, J.D. AND MACKMAN, ALEX AND WASTELL, BLAINE AND BANSODE, PRASHANT AND TAYLOR, JASON AND ARAUJO, RUDOLPH. How To: Perform a Security Code Review for Managed Code (.NET Framework 2.0). Disponível em: <https://msdn.microsoft.com/en-us/library/ff649315.aspx>. Acesso em: 15 out. 2016.

NADEEM, M.; WILLIAMS, B. J. ; ALLEN, E. B. High false positive detection of security vulnerabilities: a case study. In: PROCEEDINGS OF THE 50TH ANNUAL SOUTHEAST REGIONAL CONFERENCE, 1., 2012. **Anais**... [S.l.: s.n.], 2012, p. 359–360.

OWASP, OPEN WEB APPLICATION SECURITY PROJECT. Mobile Top Ten Contributions. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_Contributions>. Acesso em: 19 set. 2015.

OWASP, OPEN WEB APPLICATION SECURITY PROJECT. Top 10 2013/ProjectMethodology. Disponível em: <https://www.owasp.org/index.php/Top_10_2013/ProjectMethodology>. Acesso em: 11 set. 2015.

OWASP, OPEN WEB APPLICATION SECURITY PROJECT. Top Ten Mobile Risks - Mobile Security Project. Disponível em: <https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks>. Acesso em: 09 nov. 2014.

RAMOS, C.; FEITOSA, E. Bibliotecas para cache em android: uma análise na perspectiva de segurança. In: XV SIMPÓSIO BRASILEIRO DE SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS, 15., 2015. **Anais eletrônicos**... Porto Alegre: Sociedade Brasileira de Computação, 2015, p. 354–357. Disponível

em: <http://sbseg2015.univali.br/anais/AnaisSBSeg2015Completo.pdf>. Acesso em: 16.05.2016.

RAWLINSON, KRISTI. HP Research Reveals Nine out of 10 Mobile Applications Vulnerable to Attack. Disponível em: <http://www8.hp.com/us/en/hp-news/press-release.html?id=1528865#.V_L94PkrLIV>. Acesso em: 11 jul. 2016.

SAĞLAM, Ġ. A. **Measuring and Assesment of Well Known Bad Practices in Android Application Developments**. 2014. 83 f. Dissertação (Information Systems) – Middle East Technical University, Ankara, 2014. Disponível em: <http://etd.lib.metu.edu.tr/upload/12617980/index.pdf>. Acesso em: 03 fev. 2016.

SCARSELLA, ANTHONY AND REITH, RYAN AND SHIRER, MICHAEL. Worldwide Smartphone Market Will See the First Single-Digit Growth Year on Record, According to IDC. Disponível em: <http://www.idc.com/getdoc.jsp?containerId=prUS40664915>. Acesso em: 19 fev. 2016.

SCHALLER, R. R. Moore's law: past, present and future. **IEEE spectrum**, v. 34, n. 6, p. 52–59, 1997.

SOUZA, L. S. M. D. **Early Vulnerability Detection for Supporting Secure Programming**. 2015. 134 f. Dissertação (Pós-Graduação em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015. Acesso em: 12 out. de 2014.

ARXAN TECHNOLOGIES. State of Mobile App Security: Apps Under Attack. 2014. Disponível em: <https://www.arxan.com/wp-content/uploads/assets1/pdf/State_of_Mobile_App_Security_2014_final.pdf>. Acesso em: 21 jul. 2015.

WU, DAOYUAN AND LUO, XIAPU AND CHANG, ROCKY KC. A Sink-driven Approach to Detecting Exposed Component Vulnerabilities in Android Apps. Disponível em: <https://arxiv.org/abs/1405.6282>. Acesso em: 01 jul. de 2015.

8 APÊNDICES

APÊNDICE 1: RELAÇÃO DE APLICATIVOS IDENTIFICADOS

Package Name	Armazenamento Frágil	Entrada Inválida	OWASP
am.zoom.mbrowser	✓	✗	M2
am.zoom.mlauncher	✓	✗	M2
ch.rmy.android.statusbar_tacho	✓	✗	M2
com.almalence.opencam_plus	✓	✗	M2
com.andrew.apollo	✓	✗	M2
com.android.smspush	✗	✓	M7
com.app2go.sudokufree	✗	✓	M7
com.asksven.betterbatterystats	✗	✓	M7
com.asksven.betterwifionoff	✗	✓	M7
com.blogspot.marioboehmer.nfcprofile	✓	✗	M2
com.bonelazy.profileswitcher	✗	✓	M7
com.brapeba.roaminginfo	✓	✗	M2
com.brianco.colorclock	✓	✗	M2
com.bvcode.ncopter	✓	✗	M2
com.callrecorder.android	✓	✗	M2
com.code.android.vibevault	✗	✓	M7
com.concentricsky.android.khan	✓	✗	M2
com.cybrosys.palmcalc	✓	✗	M2
com.danielme.muspyforandroid	✓	✗	M2
com.dwdesign.tweetings	✓	✗	M2

Package Name	Armazenamento Frágil	Entrada Inválida	OWASP
com.dynamite.heaterrc	✓	✗	M2
com.easwareapps.g2l	✓	✗	M2
com.f.coin	✓	✗	M2
com.fairphone.updater	✓	✗	M2
com.forum.fiend.osp	✓	✗	M2
com.frankcalise.h2droid	✓	✗	M2
com.ghostsq.commander	✓	✗	M2
com.google.android.diskusage	✓	✗	M2
com.ihunda.android.binauralbeat	✓	✗	M2
com.isdp.trirose	✓	✗	M2
com.nanoconverter.zlab	✓	✗	M2
com.nolanlawson.keepscore	✗	✓	M7
com.palliser.nztides	✓	✗	M2
com.sticksports.stickcricket	✓	✗	M2
com.ushahidi.android.app	✓	✗	M2
com.volosyukivan	✓	✗	M2
com.yasfa.views	✓	✗	M2
de.cketti.dashclock.k9	✗	✓	M8
de.onyxbits.drudgery	✓	✗	M2
de.onyxbits.listmyapps	✓	✓	M2; M7
eu.veldsoft.colors.overflow	✓	✗	M2
fi.mikuz.boarder	✓	✗	M2

Package Name	Armazenamento Frágil	Entrada Inválida	OWASP
fr.xgouchet.texteditor	✓	✗	M2
it.sineo.android.noFrillsCPUClassic	✓	✗	M2
kr.hybdms.sidepanel	✓	✗	M2
me.jamesfrost.simpledo	✗	✓	M7
name.livitski.games.puzzle.android	✓	✗	M2
name.starnberger.guenther.android.cbw	✓	✗	M2
net.bmaron.openfixmap	✓	✗	M2
net.pherth.omnomagon	✓	✗	M2
net.screenfreeze.deskcon	✓	✗	M2
org.blockinger.game	✗	✓	M7
org.droidparts.battery_widget	✓	✗	M2
org.geometerplus.zlibrary.ui.android	✓	✗	M2
org.mrpdaemon.android.encdroid	✓	✗	M2
org.ncrmnt.nettts	✓	✗	M2
org.projectvoodoo.screentestpatterns	✓	✗	M2
org.pyload.android.client	✓	✗	M2
org.servalproject.maps	✓	✗	M2
org.sparkleshare.android	✓	✗	M2
org.systemcall.scores	✓	✗	M2
org.tvbrowser.tvbrowser	✓	✓	M2; M7
ro.weednet.contactssync	✓	✗	M2
sk.halmi.fbeditplus	✓	✗	M2
za.co.neilson.alarm	✗	✓	M7