UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

JORDY FERREIRA GOMES

UM GUIA PARA ANÁLISE DE SEGURANÇA DE APLICATIVOS NA PLATAFORMA ANDROID

QUIXADÁ

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

G614g Gomes, Jordy Ferreira.
Um guia para análise de segurança de aplicativos na plataforma android / Jordy Ferreira Gomes. – 2017.
51 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2017.
Orientação: Prof. Dr. Jefferson de Carvalho Silva.

1. Android (Programa de computador). 2. Projeto aberto de segurança em aplicações web. 3. Protocolo criptográfico. 4. Segurança. I. Título.

CDD 005

JORDY FERREIRA GOMES

UM GUIA PARA ANÁLISE DE SEGURANÇA DE APLICATIVOS NA PLATAFORMA ANDROID

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de informação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de informação.

Orientador: Prof. Dr. Jefferson de Carvalho Silva

QUIXADÁ

2017

JORDY FERREIRA GOMES

UM GUIA PARA ANÁLISE DE SEGURANÇA DE APLICATIVOS NA PLATAFORMA ANDROID

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de informação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de informação.

Aprovada em: _____/_____/_____

BANCA EXAMINADORA

Prof. Dr. Jefferson de Carvalho Silva (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Me. Wagner Guimarães Al-Alam
Universidade Federal do Ceará (UFC)

Prof. Dr. Wladimir Araujo Tavares
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Aos meus pais pelo carinho e apoio que me deram ao longo dessa jornada.

À minha avó pelo apoio em meses difíceis.

Aos meus amigos Leidylaura, Lucas Aguiar, Iná e Danilo pelo conforto que só amigos podem proporcionar

Ao grupo dos *Piruletas* e a mãe de cada um deles.

Aos meus colegas de trabalho da *Morphus* pelo conhecimento adquirido.

Ao time de CTF *Rage Against the Flag* pela experiência.

Aos professores e servidores que acreditaram em mim na universidade.

Ao meu orientador por tomar esse papel importante na minha formatura.

À Universidade Federal do Ceará pela educação que recebi.

RESUMO

Este trabalho propõe um guia para análise de vulnerabilidades em aplicações para o sistema operacional de *smartphones Android* que tem como resultado uma lista de vulnerabilidades encontradas nas aplicações alvo. O guia foi baseado na lista do *OWASP Mobile Top 10 2016*, que reúne as 10 principais vulnerabilidades em aplicativos móveis encontradas no ano de 2016. O método dividiu os componentes a serem analisados em três: uso de Criptografia; Mecanismos de persistência e comunicação; Código, configurações e plataforma. Cada componente teve suas possíveis vulnerabilidades listadas, nas quais tiveram notas de gravidade atribuídas a partir da calculadora do *Common Vulnerability Scoring System*(*CVSS*), junto de suas sugestões de soluções e como constatar se estas vulnerabilidades estão presentes. Para fácil assimilação, as vulnerabilidades foram divididas nas gravidades baixa, média, alta e crítica, correspondentes as notas obtidas no CVSS. No fim do trabalho, foi realizado uma auditoria de segurança em três aplicações *Android* de categorias distintas que mostraram a eficácia do guia no seu objetivo proposto de achar vulnerabilidades.

Palavras-chave: Android (Programa de computador). Projeto aberto de segurança em aplicações web. Protocolo criptográfico. Segurança.

ABSTRACT

This article proposes a guide for vulnerability analysis of applications from the *Android* operating system for smartphones that brings a list of the found vulnerabilities as result. The guide was based from *OWASP Mobile Top 10 2016*, which gathers the 10 main vulnerabilities in mobile applications that had appeared in 2016. The guide splited the components to be analised in three: cryptograph uses; persistance and communications mechanisms; code, settings and platform. Each component had it's possible vulnerabilities listed, in which had scores of severity attributed from the calculator from *Common Vulnerability Scoring System*(*CVSS*), together which it's solutions and how to find if these vulnerabilities are present. For easy assimilation, the vulnerabilities were dividied in the low, medium, high and critical severity in the way they corresponds to each grade form CVSS. At the end is made a auditory from three *Android* application from distict categories that showed how the guide was effective in it's proposed objective of finding vulnerabilities.

Keywords: Android (Computer Program). Open Web Application Security Project. Secure Socket Layer. Security.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A popularização dos computadores pela sociedade nos anos de 1980 a 1990 trouxe uma radical mudança na forma como as pessoas interagiam com o mundo e com si mesmas. Junto com a internet, esse novo paradigma marcou o próximo século com uma revolução sem precedentes, onde a informação está mais acessível a todos, com dispositivos cada vez mais intrínsecos ao ser humano moderno. Dos *mainframes* aos *personal computers*, *notebooks* e *smartphones*, a computação está a cada dia mais acessível, hoje cabendo na palma da mão.

Devido a essa popularização, abriu-se espaço para plataformas que sejam fáceis para desenvolvimento e concepção por programadores e utilização de usuários. Um dos exemplos é o sistema operacional *Android*. Para *smartphones*, o *Android* foi projetado com praticidade em mente, fazendo com que o desenvolvedor interaja por um conjunto de *API* previamente desenvolvidas pelo fabricantes, sem se preocupar com a implementação destas *API*. Como todos os sistemas operacionais, o *Android* permite o uso das capacidades do hardware por meio de uma camada de abstração e provem um ambiente limitado para as aplicações (BRAHLER, 2010).

Apesar dos sistemas operacionais para *smartphones* em geral abstraírem muito do processo de desenvolvimento e limitarem o impacto das aplicações, desenvolvedores podem não se atentar ao fato de que apenas o uso desses sistemas não garante a segurança da aplicação de maneira que tais aplicações se tornem imunes a erros de lógica e falhas de implementação segura. Ao fazer mau uso das *API* e recursos disponíveis, desenvolvedores podem criar potenciais vulnerabilidades, as quais podem ser exploradas por um agente malicioso e acabar prejudicando a confiabilidade e confidencialidade dos dados do usuário, a disponibilidade da aplicação e até afetando o próprio sistema operacional.

Nesse contexto, esse trabalho teve como objetivo a criação de um guia para análise de segurança em aplicações para o sistema operacional *Android* que ajuda a encontrar vulnerabilidades no código e na comunicação com o servidor da aplicação. O uso deste guia servirá como base para auditorias de segurança de aplicativos para *Android*, e o resultado do uso é uma lista de vulnerabilidades que pode ser revista por uma analista de riscos que decidirá qual o melhor curso para o tratamento dessas vulnerabilidades.

2 TRABALHOS RELACIONADOS

Nessa seção serão discutidos trabalhos relacionados destacando as semelhanças e diferenças com o projeto desenvolvido nesse documento.

2.1 Análise de aplicativos bancários

Em Cruz e Aranha (2015) é apresentada uma análise dos aplicativos bancários brasileiros na plataforma *Android*. O estudo consistiu em fazer uma avaliação do uso de criptografia da conexão entre os clientes e servidores e ataque à criptografia dos servidores a fim de obter uma prova de conceito para confirmar as vulnerabilidades encontradas e seu impacto.

O estudo consistiu em realizar ataques diretos às conexões servidor da aplicações, comprometendo a confidencialidade dos dados que passavam pelo protocolo SSL/TLS (*Secure Socket Layer* e *Transport Layer Security*, respectivamente) e efetuou ataques do tipo MITM (*Man in the Middle*) com o intuito de encontrar credenciais e informações financeiras vazadas.

O guia proposto nesse trabalho também analisa a criptografia dos aplicativos que estão passando pela auditoria, onde é verificada as versões do SSL/TLS utilizadas pelos servidores e se elas são vulneráveis a ataques conhecidos ao protocolo SSL/TLS, tais como o BEAST , Ataque a Cifra RC4, LUCKY 13, dentre outros (SARKAR; FITZGERALD, 2013).

Entretanto, enquanto o de Cruz e Aranha (2015) foca na análise e resultados obtidos nas aplicações escolhidas, este foca em criar um guia que sirva como guia para análise de qualquer aplicação móvel para *Android* por um auditor, de modo que este possa seguir passos objetivos na verificação de segurança do sistema alvo.

2.2 OWASP inspired mobile security

Os estudos de Acharya *et al.* (2015) apresentam um *checklist* com foco para análise de aplicativos móveis na área de saúde, mas não exclusivamente a esta, e duas análises de caso de uso do *checklist* a aplicativos de saúde na plataforma *Android*. O *checklist* foi baseado na lista *OWASP* de principais vulnerabilidades para aplicações móveis.

Este trabalho, assim como o de Acharya *et al.* (2015), focou em contribuir com um produto que auxilie na busca de vulnerabilidades em aplicações móveis, utilizando a lista do *OWASP Mobile Top 10* como inspiração.

Entretanto, o estudo citado se diferencia deste trabalho por entregar não apenas um

checklist, mas um guia de como o auditor deve procurar as vulnerabilidades e sugestões de como encontrá-las e resolvê-las de junto sugestões de ferramentas e técnicas.

3 OBJETIVOS

3.1 Objetivo Geral

O objetivo geral desse trabalho é a criação de um guia para análise de segurança em aplicações no sistema operacional *Android* que servirá como referência na auditória por analistas e desenvolvedores.

3.2 Objetivos específicos

Os objetivos específicos deste trabalho são:

- Selecionar os componentes que devem ser priorizados em uma auditoria de segurança para aplicações *Android*.
- Desenvolver um guia que consiga abranger a auditoria de segurança nos componentes escolhidos.
- Fazer uma verificação do guia a partir de três aplicações escolhidas para auditoria de forma a constatar se o guia é adequado ao seu propósito.

4 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos apresentados neste trabalho. Na seção 4.1 é apresentado o OWASP, sua missão e contribuição para a comunidade de segurança da informação. Na 4.2 é apresentado o sistema operacional *Android*, seu funcionamento, práticas e componentes. A seção 4.3 apresenta o protocolo SSL/TLS e suas falhas. Por último, a seção 4.4 trata do CVSS, um sistema para categorizar vulnerabilidades a fim de ajudar na análise e gerência de riscos.

4.1 OWASP

O *Open Web Application Security Project* - OWASP - é uma organização internacional sem fins lucrativos com a missão de tornar a segurança de *software* algo notório, de forma com que indivíduos e corporações estejam hábeis à tomada de decisões significativas sobre o tema a tempo de previnir maiores danos (OWASP, 2017).

Desde sua criação em 2004, a organização tem lançado listas com as riscos em aplicações web mais relevantes no período de lançamento. Tais listas são escritas por intermédio e voto de consultores e profissionais da área de segurança da informação, que compartilham suas experiências e elegem os riscos considerados de maior ameaça do período em qual a pesquisa está sendo realizada. A OWASP aconselha todos os líderes de empresas na área de TI e *software* a disseminarem a lista em suas equipes, de forma que haja uma auditoria em seus softwares e se mantenha uma cultura na escrita de software seguro (OWASP, 2017).

4.1.1 OWASP Top 10 Mobile 2016

Da mesma maneira que ocorre com aplicações *web*, a OWASP também lança listas com riscos para aplicações de dispositivos móveis. Com o crescimento do uso de *smartphones* nos últimos anos e a popularização de suas plataformas para escrita de *software* por desenvolvedores, as aplicações para dispositivos móveis também são suscetíveis a falhas de projeto e implementação que podem se tornar vulnerabilidades.

Em 2016, foi lançada pela OWASP uma lista com os 10 riscos mais frequentes em aplicações móveis (OWASP, 2016). Os 10 itens são:

- ***M1 - Improper Platform Usage*** - Essa categoria cobre mau uso de uma *feature* da plataforma ou falha em usar os controles de segurança da plataforma. Tais exemplos seriam os

Intents do *Android*, permissões, *TouchID*, etc.

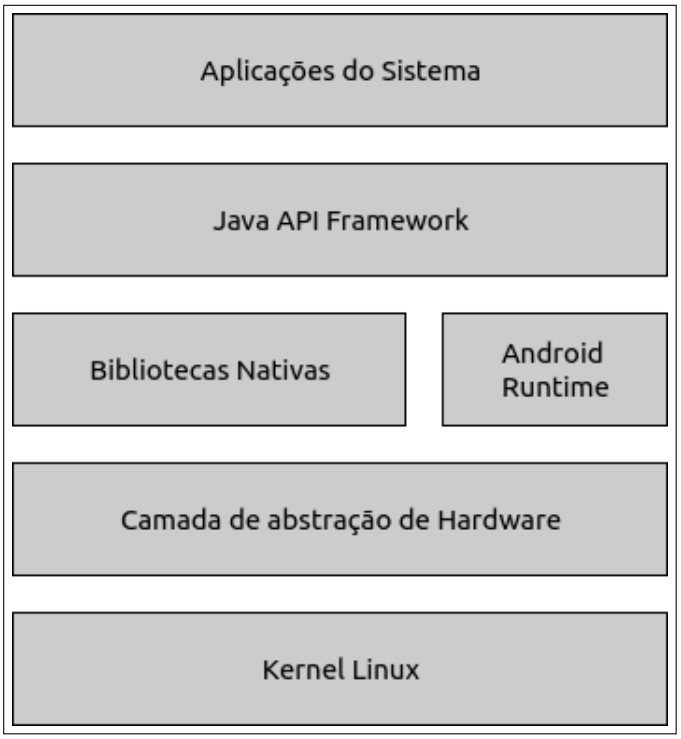
- ***M2 - Insecure Data Storage*** - Essa categoria inclui persistência insegura de dados e vazamento de informações sensíveis.
- ***M3 - Insecure Communications*** - *handshaking* do SSL malfeito, versões do SSL incorretas ou antigas, passagem de informações sensíveis em texto limpo, etc.
- ***M4 - Insecure Authentication*** - Fraqueza no manejo da sessão do usuário, falha na identificação do usuário na aplicação.
- ***M5 - Insufficient Cryptograph*** - Essa categoria cobre os casos de quando a criptografia foi utilizada de forma errada ou insuficiente. Tudo relacionado ao TLS e SSL deve ficar na categoria de *M6 - Insecure Communications*.
- ***M7 - Insecure Authorization*** - Essa categoria cobre o caso que a autenticação do aplicativo é vulnerável a ponto de deixar que um usuário malicioso se conecte como um usuário autorizado, podendo executar comandos com privilégios e comprometer os dados da aplicação.
- ***M7 - Client Code Quality*** - Essa categoria cobre o caso em que o código da aplicação cliente está vulnerável a certos *inputs* que possam comprometer o usuário que a utiliza. Um exemplo seria uma mensagem de texto cuidadosamente feita para quebrar a aplicação do usuário que recebesse uma mensagem de texto do atacante.
- ***M8 - Code Tampering*** - Essa categoria cobre as mudanças de código e do fluxo do programa que um atacante pode efetuar (*binary patching*, mudança dos recursos locais, etc).
- ***M9 - Reverse Engineering*** - Essa categoria cobre a análise do código, suas bibliteocas, algoritmos e outros ativos importantes. Por meio de softwares de decompilação e engenharia reversa, o atacante pode aprender sobre as funcionalidades da aplicação a fim de encontrar seus pontos vulneráveis.
- ***M10 - Extraneous Functionality*** - Essa categoria cobre funcionalidades que os desenvolvedores colocam nos seus aplicativos como funcionalidades para testes, tais como controles especiais, comentários com senhas, *backdoors*, dentre outros.

A lista *OWASP top 10* para riscos em aplicativos móveis será utilizada como referência para o guia a ser desenvolvido por esse trabalho.

4.2 Android

O *Android* é o sistema operacional móvel do Google e atualmente é líder mundial nesse segmento. É completamente livre e de código aberto (*open source*), o que representa uma grande vantagem competitiva para sua evolução, uma vez que diversas empresas e desenvolvedores do mundo podem contribuir para melhorar a plataforma (LECHETA, 2015). As aplicações Android são programadas com a linguagem Java e compiladas em *bytecode* para o formato *.dex* e executados em uma máquina virtual *Dalvik*. *Dalvik* é o nome da máquina virtual otimizada para executar código em dispositivos móveis.

Figura 1 – Arquitetura do Sistema operacional Android



Fonte – adaptado de GOOGLE (2017).

No topo de todo o sistema operacional, estão as aplicações de sistema, representadas na primeira camada. Nela estão as aplicações que o usuário interage, tais como aplicativos de agenda, ligação, jogos, redes sociais, dentre outros. É nessa camada que estão os aplicativos desenvolvidos por terceiros.

Logo abaixo da camada de aplicações, temos a camada da *Java API Framework*. Nela há o gerenciamento de recursos do sistema, gerenciamento de janelas, controle de eventos das aplicações, dentre outros.

Seguindo, temos as camadas de *Android Runtime* e as bibliotecas nativas. Na primeira temos as bibliotecas padrão do *Java* e do *Android*. As bibliteocas *Java* incluem classes que auxiliam na operação de manipulação de *strings*, conexões HTTP e HTTPS, manipulações de arquivo, etc. Essas classes são universais à todas as implementações de *Java*, e são comuns ao desenvolvedores desta linguagem, independente de plataforma; nas de *Android* temos as classes específicas do sistema operacional, tais como código para criação de *Activities*, *Views* e tratamento da entrada do usuário. Na segunda camada, estão as bibliotecas nativas à arquitetura do dispositivo nas quais fazem o trabalho das camadas anteriormente descritas, tais como desenhar uma figura na tela ou interpretar um arquivo de áudio para o usuário ouvi-lo pelos auto-falantes.

Na Camada de Abstração de *Hardware*, há um conjunto de classes *Java* que abstrai o uso dos recursos de hardware do sistema de modo que seja de fácil uso para os desenvolvedores. Toda essa camada consiste em uma API. Quando o *Java API Framework* faz uma chamada para acessar o hardware do dispositivo, o sistema *Android* carrega o módulo da biblioteca para este *hardware* (GOOGLE, 2017).

A última camada é o *kernel Linux*. Esta camada faz a ponte entre o *software* e o *hardware*. É por meio dela que as outras camadas se comunicam para obter acesso aos recursos de *hardware* do dispositivo. Nela também consta a responsabilidade de gerenciar a memória, os processos, *threads*, segurança dos arquivos e pastas, além de redes e *drivers* (LECHETA, 2015). Toda a segurança das as aplicações *Android* é baseada na segurança do *Linux*. Nele, cada aplicação é executada em um único processo e cada processo por sua vez contém uma *thread* dedicada. Para cada aplicação, é criado um usuário no sistema operacional, não havendo acesso entre os arquivos de usuários distintos, ou seja, aplicações não podem interagir diretamente com arquivos de outras aplicações (LECHETA, 2015).

As aplicações analisadas e o método criado neste trabalho serão voltados para o sistema operacional *Android*. As análises das aplicações serão na camada de aplicações do sistema e chamadas das *Android Runtime*.

4.2.1 Riscos de aplicações Android

O sistema operacional *Android* possui alguns componentes que podem ser utilizados de forma insegura pela camada de aplicação, acarretando assim em vulnerabilidades que colocam em riscos os usuários das aplicações. Serão destacadas alguns destes componentes a seguir.

4.2.1.1 Intents

Intents são mensagens das aplicações do sistema operacional *Android* que são recebidas pelo próprio sistema e passadas para outras aplicações, de forma com que possam ser tratadas. É a maneira do *Android* de fazer comunicaçao entre processos (IPC). Intents representam mensagens de algo que precisa ser realizado, tal como o ato de tirar uma foto, enviar um e-mail ou compartilhar um texto em uma rede social (LECHETA, 2015).

O objeto que recebe e trata as intents é um *Broadcast Receiver*. Como há vários tipos de *intent* que um *Boardcast Receiver* pode receber, o desenvolvedor pode configurar que tipo de mensagens o seu *Receiver* irá tratar. Essa configuração é tratada pelo sistema operacional como um filtro. Todo o trabalho do *Broadcast Receiver* é feito em segundo plano. Ele não gera interface gráfica nem interação direta com o usuário.

Como as *intents* levam conteúdo, tal conteúdo pode ser propositalmente criado para explorar a aplicação alvo, a partir de vulnerabilidades no código de seu *Broadcast Receiver* ou em componentes mais intrísecos da aplicação. Um atacante pode, por exemplo, enviar um texto em uma codificação não suportada para a aplicação, notando que esse não irá ser interpretado da forma esperada e acaba por gerar fluxo inesperado no código ou até mesmo uma falha no serviço (*Denial of Service*).

Dessa forma, teremos também uma análise do recebimento de *intents* pela aplicação a ser auditada.

4.2.1.2 Banco de dados

O *Android* tem integração com a *engine* de banco SQL *SQlite* que permite as aplicações de usuários utilizar banco de dados por meio da *API* do sistema (LECHETA, 2015). Há ainda, como ferramenta, o programa *sqlite3* para análise de banco de dados das aplicações, a partir do acesso de uma *shell* ao dispositivo. Com esse programa, é possível analisar o banco e efetuar consultas SQL a afim de extrair os dados ali inseridos. O acesso ao banco de dado só é permitido à aplicação que o criou. Qualquer outra aplicação não tem acesso para leitura ou escrita nesse banco.

Assim como em outros sistemas que utilizam banco de dados, aplicações do sistema operacional *Android* também são suscetíveis à injeções de SQL (*SQL injections*) em uma aplicação vulnerável (DRAKE *et al.*, 2014).

Vale ressaltar que os bancos não são encriptados e caso a haja comprometimento da regra de permissões do sistema aos arquivos da aplicação alvo, não há como garantir a confidencialidade dos dados ali persistidos.

4.2.1.3 Arquivos

O *Android* também possibilita trabalhar com criação e leitura de arquivos direto na memória externa do dispositivo (cartões SD). Tais arquivos podem ser erroneamente utilizados para guardar informações sensíveis, visto que arquivos criados na memória externa não dispõem da mesma regra de acesso que os arquivos criados na memória interna, logo podem ser acessados por qualquer aplicação do sistema.

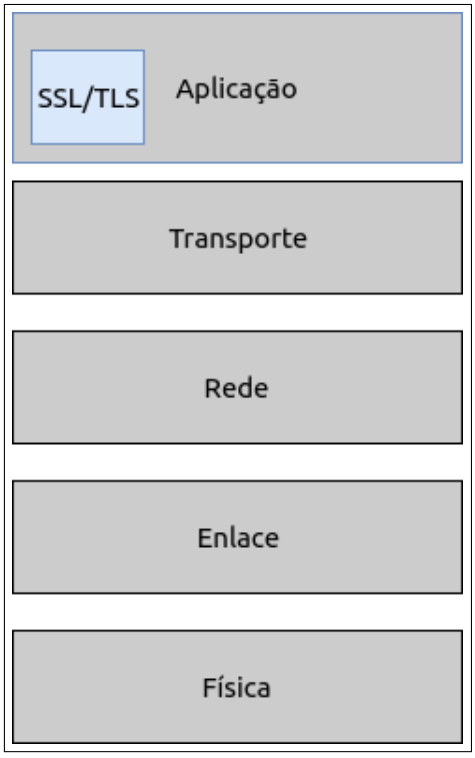
4.3 Protocolo SSL/TLS

O *Transport Layer Security* (TLS) e seu predecessor, *Secure Socket Layer* (SSL) , ambos referenciados como "SSL", são protocolos de segurança que provem comunicação segura na rede de computadores. Ou seja, permite a comunicação entre aplicações clientes/servidores de forma que estejam prevenidos de *eavesdropping* (espionagem, escuta, interceptação indevida), interferência e falsificação de mensagem (DIERKS; RESCORLA, 2008).

A história do SSL começou em 1994, quando a empresa *Netscape Communications*, criadora do navegador *web* de mesmo nome, reconheceu a necessidade de estabelecer um protocolo que permitisse a comunicação segura entre duas entidades pela internet, requisito para a consolidação do comércio eletrônico. Este protocolo foi o SSL 1.0, concebido com o objetivo de oferecer as propriedades clássicas da segurança de informação (CRUZ; ARANHA, 2015):

- **Autenticidade** - Define que a mensagem está sendo enviada por uma fonte legítima e confiável.
- **Confidencialidade** - Define que haverá sigilo e proteção da mensagem contra agentes não autorizados.
- **Disponibilidade** - Define que a informação deve estar disponível o máximo de tempo possível, com resistência à falhas e manutenção do serviço.
- **Integridade** - Define que a informação terá seu conteúdo integro e inalterado. Ou seja, não será alterada sem permissão.
- **Irretratabilidade** - Define que algo que foi rejeitado e não pode ser tratado novamente.

Figura 2 – O protocolo SSL/TLS na pilha de camadas TCP/IP.



Fonte – O próprio autor.

Está associado aos certificados de assinatura digitais.

Após varias mudanças, a *Netscape Communications* lançou a versão final do protocolo em 1996, a versão SSL 3.0 (NETSCAPE, 1996). Essa versão teria sido a base para criação do sucesso do SSL, o TLS 1.0, lançado em 1999, agora controlado por uma comunidade aberta - a *IETF - Internet Engineering Task Force* (CRUZ; ARANHA, 2015).

4.3.1 Descrição do protocolo

O SSL funciona dentro da camada de aplicação no modelo TCP/IP. Todos os dados de aplicações que utilizam esse protocolo devem passar primeiramente pela camada SSL antes de seguirem pela Camada de Transporte (TCP) (KUROSE; ROSS, 2013).

Dentro do SSL há quatro protocolos:

- **Handshake Protocol** - Negocia as informações necessárias para a sessão entre o cliente o servidor. Verificar qual versão do protocolo usar, qual algoritmo de criptografia utilizar, autenticação por troca de certificados, validação de certificados por meio de uma autoridade certificadora (CA) e troca de segredo para utilização de chave simétrica.
- **Cipher Change Protocol** - É utilizado para mudar as chaves de encriptação sendo uti-

lizadas pelo cliente e servidor. É normalmente utilizado como parte do protocolo de *Handshake* para mudar de encriptação assimétrica para simétrica. Esse protocolo se compõe de uma única mensagem enviada à entidade destinatária de que a remetente que mudar para um novo conjunto de chaves, nas quais são criadas a partir de informação trocada pelo protocolo de *Handshake*.

- **Alert Protocol** - Protocolo utilizado para indicar mudanças de estado ou erros entre as entidades envolvidas. Um exemplo seria quando a conexão é encerrada, acarretando em uma mensagem inválida a ser recebida pelo remetente.
- **Record Protocol** - Faz a verificação dos dados da aplicação utilizando a chave simétrica derivadas durante a etapa de *handshake*.

No guia que desenvolvido nesse estudo há sugestões de testes no protocolo SSL/TLS da conexão entre aplicaçao e o servidor a fim de achar vulnerabilidades que possam comprometer a confidencialidade de informações sensíveis do usuário.

4.3.2 Vulnerabilidades do SSL/TLS

Aqui serão descritos alguns ataques que exploram vulnerabilidades relevantes do SSL/TLS.

4.3.2.1 BEAST

O ataque BEAST (*Browser Exploit Against SSL/TLS*) consiste em se aproveitar da escolha não aleatória dos vetores de inicialização (*Initialization Vectors, IV*) do SSL/TLS quando se usa o modo de operação de cifra de blocos encadeados (*Cipher Block Chaining, CBC*). O ataque se baseia em um tipo de ataque criptografico de texto escolhido (*Choosen-plaintext attack*). Nesse tipo de ataque, o atacante tem acesso à mensagem criptograda de qualquer texto limpo de sua escolha . O atacante pode fazer uma tentativa de achar o texto limpo correto para uma dada mensagem cifrada, apenas enviando texto limpo ao oráculo de criptografia e comparando o resultado com a mensagem cifrada alvo (FERGUSON *et al.*, 2011).

Para se defender dos ataques de texto escolhido, as implementações do TLS utilizam dois mecanismos: um vetor de inicialização (IV) e o modo de operação de bloco encadeado (CBC). O IV é um vetor de texto aleatório que cifra a mensagem limpa com um criptografia *XOR*, processo feito antes da encriptação da mensagem - mesmo que a mesma mensagem seja cifrada duas vezes, os textos resultantes serão diferentes graças à aleatoriedade da chave IV. A

chave IV não é secreta, pois ela é enviada junto da mensagem em texto limpo. Seria pesado utilizar um novo IV para cada bloco cifrado em mensagens muito extensas (o padrão AES[1], por exemplo, utiliza em blocos de 16-bytes), então, nestes casos, o modo CBC utiliza o bloco cifrado anterior como IV para utilizar como chave na encriptação da próxima mensagem em texto limpo.

O uso de IVs e CBC não garantem que a mensagem cifrada é invulnerável à ataques de texto escolhido: um atacante pode ser capaz de prever o IV que será utilizado na encriptação da mensagem em seu controle e também saber o IV da mensagem cifrada que está tentando quebrar (FERGUSON *et al.*, 2011).

4.3.2.2 CRIME

O ataque CRIME (*Compression Ratio Info-leak Made Easy*) é um ataque de *side-channel*[2] que pode ser utilizado para descobrir tokens de sessão ou outras informações secretas na mensagem cifrada utilizando informação baseada no tamanho da compressão de requisições HTTP. A técnica explora sessões web protegidas por SSL/TLS quando esta utiliza esquemas de compressão de dados como o DEFLATE ou gzip, os quais são inclusos no protocolo e utilizados para reduzir o congestionamento na rede e tempo para carregar as páginas web (SARKAR; FITZGERALD, 2013).

O ataque funciona da seguinte maneira: a partir do conhecimento parcial de uma parte do texto de uma mensagem secreta, o atacante faz um ataque de força bruta para gerar o resto da mensagem por meio de um MITM(*Man In The Middle*), verificando se o seu resultado é correto a partir do tamanho da mensagem ao sofrer compressão. Um exemplo poderia ser um token de sessão enviado por HTTP escrito como *token=cdc123*. Com o conhecimento que a mensagem do token tem formato *token=**, o atacante realiza um ataque por força bruta de possíveis valores deste token (*token=a,token=b,token=c...*) injetados na conexão HTTP da vítima ao mesmo tempo que verifica o tamanho das requisições geradas pelo SSL/TLS que passaram por compressão de dados. Nesse exemplo específico, quando o pacote SSL/TLS conter o dado injetado *token=c*, o algoritmo de compressão notará a semelhança entre as duas strings *token=c* e *token=cdc123*, fará a compressão na string *token=cdc123* e gerará um pacote de tamanho menor do que os pacotes que continham um outro valor para *token=** tal que * fosse diferente de c. A partir daí, o atacante confere que fez a tentativa correta e continua tentando quebrar o resto da

[1] <https://pt.wikipedia.org/wiki/Advanced_Encryption_Standard>

[2] Um ataque que faz uso de informação da mensagem além do seu conteúdo ou mensagem cifrada, tal como tamanho da mensagem cifrada ou tempo de operação do algoritmo de cifragem (FERGUSON *et al.*, 2011)

mensagem por força bruta (*token=ca,token=cb...*) até conseguir gerar a mensagem esperada.

Como o uso de compressão produz essa vulnerabilidade no protocolo, a mitigação de menor impacto adotada por várias entidades que fazem uso do SSL/TLS foi da desativação total da compressão por SSL/TLS por ser a solução de menor intervenção manual (SARKAR; FITZGERALD, 2013).

4.3.2.3 TIME

O ataque TIME (*Time Info-leak Made Easy*) consiste em um outro ataque de texto escolhido nas respostas HTTP. O ataque é análogo ao CRIME, mas em vez de utilizar o tamanho da compressão da mensagem cifrada, o TIME analisa a diferença de tempo do recebimento das mensagens comprimidas.

Visto que esse ataque tem certas semelhanças com o CRIME, citaremos dois inconvenientes do ataque CRIME:

1. Como o uso de compressão por SSL/TLS foi desativado na maior partes das entidades que utilizam esse protocolo, o ataque tornou-se irrelevante.
2. Para obter sucesso, o atacante necessita efetuar um ataque MITM para poder medir os tamanhos dos pacotes SSL/TLS.

O TIME consegue resolver essas duas limitações. Baseando-se no fato da internet recomendar compressão da resposta do protocolo HTTP como boa prática para ter maior velocidade e diminuir o congestionamento de rede, o TIME mudou o alvo do seu ataque para a resposta HTTP vinda do servidor, em vez da solicitação do cliente.

Sendo assim, o TIME consiste em fazer um ataque de força bruta na conexão do cliente, da mesma maneira que o CRIME, porém dessa vez explorando a compressão do HTTP e verificando o tempo que a conexão de resposta levou para ser recebida pelo cliente. Se a conexão levou menos tempo, é porquê tinha menos dados; se tinha menos dados, é porquê houve compressão dos valores corretos (assim como no CRIME), validando a tentativa de força bruta. Sucessões de ataques validados pelo tempo de chegada dos pacotes geram o valor da texto secreto, quebrando a mensagem cifrada.

Diferentemente do CRIME, uma mitigação eficiente para tornar o TIME irrelevante não foi apresentada. Há sugestões que consistem tanto na adição de *delays* aleatórios durante o envio das respostas ao cliente que requisitou uma conexão quanto na eliminação de parâmetros desnecessários de requisições GET do protocolo HTTP.

4.3.2.4 BREACH

O ataque BREACH(*Brower Reconnaissance and Exfiltration via Adaptive Compression of Hypertext*) reviveu o CRIME de uma maneira diferente dos anteriormente citadas: utilizando a compressão do conteúdo da resposta HTTP, o BREACH consegue validar as tentativas na quebra de textos contidas no pacote de resposta de acordo com o tamanho.

Esse método necessita de um campo de entrada de dados pelo atacante que seja refletido na resposta do servidor alvo. Um exemplo poderia ser um campo em um formulário que espelhe o dado enviado pelo atacante. Dessa forma, o atacante pode, por exemplo,quebrar o valor de tokens CSRF para realizar um ataque em nome da vítima.

Um exemplo prático em alto nível seria da seguinte forma: imagine que há um token chamado `csrf=gcdefg123`. O atacante descobre que, ao enviar um campo do site ao formulário, esse mesmo campo é refletido na resposta do site. Logo, ele começa a tentar um ataque por força bruta no token `csrf`, enviando no campo o valor `csrf=a`, `csrf=b`, `csrf=c`, em diante. A tentativas erradas terão o mesmo tamanho, mas a correta (no caso, `csrf=g`) será um `byte` menor que as outras. Ao descobrir que essa é a tentativa correta, checando os tamanhos do pacote, o atacante prossegue com sua tentativa por força bruta. É agora testada a próxima letra do token, `csrf=ga`, `csrf=gb`, em diante.

Não há uma solução prática que torne o BREACH irrelevante. Por ser um ataque de *side-channel* que utiliza a compressão do protocolo HTTP, a solução eficaz seria eliminar essa compressão. Infelizmente tal solução traria outra gama de problemas não só aos servidores da rede, mas também aos usuários. Compressão do protocolo HTTP é uma das tecnologias que possibilitaram uma maior transferência de dados entre sistemas computacionais sem sobrecarregar a largura de banda. Desabilitá-la irá trazer uma considerável carga de trabalho aos clientes, nós e servidores da rede.

4.4 Common Vulnerability Scoring System

O *Common Vulnerability Scoring System* (CVSS), da organização *FIRST*, é um sistema de pontuação que tenta prover uma maneira de capturar as principais características de uma vulnerabilidade e produzir uma pontuação refletindo sua severidade. Essa pontuação resultante pode ser utilizada por qualquer entidade para definir uma representação qualitativa (tal como baixo, médio, alto e crítico) e preparar as organizações para uma melhor gestão de

vulnerabilidades (FIRST, 2017).

Este trabalho utilizou a calculadora CVSS para qualificar as vulnerabilidades encontradas a fim de categoriza-las nos níveis de gravidade baixa, média, alta e crítica, com o seguinte critério:

- *Nota de 0.1 a 4.0* - Baixo
- *Nota de 4.1 a 7.0* - Médio
- *Nota de 7.1 a 9.0* - Alto
- *Nota de 9.1 a 10.0* - Crítico

5 METODOLOGIA

Esta seção tem como objetivo expor a metodologia utilizada para construção desse trabalho.

5.1 Identificação dos componentes necessários para avaliação de segurança de aplicativos na plataforma *Android*

A primeira etapa do trabalho consistiu em fazer uma seleção de componentes importantes para a segurança das aplicações e métodos de análises de segurança em aplicações móveis utilizando a literatura. Foram considerados elementos específicos do sistema operacional *Android* e sua arquitetura e elementos da conexão entre a aplicação e o servidor.

Um exemplo seria a verificação do protocolo criptografia da conexão entre a aplicação e o servidor: qual versão do protocolo SSL/TLS o servidor utiliza, caso utilize alguma criptografia para transito de informações. Outro exemplo seria vazamento de informações sensíveis em arquivos de texto sem a devida proteção que o sistema operacional *Android* oferece.

Cada componente de avaliação escolhido foi detalhado. O detalhamento se deu na explicação da funcionalidade deste componente, como ele se aplica ao sistema *Android*, quais vulnerabilidades o componente pode gerar e como essa vulnerabilidade pode ser corrigida.

5.2 Modelagem

Os componentes que foram revisados e justificados foram compilados em uma lista de vulnerabilidades. A partir desta lista, foram propostos procedimentos para análise de aplicações móveis para o sistema operacional *Android*, que resultou em um guia para o auditor que realizará a análise de segurança. Cada item da lista teve uma pontuação a partir da calculadora *CVSS*, e a partir desta nota foi alocado à vulnerabilidade um rótulo de gravidade baixa, média, alta e crítica.

5.3 Estudo de caso

Foi realizada uma análise de vulnerabilidade de três aplicativos *Android* utilizando o guia. Essa análise verificou se o guia era adequado para o seu uso proposto. Como resultado da análise, foram ser encontradas vulnerabilidades nas aplicações e foi demonstrado sugestões para

corrigi-las.

5.4 Guia

Nesta seção é apresentado o Guia para análise e auditoria de segurança de aplicativos *Android* e o uso de criptografia. O guia faz uma análise de três componentes: uso de Criptografia; Mecanismos de Persistência e Comunicação; Qualidade de Código, Configurações e Plataforma. Cada componente tem uma seção correspondente e uma tabela que lista todas suas vulnerabilidades, o risco e a gravidade de cada vulnerabilidade.

Esse componentes foram escolhidos por terem uma maior proximidade e controle com o desenvolvedor, onde há a maior chance de conter vulnerabilidades advindas de código ou decisões de lógica de negócio.

Cada vulnerabilidade considerada nos componentes tem um nome, uma breve explicação, sua gravidade, seu código da calculadora CVSS, como constatar se a vulnerabilidade existe e uma sugestão de solução.

Todas as vulnerabilidades da listas são independentes entre si, com exceção da vulnerabilidade 5.4.1.1, que ao ser dada como positivo na aplicação torna desnecessário checar as vulnerabilidades dos itens 5.4.1.2, 5.4.1.3, 5.4.1.4, 5.4.1.5 e 5.4.1.6.

Para utilização deste guia o analista deve auditar a aplicação procurando por cada vulnerabilidade. Há uma sugestão na parte "como constatar"que sugere uma maneira de encontrar a vulnerabilidade, podendo ser diferente de acordo com a preferência do auditor.

O fim deste capítulo contém uma tabela com todas as vulnerabilidades listadas nas seções. Esta tabela pode ser usada como referência para rápida visualização pelo auditor.

5.4.1 Criptografia

A criptografia é o componente que protege os dados sensíveis do usuário. Por meio do protocolo SSL/TLS, a aplicação faz uma negociação de chave pública-privada e assim efetua uma um conexão segura por meio de dados criptografados por uma chave simétrica. A criptografia também pode ser utilizada para cifrar dados locais importantes, como um arquivo de informações sensíveis.

Essa seção equivale aos componentes M3 e M5 da *OWASP Mobile Top 10*.

Tabela 1 – Conexão entre servidor e aplicação

Item	Risco	Gravidade
Passa info. sens. sem SSL	Vazamento de info. do usuário	Crítica
Suporta SSL 3.0	Vuln. ao POODLE	Média
Suporta TLS 1.0	Vuln. ao BREACH	Média
Compressão SSL/TLS	Vuln. ao BEAST	Média
Suporte a RC4	Vuln. RC4	Média
Não verifica assin. do cert. do servidor	Interceptação de dados	Alta
Algo. de *hash* fracos	Vaz. de info. sensíveis	Alta
Chaves *hardcoded*	criptografia comprometida	Alta

Fonte – Autor.

Vulnerabilidades

As vulnerabilidades listadas para esse componente são:

5.4.1.1 Passa informações sensíveis sem SSL

A aplicação pode dispensar o uso de SSL/TLS e fazer autenticação por meio de texto limpo, como um *socket* TCP ou HTTP. Essa prática faz com que um atacante seja capaz de *farejar* a conexão com uma ferramenta e seja capaz de interceptar todo o texto que é passado no enlace, comprometendo a confidencialidade e a integridade destes dados.

- **Gravidade:** Crítica.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:H
- **Como constatar:** Utilizar um *sniffer* de rede para analisar os pacotes da aplicação e verificar se é utilizado ou não o protocolo SSL/TLS.
- **Solução:** Utilizar o protocolo SSL/TLS. Se for utilizar o protocolo HTTP, utilizar o HTTPS e garantir que o servidor possue um certificado válido assinado por uma autoridade certificadora (CA). O uso de pinagem de certificado aumenta as defesas da aplicação contra um ataque MITM.

5.4.1.2 A conexão suporta SSL 3.0

O uso do SSL3.0 remete a uma vulnerabilidade por suportar um protocolo já antigo que tem inúmeras falhas já conhecidas com provas de conceito, como o *POODLE*.

- **Gravidade:** Média.

- **Código *CVSS:*** CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:N/A:N
- **Como constatar:** Verificar o servidor da aplicação com o site <https://www.ssllabs.com> ou utilizar uma ferramenta como o *curl*, utilizando o comando *curl -vvI <http://exemplo.com>*.
- **Solução:** Desativar o suporte a SSL3.0 no servidor ou serviço da aplicação, deixando suporte apenas ao TLS1.1 e o TLS1.2.

5.4.1.3 A conexão suporta TLS 1.0

O uso do TLS1.0 remete a uma vulnerabilidade por suportar um protocolo já antigo que tem inúmeras falhas já conhecidas com provas de conceito, como o *BREACH*.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:N/A:N
- **Como constatar:** Verificar o site com o <https://www.ssllabs.com> ou utilizar uma ferramenta como o *curl*, utilizando o comando *curl -vvI <http://exemplo.com>*.
- **Solução:** Desativar o suporte a TLS1.0 no servidor ou serviço da aplicação, deixando suporte apenas ao TLS1.1 e o TLS1.2.

5.4.1.4 A conexão suporta compressão SSL/TLS

A compressão SSL/TLS é um vetor de ataque *side-channel* por permitir que um atacante possa fazer força bruta de *cookies* e outros valores que possam comprometer a sessão e informações sensíveis do usuário.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:N/A:N
- **Como constatar:** Verificar o servidor com o <https://www.ssllabs.com>.
- **Solução:** Desativar o suporte a compressão de SSL/TLS na aplicação.

5.4.1.5 A conexão tem suporte a RC4

RC4 é um *stream cipher* utilizado por 30% da internet em 2015. Foi bastante difundido por sua eficiência e simplicidade. Porém, em 2013 foi provado que era inseguro e era vulnerável a uma quebra por força bruta que tinha uma magnitude de dificuldade praticável por instituições que possuíssem infra-instruturas como *clusters* e supercomputadores (INITIATIVE, 2015).

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:N/A:N
- **Como constatar:** Verificar o servidor com o <https://www.ssllabs.com>.
- **Solução:** Desativar o suporte a compressão à cifra RC4.

5.4.1.6 Não verifica a assinatura do certificado

A assinatura do certificado digital é um componente eletrônico que infere a origem e integridade do documento. No caso de um *site*, o mesmo atesta que o certificado daquele *host* comprova ser quem o seu domínio diz que é. Porém, se uma aplicação aceitar conexões SSL sem verificar se a assinatura do certificado é válida, um atacante pode fazer um ataque MITM e poder interceptar toda a conexão entre os dois pontos. Caso a aplicação utilize de certificados auto-assinados, pode-se realizar uma pinagem de certificado para garantir a autenticidade do *host*.

- **Gravidade:** Alta.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:C/C:H/I:H/A:N
- **Como constatar:** Realizar um ataque *Man in the Middle* utilizando um certificado auto-assinado e observado se a aplicação continua funcionando normalmente, o que permitiria uma interceptação dos dados.
- **Solução:** Utilizar um certificado assinado ou pinagem de certificado na aplicação.

5.4.1.7 Algoritmos de hash fracos

A aplicação pode se utilizar de algoritmos de *hash* para funções diversas, como transformar uma senha ou gerar valores que sejam de uso da sua lógica. Caso sejam utilizados *hashs* fracos como MD5 ou SHA1, a aplicação torna-se vulnerável, pois estes já foram provados inseguros ao terem colisões descobertas(SECURITY, 2017).

- **Gravidade:** Alta.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N
- **Como constatar:** Verificar o código fonte da aplicação para encontrar o uso de *hashs* fracas em contextos onde deveriam proteger informações sensíveis.
- **Solução:** Utilizar algoritmos de *hash* fortes como SHA3 ou SHA256.

5.4.1.8 Chaves hardcoded

A aplicação pode conter chaves *hardcoded* (estáticas) no código, fazendo com que toda a criptografia entre a aplicação e o servidor seja comprometida caso o atacante consiga obter acesso a essas chaves por meio de uma análise estática do código utilizando decompilação e engenharia reversa.

- **Gravidade:** Alta.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:N/UI:R/S:C/C:H/I:H/A:N
- **Como constatar:** Fazer uma análise do código fonte ou de código descompilado procurando trechos de código que mostrem uma chave *hardcoded*.
- **Solução:** Não utilizar chaves simétricas como único método de criptografia de uma conexão ponta-a-ponta e sim utilizar o método de criptografia assimétrica por meio de implementações do SSL/TLS como o *OpenSSL*.

5.4.2 Qualidade de Código, configurações e plataforma

Outro componente a se levar em consideração é o código do aplicativo. Nele pode se esconder falhas que precisam de certa criatividade do atacante para serem exploradas, além de um trabalho minucioso de análise do código conseguido a partir de engenharia reversa.

Nesse ambiente, também temos problemas em lidar com as permissões do *Android*: um aplicativo que tem permissões de escrita no cartão SD(*Secure Digital*) pode ocasionar um *Denial of Service* caso essa funcionalidade esteja exposta por uma *Intent* que possa ser explorada pelo atacante de forma a encher o cartão com uma grande quantidade de dados. Pode haver o caso do aplicativo exigir uma permissão que não é necessária ao seu uso, e para diminuir a superfície de ataque, tal permissão deve ser removida.

Nisso, também há as configurações da aplicação: falta ofuscação de código ou aplicativos no modo *debug* são exemplos de má configurações que podem acabar indo para a loja de aplicativos e assim oferecer uma maior possibilidade ao atacante.

Essa seção relaciona aos componentes M1, M7 e M10 da *OWASP Mobile Top 10*.

Vulnerabilidades

As vulnerabilidades listadas para esse componente são:

Tabela 2 – Qualidade de Código, configurações e plataforma

Item	Risco	Gravidade
Mau tratamento de exceções	Indisponibilidade da aplicação	Média
Aplicativo não está em *Release*	Engenharia Reversa e exploração	Média
Contém código de *debugging*	Exploração da aplicação	Média
Uso indiscriminado de *Javascript* pelas *Webviews*	Ataques de XSS	Alta
Permissões não necessárias	Aumento do vetor de ataque	Média
Código não ofuscado	Engenharia Reversa	Média

Fonte – Autor.

5.4.2.1 Mau tratamento de exceções

A aplicação pode expor locais para entrada de dados do usuário nos quais não teve suas exceções corretamente tratadas e assim expondo o usuário final a ataques de indisponibilidade.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:C/C:L/I:L/A:L
- **Como constatar:** Colocar dados inválidos em formulários, utilizar o *Drozer* para realizar um processo de *fuzzing* nas *intents*.
- **Solução:** Fazer o correto tratamento de exceções no código, planejando de antemão os casos que comprometer o uso da aplicação.

5.4.2.2 Aplicativo não está em Release

Ao ser disponibilizada para o usuário, a aplicação pode ser exportada para um arquivo *APK*(*Android PacKage*). Esse tipo de arquivo é análogo ao *JAR* encontrado nas aplicações *Java*. O arquivo *APK* pode ser distribuído em modo *debug* ou *release*, nos quais o *release* contém algumas proteções como *minifyEnabled* do *Androguard*.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N
- **Como constatar:** Verificar com a ferramenta *jarsign*, *jarsigner -verify -verbose -certs myApp.apk*. Se a saída for "*CN=Android Debug*", então a aplicação está em modo *debug* e não *release*.
- **Solução:** Garantir que a aplicação esteja em modo *release* quando disponibilizada aos usuários finais.

5.4.2.3 Contém código de debugging

Desenvolvedores usam código específico na produção para efetuarem testes rápidos ou fazer *debbuging* em uma aplicação. Esse código pode ser para imprimir informações sensíveis em *logs* de forma a notificar o desenvolvedor sobre uma mudança de estado ou uma mudança no fluxo do código para *bypass* de métodos de autenticação.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:N/UI:R/S:C/C:H/I:N/A:N
- **Como constatar:** Usar ferramentas de monitoramento de log como o *logcat* e fazer uma revisão do código fonte. Caso o código fonte não esteja disponível, pode ser feita uma decompilação do código com *JD-gui* e *Apktool*.
- **Solução:** Efetuar uma revisão de código para não deixar que códigos para *debbuging* estejam na versão de produção do aplicativo.

5.4.2.4 Uso indiscriminado de Javascript pelas Webviews

O *Javascript* é uma linguagem de programação poderosa que teve suas origens em navegadores *web* e é amplamente utilizada no *front-end* e *back-end* de sites. Um tipo de *View* do *Android*, a *WebView*, permite o carregamento de páginas *web* pelo aplicativo.

Se a página carregada for vulnerável a ataques XSS(*Cross Site Scripting*), um atacante pode ter acesso a recursos do sistema com a diretiva "file:///"e até efetuar um ataque de *phishing* contra o usuário, mascarando o site que ele está navegando.

- **Gravidade:** Alta.
- **Código *CVSS:*** CVSS:3.0/AV:N/AC:L/PR:L/UI:R/S:C/C:H/I:N/A:H
- **Como constatar:** Verificar se o *WebView* possue *Javascript* habilitado e se seu uso é necessário para a funcionalidade da página.
- **Solução:** Caso seja desnecessário o uso de *Javascript*, não utilizar a função *setJavaScriptEnabled(true)* que o habilita. Evitar o uso do método *addJavaScriptInterface()* para código *Javascript* carregado por entidades externas, sendo recomendado apenas expor as interfaces do *Android* por *Javascript* apenas ao *Javascript* que vem junto do *APK* em sua aplicação.

5.4.2.5 Permissões não necessárias

Os aplicativos *Android* tem permissões do sistemas que são concebidas a um aplicativo com consentimento do usuário. Essas permissões deixam aplicativo acessar componentes como as câmeras do dispositivo ou poder escrever na memória do usuário, por exemplo. Permissões possibilitam com que o aplicativo possa ter o mínimo privilégio para realizar suas tarefas e que o usuário possa confiar em habilitá-las ou não durante o uso do aplicativo ou antes de instalá-lo. Um aplicativo que possui mais permissões do que o mínimo necessário ao seu uso pode comprometer a segurança do aparelho, pois um atacante pode mover lateralmente pelas permissões a afim de realizar um maior estrago no usuário.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:N/UI:R/S:U/C:H/I:H/A:N
- **Como constatar:** Verificar se a aplicação requisita mais permissões do que realmente utiliza em suas funcionalidades e verificar se os recursos destas permissões estão expostos diretamente por *Intents* que possam vir de aplicações externas.
- **Solução:** Minimizar o uso das permissões somente as funcionalidades utilizadas no aplicativo.

5.4.2.6 Código não ofuscado

Graças ao *ProGuard*, o *Android* tem várias proteções de código e aplicações de forma simples e rápida. Porém, nem sempre elas foram implementadas corretamente. Um desses casos é a ofuscação de código-fonte. Sem ela, o código *Java* da aplicação pode ser descompilado e facilmente lido por um atacante a fim de verificar como a aplicação funciona e poder tirar informações dela. Logo, é necessário que qualquer aplicação que esteja disponível para distribuição tenha seu código ofuscado, de forma a dificultar a engenharia reversa e diminuir o tamanho do seu binário para distribuição.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N
- **Como constatar:** Utilizar *decompilers* como o *Apktool*,*dex2jar* e *JD-GUI* a fim de fazer uma leitura no código descompilado e verificar sua semelhança ao código fonte.
- **Solução:** Utilizar a opção *minifyEnabled true* no arquivo *build.gradle*.

5.4.3 Mecanismos de Persistência e Comunicação

Persistência nas aplicações *Android* também é um componente importante para ser analisado na segurança do aplicativo. Uma aplicação que lida com dados sensíveis precisa garantir que esses dados estejam seguros e não sejam expostos para escrita ou leitura por outras aplicações. Também é preciso assegurar se a aplicação está usando corretamente mecanismos de criptografia nos arquivos.

Essa seção se relaciona com aos componentes M1, M2 e M10 da lista *OWASP Top 10 Mobile 2016*.

Tabela 3 – Mecanismos de Persistência e Comunicação

Item	Risco	Gravidade
Informação sensível na memória externa	Quebra de confidencialidade	Alta
Cache do teclado e clipboard com informações sensíveis	Quebra de confidencialidade	Alta
Exposição de dados sensíveis por IPC	Quebra de confidencialidade	Baixa
Exposição de dados sensíveis pela interface	Quebra de confidencialidade	Baixa
Escrita de dados sensíveis nos logs	Quebra de confidencialidade	Média
Uso inseguro da API de banco de dados	Quebra de conf. e integ dos dados	Média

Fonte – Autor.

Vulnerabilidades

As vulnerabilidades listadas para esse componente são:

5.4.3.1 Informação sensível na memória externa

O *Android* possibilita ao desenvolvedor gravar arquivos na memória externa como conveniência para que elas possam ser acessadas por outros aplicativos. O problema é quando essa função é erroneamente utilizada para gravar informações sensíveis: qualquer aplicação pode ler os arquivos da memória externa desde que tenha a permissão para tal.

- **Gravidade:** Alta.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N
- **Como constatar:** Verificar em um aparelho os conteúdos na memória externa e identificar se o aplicativo está guardando informações nele.
- **Solução:** Não utilizar a memória externa para guardar dados importantes da aplicação.

Caso houver necessidade de persistir informações sensíveis, utilizar a API do banco *SQLite* ou guardar na memória interna, ou mesmo no *SharedPreferences*.

5.4.3.2 Cache do teclado e clipboard com informações sensíveis

No preenchimento de formulários, o usuário precisa preenche-los com informações para serem processadas pelo aplicativo. Em alguns casos, essa informações podem ser sensíveis e o aplicativo precisa tomar cuidado para que elas não sejam salvas no cache do teclado ou mesmo na área de transferência (*clipboard*). Os dados salvos nessas áreas podem ser interceptados por outros aplicativos, gerando um ponto de falha.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:L/UI:R/S:U/C:H/I:N/A:N
- **Como constatar:** Verificar se o aplicativo coloca informação sensível no *clipboard* sem limpá-lo ou se senhas são gravadas no *cache* do teclado por não estarem sendo digitadas em campos específicos para senha.
- **Solução:** Caso o aplicativo coloque informação sensível no *clipboard*, eliminar o texto do clipboard depois de algum tempo previamente determinado. Caso precise colocar senha em um formulário, utilizar o atributo *android:inputType="textPassword"* na *View* de entrada,o que garante que não salvará o texto no dicionário do teclado.

5.4.3.3 Exposição de dados sensíveis por IPC

Dados sensíveis persistidos no disco rígido do aparelho podem estar expostos por meio de IPC(*Inter Process Comunication*), que no *Android* é feito com *Intents*. Caso o código que receberá a mensagem da *Intent* não for tratado corretamente, poderá haver exploração por parte de um atacante e acabar revelando dados sensíveis da aplicação.

- **Gravidade:** Baixa.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N
- **Como constatar:** Enviar *Intents* para *Activities* previamente estudadas da aplicação a fim de que ela retorne uma informação importante, como um token de sessão, credencial ou outra informação sensível.
- **Solução:** Evitar passagem de informações sensíveis por *Intents* para aplicações externas. Caso a lógica do negócio exija tal funcionalidade, tentar minimizar o impacto causado caso essa informação seja requisitada por um atacante.

5.4.3.4 Exposição de dados sensíveis pela interface

Exposição de dados sensíveis, como senhas e números de cartões de créditos, não devem ser retornados ao usuário nem ficarem ofuscados na interface. Um atacante que tomou o controle da aplicação tentar retirar os valores que contém nos objetos mostrados na interface, tornando não só a ofuscação de interface inútil, mas também perigosa.

- **Gravidade:** Baixa.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:H/PR:H/UI:R/S:U/C:H/I:N/A:N
- **Como constatar:** Utilizar a aplicação e verificar se alguma activity mostra esses dados objetivamente ao usuário. Caso só mostre dados parciais, como um número de cartão de crédito que contem parte do seu conteúdo em asteriscos ou uma senha em asteriscos que tem o mesmo tamanho da senha original do usuário, realizar um *debugging* para averigar se o valor contido dentro da entrada de texto é o dado 'limpo' ou ofuscado.
- **Solução:** Não colocar dados sensíveis como senhas do usuário em formulários ou mostrá-los na tela.

5.4.3.5 Escrita de dados sensíveis nos logs

Os *logs* de uma aplicação *Android* podem registrar informações sensíveis, como *token* de sessão do usuário ou até mesmo senhas. Logo, informações como estas não devem ser registradas em *logs*. No *Android* apenas o usuário *root* pode ler *logs* que não sejam feitos por ele mesmo.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:L/PR:H/UI:N/S:U/C:H/I:N/A:N
- **Como constatar:** Utilizar o programa *logcat* para visualizar toda a saída dos *logs* e procurar por informações sensíveis. O uso de filtros pode auxiliar na busca por informações de uma aplicação específica.
- **Solução:** Remover os trechos de código que imprimem informações sensíveis nos logs.

5.4.3.6 Uso inseguro da API de banco de dados

O *Android* possue uma *API* para escrita de dados utilizando a engine *Sqlite*. Essa *API* permite que o desenvolvedor utilize a engine em vez de desenvolver sua própria implementação de banco de dados. Logo, apesar do *SQLite* já ser um *software* maduro em relação a segurança,

seu mau uso pode ocasionar brechas. Um exemplo de mal uso é a utilização de *queries* não parametrizadas e de *queries* que utilizam concatenação direta(sem sanitização) com a entrada do usuário para operações que trabalhem com os dados do bancos. Tais operações poderiam quebrar a integridade do banco, fazendo com que um atacante possa abusar dos dados locais da aplicação para fins maliciosos.

- **Gravidade:** Média.
- **Código *CVSS:*** CVSS:3.0/AV:L/AC:L/PR:L/UI:R/S:C/C:L/I:L/A:L
- **Como constatar:** Fazer uma análise do código e verificar se as chamadas à *API* dos métodos de acesso ao banco *SQLite* contem *Strings* não parametrizadas concatenadas ao texto da *query*. Caso o código não esteja disponível, uma decompilação pode ser feita, ou um teste por meio de *fuzzing*.
- **Solução:** Utilizar parametrização de *query* e sanitizar a entrada do usuário para *queries* que necessariamente precisam ser concatenadas com entrada do usuário.

5.4.4 Considerações Gerais

Visto cada vulnerabilidade em seu detalhamento, a seguir é apresentado uma tabela compilando todas as vulnerabilidades de cada componente.

Para utilizar o método, o analista pode seguir os itens da tabela abaixo sequencialmente, checando as vulnerabilidades. No caso de positivo em alguma delas, é ideal tomar nota de qual parte do componente essa vulnerabilidade foi achada (qual parte do código, que interface, qual formulário preenchido), a maneira de como foi encontrada (qual ferramenta, como foi o processo para o analista encontrar aquela falha) e uma sugestão de como resolve-la. Um exemplo de resultado encontrado a partir do uso do método pode ser visto no capítulo 6.

Tabela 4 – Vulnerabilidades Gerais

Criptografia		
Item	Risco	Gravidade
Passa info. sens. sem SSL	Vazamento de info. do usuário	Crítica
Suporta SSL 3.0	Vuln. ao POODLE	Média
Suporta TLS 1.0	Vuln. ao BREACH	Média
Compressão SSL/TLS	Vuln. ao BEAST	Média
Suporte a RC4	Vuln. RC4	Média
Não verifica assin. do cert. do servidor	Interceptação de dados	Alta
Algo. de *hash* fracos	Vaz. de info. sensíveis	Alta
Chaves *hardcoded*	criptografia comprometida	Alta
Qualidade de Código, Configurações e Plataforma		
Item	Risco	Gravidade
Mau tratamento de exceções	Indisponibilidade da aplicação	Média
Aplicativo não está em *Release*	Engenharia Reversa e exploração	Média
Contém código de *debugging*	Exploração da aplicação	Média
Uso indiscriminado de *Javascript* pelas *Webviews*	Ataques de XSS	Alta
Permissões não necessárias	Aumento do vetor de ataque	Média
Código não ofuscado	Engenharia Reversa	Média
Mecanismos de Persistência e Comunicação		
Item	Risco	Gravidade
Informação sensível na memória externa	Quebra de confidencialidade	Alta
Cache do teclado e clipboard com informações sensíveis	Quebra de confidencialidade	Alta
Exposição de dados sensíveis por IPC	Quebra de confidencialidade	Baixa
Exposição de dados sensíveis pela interface	Quebra de confidencialidade	Baixa
Escrita de dados sensíveis nos logs	Quebra de confidencialidade	Média
Uso inseguro da API de banco de dados	Quebra de conf. e integ dos dados	Média

Fonte – Autor.

6 RESULTADOS

Este capítulo apresenta o estudo de caso com as vulnerabilidades encontradas das três aplicações escolhidas para auditória de vulnerabilidades utilizando o guia desenvolvido neste trabalho.

6.1 Táxi Brasil

A primeira aplicação escolhida foi a aplicação *Táxi Brasil*, achada na loja de aplicativos *Play*. Uma análise holística da aplicação decompilada fez perceber que a tecnologia utilizada para desenvolvimento foi o *framework* híbrido *Ionic*, no qual toda a lógica, visual e controle da aplicação é feito utilizando tecnologias *web* como HTML, *Javascript* e CSS. A decompilação foi feita com a ferramenta *Apktool*.

6.1.1 Código não ofuscado

Pela aplicação utilizar *Ionic*, o código que faz toda sua lógica a processamento está em *Javascript* e não em *Java*. Logo, ao procurar pelos arquivos, foi possível achar todo o código da compilação em *Javascript* não ofuscado. Assim, a aplicação apresentou a vulnerabilidade **Código não ofuscado**, com gravidade **média** (figura 3).

Solução: Utilizar um ofuscador de *Javascript* para esconder o código e dificultar que o atacante consiga ver o código e estudá-lo.

6.1.2 Passa informações sensíveis sem criptografia

No código é possível achar a *URL* base para todas as operações entre o cliente e servidor, onde foi possível descobrir que toda as informações passam por uma conexão HTTP sem SSL. Logo, um atacante conectado na mesma rede *Wifi* de uma vítima é capaz de 'farejar' os pacotes e descobrir as informações sensíveis do usuário. A aplicação apresenta a vulnerabilidade **1.1 - Passa informações sensíveis sem criptografia**, com gravidade **crítica** (figura 4).

Solução: Utilizar conexão HTTPS para passar informações sensíveis para evitar interceptação e perda de confidencialidade por um atacante.

Figura 3 – Exemplo de trecho de código não ofuscado

```
angular.module("starter").factory("loginService", function($http, config, PushTokenService) {

    var self = this;
    self.userpushtoken = new Object();
    self.db = new DataBase();

    //Explicando padrão observer para o Marcio
    self.callbacks = [];

    var addCallback = function(callback){
        self.callbacks.push(callback);
    };

    var faz = function(data){
        for(var i;i<self.callback.length;i++){
            self.callback[i](data);
        }
    };

    var _getUserPushToken = function(){
        return self.userpushtoken;
    };

    var _setUserPushToken = function(token){
```

Fonte – Autor.

Figura 4 – Arquivo de configuração do aplicativo

```
angular.module('starter').constant("config", {

    baseUrl: "http://[redacted]/",
    //baseUrl: "http:[redacted]",

    baseUrlMonet: "http://[redacted]",
    //baseUrlMonet: "http:[redacted]",

    token: "[redacted]",
    keygoogle: "[redacted]",
    tenant: "[redacted]",
    environment: "prod"
    //environment: "homo"
    //environment: "dev"
});
```

Fonte – Autor.

6.2 Transdroid

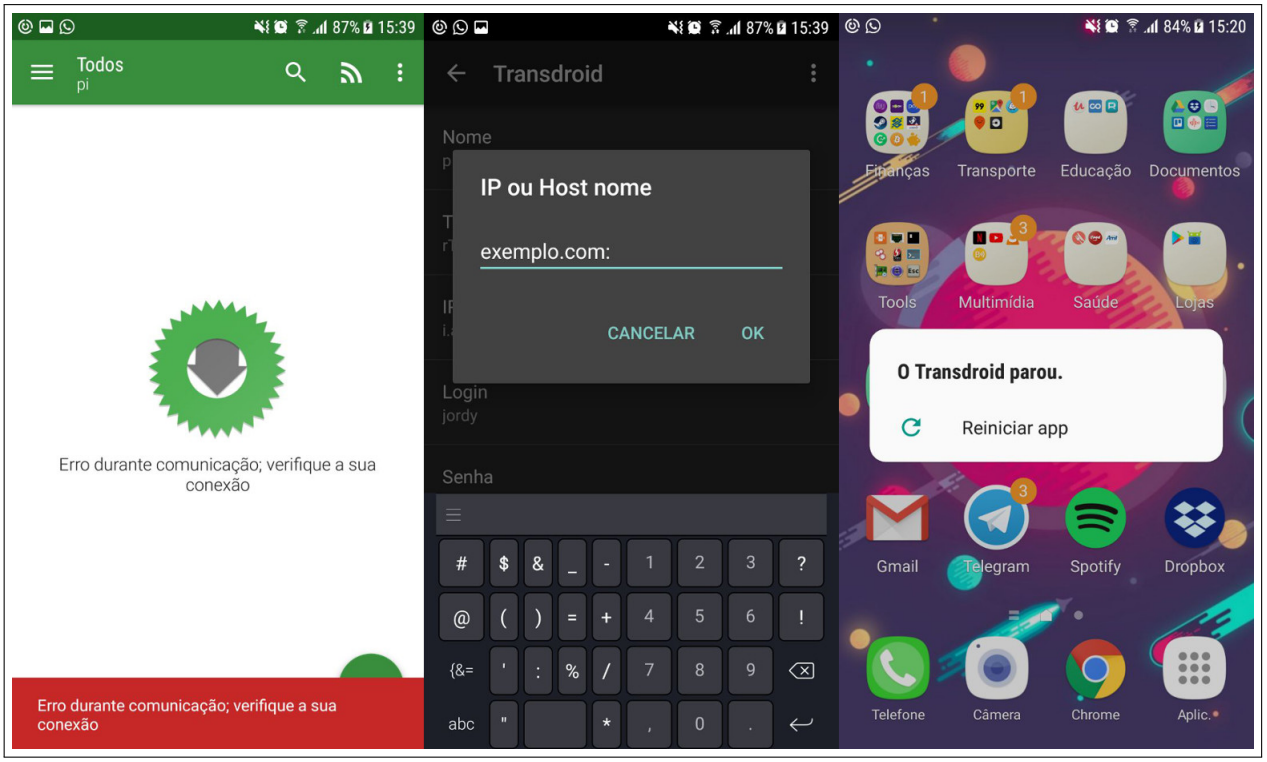
Transdroid é uma aplicação para gerenciamento de clientes *Torrent web*. Por ser de código livre, foi possível baixar o código fonte e fazer uma análise cuidadosa do programa.

6.2.1 Mau tratamento de exceções

Na entrada de dado para inserir o *host*, caso o usuário tente colocar o caractere ':', a configuração é salva no aplicativo e ao fazer a conexão com o servidor, a aplicação gera uma exceção que não é tratada e a fecha o aplicativo com um erro. Logo a aplicação apresenta a vulnerabilidade **Mau tratamento de exceções**, com gravidade **média** (figura).

Após o processo de *debugging* e inspeção da aplicação, foi possível descobrir o *bug*. Dentro da classe *XMLRPCClient* não tratava a exceção de caso o endereço da conexão estivesse inválido (como *exemplo.com:*). Como o endereço ficava salvo nas configurações da aplicação, era impossível abri-la novamente, causando um *Denial of Service* (figura 6).

Figura 5 – Telas do aplicativo *Transdroid* e erro pela exceção



Fonte – Autor.

Solução: após tratar exceção o código na classe *XMLRPCClient*, o aplicativo voltou a funcionar normalmente.

Figura 6 – Código do *Transdroid* com exceção tratada

Fonte – Autor.

6.3 Eaí

O aplicativo *Eaí* é uma aplicação para auxiliar os alunos da Universidade Federal do Ceará com assuntos relacionados ao Restaurante Universitário, tais como cardápio, votação sobre a satisfação com a comida, notícias, dentre outros.

6.3.1 Passa informações sensíveis sem criptografia

No código é possível achar a URL base para todas as operações entre o cliente e servidor, onde foi possível descobrir que toda as informações passam por uma conexão HTTP sem SSL. Logo, um atacante conectado na mesma rede *Wi-fi* de uma vítima é capaz de 'farejar' os pacotes e descobrir as informações sensíveis do usuário. A aplicação apresenta a vulnerabilidade **Passa informações sensíveis sem criptografia**, com gravidade **crítica**(figura 7).

Figura 7 – Código de configuração do aplicativo Eaí

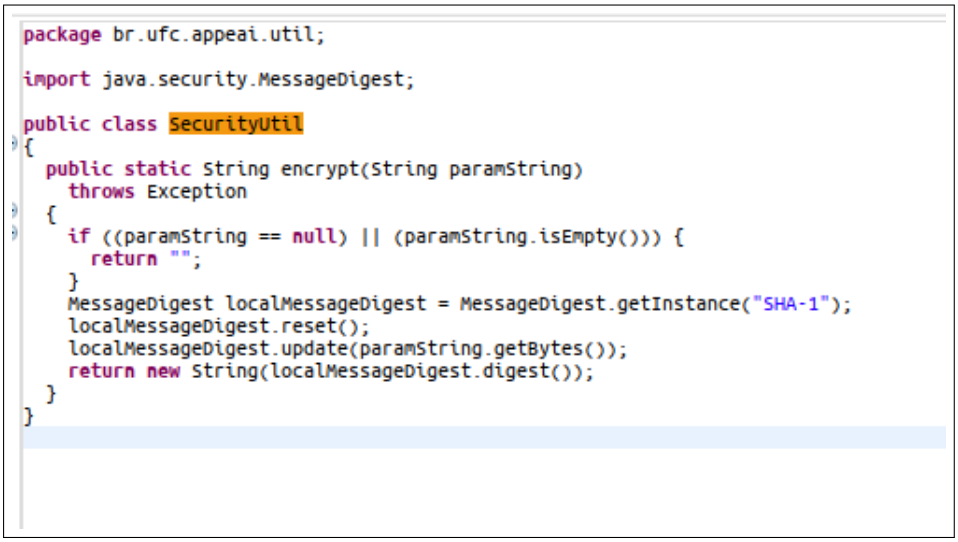
Fonte – Autor.

Solução: Utilizar conexão HTTPS para passar informações sensíveis para evitar interceptação e perda de confidencialidade por um atacante.

6.3.2 Algoritmos de hash fracos

A aplicação utiliza um algoritmo de *hash* SHA-1 para trabalhar com a senha do usuário. O problema com a utilização desse método é que o SHA-1 já foi provado inseguro e há algoritmos de *hash* melhores no ponto de vista de segurança. Logo o aplicativo apresenta **Algoritmos de hash fracos**, com a gravidade **alta**(figura 8).

Figura 8 – Código do aplicativo Eaí utilizando SHA-1



```
package br.ufc.appeai.util;

import java.security.MessageDigest;

public class SecurityUtil
{
  public static String encrypt(String paramString)
    throws Exception
  {
    if ((paramString == null) || (paramString.isEmpty())) {
      return "";
    }
    MessageDigest localMessageDigest = MessageDigest.getInstance("SHA-1");
    localMessageDigest.reset();
    localMessageDigest.update(paramString.getBytes());
    return new String(localMessageDigest.digest());
  }
}
```

Fonte – Autor.

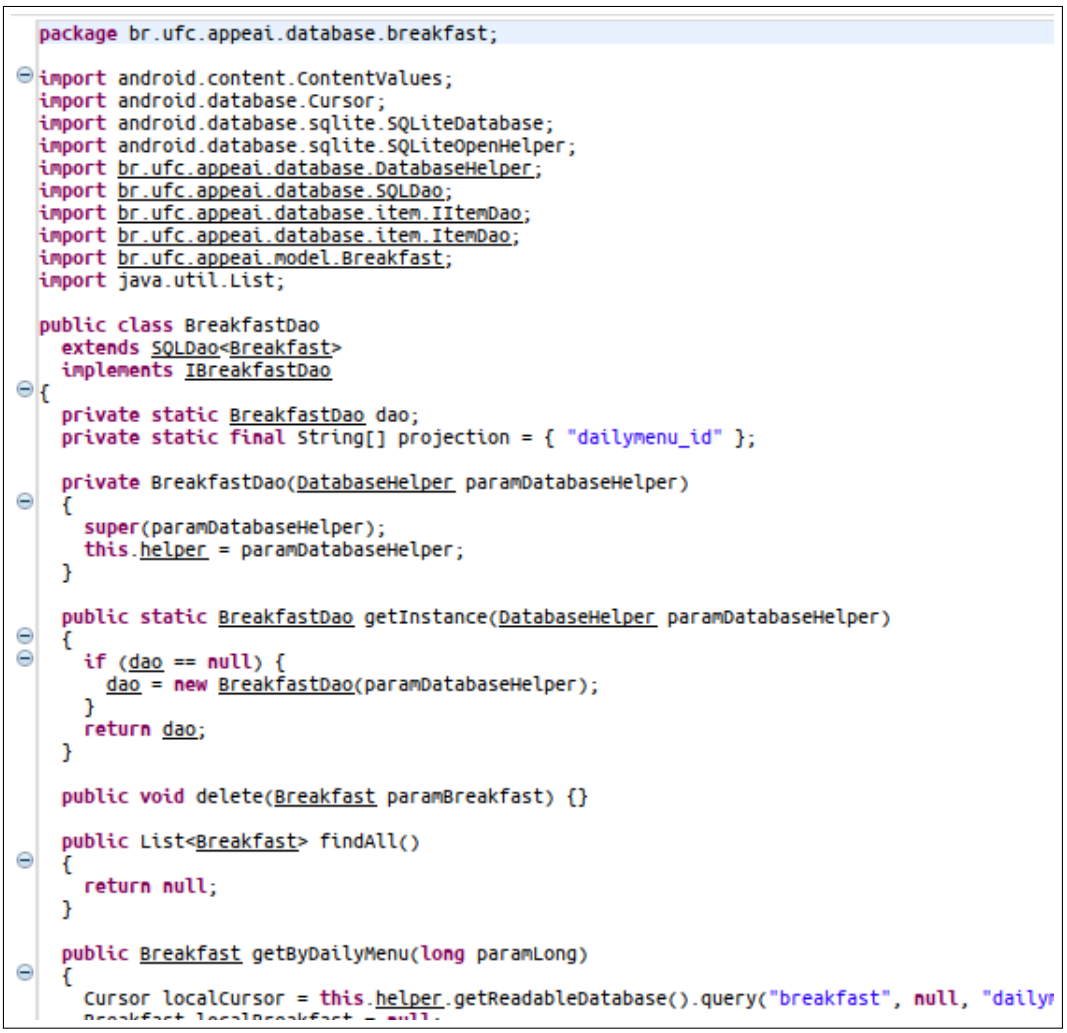
Solução: Utilizar algoritmos de *hash* fortes como SHA3 ou SHA256.

6.3.3 Código não ofuscado

O código da aplicação *Eaí* não foi ofuscado, sendo possível vê-lo tal como o código fonte. Com as ferramentas *dex2jar* e *JD-gui*, foi possível extrair esse código a partir da APK, onde *Strings* se manteram intactas no binário. Assim foi possível visualiza-lo e estudá-lo sem muitas dificuldades. Logo, o aplicativo apresenta a vulnerabilidade **Código não ofuscado**, com gravidade **média**(figura 9).

Solução: ativar a *flag minifyEnabled* no arquivo *build.gradle*, que fará com que o código seja ofuscado, dificultando a engenharia reversa por um atacante.

Figura 9 – Código da aplicação Eaí



```
package br.ufc.appeai.database.breakfast;

import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import br.ufc.appeai.database.DatabaseHelper;
import br.ufc.appeai.database.SQLDao;
import br.ufc.appeai.database.item.IItemDao;
import br.ufc.appeai.database.item.ItemDao;
import br.ufc.appeai.model.Breakfast;
import java.util.List;

public class BreakfastDao
  extends SQLDao<Breakfast>
  implements IBreakfastDao
{
  private static BreakfastDao dao;
  private static final String[] projection = { "dailymenu_id" };

  private BreakfastDao(DatabaseHelper paramDatabaseHelper)
  {
    super(paramDatabaseHelper);
    this.helper = paramDatabaseHelper;
  }

  public static BreakfastDao getInstance(DatabaseHelper paramDatabaseHelper)
  {
    if (dao == null) {
      dao = new BreakfastDao(paramDatabaseHelper);
    }
    return dao;
  }

  public void delete(Breakfast paramBreakfast) {}

  public List<Breakfast> findAll()
  {
    return null;
  }

  public Breakfast getByDailyMenu(long paramLong)
  {
    Cursor localCursor = this.helper.getReadableDatabase().query("breakfast", null, "daily
```

Fonte – Autor.

6.4 Considerações Gerais

Graças ao guia desenvolvido neste trabalho, foi possível achar todas as vulnerabilidades listadas em aplicações reais. Isso demonstra como a segurança de aplicativos *Android* é um assunto relevante e que merece estudo e contribuições da comunidade acadêmica.

Com esses resultados se prova que o guia é eficaz no seu objetivo e que contribui ao tema de segurança da informação e de aplicativos móveis.

7 CONCLUSÃO

Este trabalho apresenta um guia que se propõe identificar e categorizar vulnerabilidades em aplicativos *Android*, de forma que possa ser útil para analistas de segurança como referência onde for é necessário realizar uma auditoria de um aplicativo *Android*, com ou sem código fonte. O resultado do uso do guia é uma lista de vulnerabilidades encontradas.

O guia proposto foi baseado na lista *OWASP top 10 mobile 2016* e trás uma abordagem prática para analise de aplicativos *Android* em três componentes: *Criptografia*, *Mecanismos de Persistência e Comunicação* e *Qualidade de Código, Configurações e Plataforma*. Cada um desses componentes teve vulnerabilidades discriminadas com pontuações que definiam o grau de gravidade dessas vulnerabilidades a partir da pontuação na calculadora CVSS. Há também sugestões de como constatar e solucionar para cada uma das vulnerabilidades descritas.

A partir do guia proposto, foram feitas análises de aplicativos e as vulnerabilidades encontradas foram enquadradas na lista gerada por esse trabalho e registradas neste trabalho, junto de sugestões de solução para os problemas encontrados.

No geral, o guia se mostrou eficaz para seu objetivo por conseguir enumerar vulnerabilidades e espera-se que seja uma contribuição não só para a comunidade acadêmica mas para a comunidade de segurança de informação em geral.

7.1 Trabalhos Futuros

Como possíveis trabalhos futuros, pode-se sugerir os seguinte temas:

- A implementação de um guia que além de testar todas as vulnerabilidades aqui apresentadas na aplicação cliente, também ateste vulnerabilidades na aplicação servidor. O estudo feito neste trabalho encontrou uma vulnerabilidade no servidor de uma das aplicações, onde um usuário poderia requisitar informações sensíveis de todos os outros, tais como senhas, *e-mail*, telefone, códigos de cartões de crédito, dentre outras (figura 10). Como a vulnerabilidade estava fora do escopo do estudo, optou-se por não registra-lá no trabalho, o que deixa a oportunidade para um novo estudo que inclua este vetor em seu escopo.
- Um guia para encontrar vulnerabilidades exclusivas de aplicações de *frameworks* híbridos, nos quais geralmente são feitos com tecnologias *web* e apresentam um novo vetor para vulnerabilidades vindas dessas tecnologias. Cada vez mais esses *frameworks* tem se popularizado por sua fácil portabilidade entre plataformas distintas e facilidade de programação

quando comparado com as tecnologias que utilizam código nativo.

- Um guia para encontrar vulnerabilidades que foque ou inclua aplicações do sistema operacional *iOS*, que simboliza a segunda maior parcela entre os *smartphones* no mercado. Assim como acontece com as aplicações *Android*, esse sistema também utiliza *APIs* complexas, que exigem que o desenvolvedor tome cuidado com a segurança e não ofereça perigo aos seus usuários.

Figura 10 – Dados sensíveis de usuários

[{"id":1,"syncOperation":"NONE","numero":"**** **** ****
","bandeira":"MasterCard","token":"
","idCustomer":"
","cvc":
,"at
ivo":true,"cliente":{"id":1,"syncOperation":"NONE","nome":"
","sobrenome":"
","email":"lean
","username":null,"senha":"
","telefone":"(85)
","tipo":"APP","inativo":false,"usuarioCadastro":null,"dataCadastro":null,"matricula":null,"
limiteMensal":null,"cnpjCpf":null,"cnpj":null,"cpf":null,"autorizaEticket":null,"emailFacebook":"
","idFacebook":"
","observacao":null,"foto":null,"idCustomerCarta
o":"
","enderecosFavoritos":[],"telefonesFavoritos":[],"motoristaBloqueados":[],"empresa":null,
"centroCusto":null,"parceiro":null}}]
[{"id":3,"syncOperation":"NONE","numero":"**** **** ****
","bandeira":"MasterCard","token":"
","idCustomer":"
","cvc":
,"at
ivo":true,"cliente":{"id":4,"syncOperation":"NONE","nome":"
","sobrenome":"
","email":"
","username":null,"senha":"
","telefone":"(85)
","tipo":"APP","inativo":false,"usuarioCadastro":null,"dataCadastro":null,"matricula":null,"
":null,"cnpjCpf":null,"cnpj":null,"cpf":null,"autorizaEticket":null,"emailFacebook":"
","idFacebook":"
","observacao":null,"foto":null,"idCustomerCartao":"
","enderecosFavoritos":[],"telefonesFavoritos":[],"motoristaBloqueados":[],"empresa":null,"cent
roCusto":null,"parceiro":null}}]
[{"id":4,"syncOperation":"NONE","numero":"**** **** ****
","bandeira":"Visa","token":"
","idCustomer":"
","cvc":
,"ativo":t
rue,"cliente":{"id":
,"syncOperation":"NONE","nome":"
","sobrenome":"
","email":"
","username":null,"senha":"
","telefone":"
","tipo":"APP","inativo":false,"usuarioCadastro":null,"dataCadastro":null,"matricula":null,"
limiteMensal":null,"cnpjCpf":null,"cnpj":null,"cpf":null,"autorizaEticket":null,"emailFacebook":"
","idFacebook":"
","observacao":null,"foto":null,"idCustomerCartao":"
","enderecosFavoritos":[],"telefonesFavoritos":[],"motoristaBloqueados":[],"empresa":null,"centroCu
sto":null,"parceiro":null}}]

Fonte – Printscreen dos dados conseguidos por meio da exploração da falha do servidor na aplicação.

REFERÊNCIAS

ACHARYA, S.; EHRENREICH, B.; MARCINIAK, J. Owasp inspired mobile security. In: **2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)**. [S.l.: s.n.], 2015. p. 782–784.

BRAHLER, S. Analysis of the android architecture. **Karlsruhe institute for technology**, v. 7, p. 8, 2010.

CRUZ, R. J.; ARANHA, D. F. Análise de segurança em aplicativos bancários na plataforma android. In: **Workshop de Trabalhos de Iniciação Científica e de Graduação (WTICG), 2015, Florianópolis. XIV Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSEG)**. [S.l.: s.n.], 2015. p. 377–387.

DIERKS, T.; RESCORLA, E. **RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2.** 2008. Disponível em: <https://tools.ietf.org/html/rfc5246>. Acesso em: 21 maio 2017.

DRAKE, J. J.; LANIER, Z.; MULLINER, C.; FORA, P. O.; RIDLEY, S. A.; WICHERSKI, G. **Android hacker's handbook**. [S.l.]: John Wiley & Sons, 2014.

FERGUSON, N.; SCHNEIER, B.; KOHNO, T. **Cryptography engineering: design principles and practical applications**. [S.l.]: John Wiley & Sons, 2011.

FIRST. **Common Vulnerability Scoring System**. 2017. Disponível em: <https://www.first.org/cvss/>. Acesso em: 08 dezembro 2017.

GOOGLE. **Arquitetura da plataforma.** 2017. Disponível em: <https://developer.android.com/guide/platform/index.html>. Acesso em: 17 maio 2017.

INITIATIVE, H. I. Attacking ssl when using rc4. 2015.

KUROSE, J. F.; ROSS, K. W. **Computer networking: a top-down approach**. [S.l.]: Addison-Wesley Reading, 2013. v. 6.

LECHETA, R. R. **Google Android-4ª Edição: Aprenda a criar aplicações para dispositivos móveis com o Android SDK**. [S.l.]: Novatec Editora, 2015.

NETSCAPE. **RFC 6101 - The Secure Sockets Layer (SSL) Protocol Version 3.0.** 1996. Disponível em: <https://tools.ietf.org/html/rfc6101>. Acesso em: 21 maio 2017.

OWASP. **Mobile Top 10 2016**. 2016. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10>. Acesso em: 21 maio 2017.

OWASP. **About The Open Web Application Security Project - OWASP**. 2017. Disponível em: <https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project>. Acesso em: 21 maio 2017.

SARKAR, P. G.; FITZGERALD, S. Attacks on ssl a comprehensive study of beast, crime, time, breach, lucky 13 & rc4 biases. **iSEC Partners**, 2013.

SECURITY, G. **Announcing the first SHA1 collision**. 2017. Disponível em: <https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html>. Acesso em: 04 dezembro 2017.