UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO OLIVEIRA SILVA

TESTES DE SEGURANÇA EM APLICAÇÕES *ANDROID* BASEADOS NA METODOLOGIA OWASP

FORTALEZA-CE
2022

LEONARDO OLIVEIRA SILVA

TESTES DE SEGURANÇA EM APLICAÇÕES *ANDROID* BASEADOS NA METODOLOGIA OWASP

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Emanuel Bezerra Rodrigues.

FORTALEZA

2022

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

S581t Silva, Leonardo Oliveira.
Testes de segurança em aplicações Android baseados na metodologia OWASP / Leonardo Oliveira Silva. – 2022.
52 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Computação, Fortaleza, 2022.
Orientação: Prof. Dr. Emanuel Bezerra Rodrigues.

1. Testes de segurança. 2. Android. 3. Vulnerabilidades. I. Título.

CDD 005

LEONARDO OLIVEIRA SILVA

TESTES DE SEGURANÇA EM APLICAÇÕES *ANDROID* BASEADOS NA METODOLOGIA OWASP

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Ciência da Computação.

Aprovada em: xx/xx/xxxx.

BANCA EXAMINADORA

_____
Prof. Dr. Emanuel Bezerra Rodrigues (Orientador)
Universidade Federal do Ceará (UFC)

_____
Dr. Ismayle de Sousa Santos
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Rodrigo Carvalho Souza Costa
Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

AGRADECIMENTOS

Primeiramente, agradeço a Deus por me sustentar até aqui e por permitir a minha caminhada ao longo dos cinco anos de graduação.

Aos meus pais e avós, que sem eles nada disso seria possível, sou eternamente grato por todo o esforço que fizeram pela minha educação, por sempre me permitirem terem o melhor. Além do apoio ao longo da minha formação e por me incentivarem a sempre estudar e trilhar meu próprio caminho.

Ao Professor Emanuel Bezerra Rodrigues, pela excelente orientação, suas sugestões, correções e apoio durante a escrita deste trabalho.

Aos Professores participantes da banca examinadora Ismayle de Sousa Santos e Rodrigo Carvalho Souza Costa pelo tempo, pelas valiosas colaborações e sugestões.

Aos meus amigos e colegas que fiz na universidade, compartilhando conhecimentos e experiências que tornaram minha vida acadêmica mais agradável.

A Universidade Federal do Ceará, por toda estrutura e apoio necessário para o meu aprendizado.

“O sucesso nasce do querer, da determinação e persistência em se chegar a um objetivo. Mesmo não atingindo o alvo, quem busca e vence obstáculos, no mínimo fará coisas admiráveis.”

(José de Alencar)

RESUMO

O sistema operacional *Android*, por ser de código aberto e por ser possível de ser executado em diversos tipos de dispositivos, como *smartphones* e *tablets*, faz com que seja adquirido e utilizado no mundo inteiro, junto com as aplicações que podem ser executadas nessa plataforma. Como qualquer outro software, elas podem apresentar vulnerabilidades comprometedoras, como as elencadas pelo OWASP *Mobile Top Ten*, e, por esse motivo, se faz necessária a aplicação de uma metodologia de testes de segurança, principalmente abordando o contexto estático e dinâmico, para averiguar a existência dessas vulnerabilidades. Para isso, a utilização de um guia de requisitos de segurança e de procedimentos para realização desses testes se faz necessário para averiguar as 10 vulnerabilidades mais perigosas, por meio de ferramentas indicadas pelo guia de procedimentos de testes, que envolvem ferramentas de análise estática de código fonte, análise de comunicação de rede, e ferramentas para análise da aplicação em execução. As 5 aplicações públicas, que foram testadas em um emulador *Android*, continham o conjunto das 10 vulnerabilidades e, com a análise de cada vulnerabilidade, foi possível notar que boa parte delas eram causadas por más práticas de programação.

Palavras-chave: testes de segurança; *Android*; vulnerabilidades.

ABSTRACT

The Android operating system, being open source and being able to be run on different types of devices, such as smartphones and tablets, makes it acquired and used worldwide, along with the applications that can be run on this platform. Like any other software, they can have compromising vulnerabilities, such as those listed by the OWASP Mobile Top Ten, and for this reason, it is necessary to apply a security testing methodology, mainly addressing the static and dynamic context. For this, the use of a guide of security requirements and procedures for carrying out these tests is necessary to investigate the 10 most dangerous vulnerabilities, through tools indicated by the test procedures guide, which involve static source code analysis tools, network communication analysis, and tools for analyzing the running application. The 5 public applications, which were tested on an Android emulator, contained the set of 10 vulnerabilities and, with the analysis of each vulnerability, it was possible to notice that most of them were caused by bad programming practices.

Keywords: security tests; Android; vulnerabilities.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CWE	*Common Weakness Enumeration*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IPC	*Inter-Process Communication*
MAS	*Mobile Application Security*
MASTG	*Mobile Application Security Testing Guide*
MASVS	*Mobile Application Security Verification Standard*
OWASP	*Open Web Application Security Project*
Pentest	*Penetration Test*
SDLC	*Software Development Life Cycle*
SSDLC	*Secure Software Development Life Cycle*
TLS	*Transport Layer Security*

SUMÁRIO

1 INTRODUÇÃO

O *Android*[1] é uma plataforma de código aberto baseado em *Linux* que serve como um sistema operacional móvel desenvolvido por um consórcio liderado pela *Google*. As aplicações para *Android* podem ser executadas em *smartphones*, *tablets*, *smart TVs*, etc… *Java* e *Kotlin* são as linguagens utilizadas para desenvolver essas aplicações, mas, atualmente, existem *frameworks* para o desenvolvimento de outras aplicações móveis utilizando várias linguagens, como *React Native*[2] e *Flutter*[3] (MASTG). A partir disso, o número de aplicações *Android* teve um crescimento considerável: segundo o site oficial da plataforma *Android*, mais de 2 milhões de aplicativos estão disponíveis na *Google Play Store*[4], loja oficial de aplicativos *Android* da *Google*. Esses aplicativos podem ser voltados para diversas finalidades, como entretenimento, comunicação, educação e finanças, que podem ser bastante relevantes para a sociedade. Esse número leva a crer que vulnerabilidades nas aplicações podem afetar pessoas no mundo todo.

Esse desenvolvimento pode fazer com que o programador se preocupe apenas com a implementação das funções, deixando um pouco a segurança de lado. Atualmente, a preocupação com a segurança está se fazendo bastante necessária devido à complexidade das aplicações e dos dados que elas podem movimentar. Por exemplo, aplicativos financeiros necessitam de total sigilo com as informações dos usuários, como saldo de conta, transações feitas, dentre outros dados.

Nesse contexto, os testes de segurança de aplicações móveis durante o ciclo de desenvolvimento estão ganhando força, principalmente nas fases de desenvolvimento e de testes propriamente ditos, em que são utilizadas abordagens estáticas e dinâmicas para analisar o código ou a aplicação em execução e identificar vulnerabilidades. Com a realização dos testes, é decidido os próximos passos do projeto, ou seja, se devem ser feitas melhorias na implementação ou se já pode avançar para a fase de produção do software.

Uma pesquisa realizada pela empresa de pesquisa e consultoria de mercado, *Strategy Analytics*, informou que, em 2021, a estimativa do número de pessoas que possuem e utilizam um celular é de cerca de 3,85 bilhões (*Strategy Analytics*, 2021).

---

[1] Site oficial da plataforma *Android* disponível em: https://www.android.com/intl/pt-BR_br/
[2] Mais informações sobre *React Native* disponíveis em: https://reactnative.dev/
[3] Mais informações sobre *Flutter* disponíveis em: https://flutter.dev/
[4] Disponível em: https://play.google.com/store/games?hl=pt_BR&gl=US

1.1 Objetivos

O principal objetivo deste trabalho é servir como uma espécie de tutorial e direcionamento para os leitores em como conduzir os testes de segurança nas aplicações *Android*, seguindo 2 guias metodológicos bastante eficientes desenvolvidos pela OWASP, que trazem requisitos de segurança que podem ser elencados em uma aplicação e os procedimentos de testes que verificam se os requisitos estão sendo seguidos. Foram utilizadas ferramentas, automatizadas e de uso manual, listadas pela OWASP, bem como os resultados que elas podem trazer referentes às vulnerabilidades presentes nas aplicações, além dos procedimentos de testes sugeridos pelo guia da OWASP. Também, baseado nesses resultados, é possível identificar requisitos de segurança que não estão sendo seguidos com a existência das vulnerabilidades. Informações adicionais, como as maneiras de configurar as ferramentas e o emulador *Android* utilizado para testar as aplicações, são deixadas nas referências, pois o foco do trabalho é em cima dos resultados que elas podem trazer.

1.1.1 Objetivos específicos

Além do objetivo geral, o trabalho explora mais alguns tópicos que se tornam relevantes e interessantes para apoiar o objetivo principal, que são:

- Mostrar o conjunto mínimo de ferramentas utilizadas em conjunto para constatar a existência da vulnerabilidade, podendo elas serem ferramentas estáticas ou dinâmicas. Por exemplo, para a vulnerabilidade X foi suficiente utilizar uma ferramenta de análise estática para comprovar a existência de X na aplicação, enquanto para comprovar a presença da vulnerabilidade Y foi preciso utilizar uma ferramenta estática e uma dinâmica;

- Feita a exploração e a constatação da existência da vulnerabilidade, algumas medidas para solucionar ou evitar o problema serão indicadas, que podem se resumir a citação de boas práticas de programação para um determinado caso de uso. Logo, esse objetivo visa mapear medidas para mitigar as vulnerabilidades.

1.2 Organização do trabalho

As seções seguintes tratam especificamente das segurança de aplicativos. Na

Seção 2, são abordados os conceitos teóricos relacionados ao tema, como a OWASP e seus guias metodológicos, ciclo de desenvolvimento seguro, teste de intrusão e conceitos sobre a plataforma *Android*. Na Seção 3, são abordadas informações sobre as ferramentas utilizadas, o emulador para testar as aplicações e as aplicações em si. Na Seção 4, a divisão foi baseada em cada vulnerabilidade listada pela OWASP. Para cada uma, foi mostrado o procedimento de teste utilizado, a ferramenta utilizada, a origem do problema, o requisito de segurança que não estava sendo respeitado e recomendações para solucionar o problema, Por último, na Seção 5, é feito um resumo do que foi encontrado na exploração da Seção 4 e o se pode concluir sobre os experimentos.

2 FUNDAMENTAÇÃO TEÓRICA

Ameaças (Peltier, 2010), na área de desenvolvimento de software, podem ser definidas como eventos indesejáveis que tenham chances de prejudicar um software ou sistema, causando prejuízos ao negócio, bem como pessoas e/ou instituições que tenham ligação com o ativo afetado.

Vulnerabilidades são falhas ou fraquezas de um software ou sistema que podem ser exploradas por meio de uma ameaça e, então, causar algum impacto negativo aos responsáveis pelo produto, dependendo do quão sensível o produto afetado pode ser em relação ao problema. Essas falhas podem ser oriundas de más práticas de programação e componentes desatualizados, por exemplo (Peltier, 2010).

Por fim, riscos são referentes à possibilidade de ameaças explorarem as vulnerabilidades existentes (Peltier, 2010). Nesse quesito, em um projeto de software normalmente haverá uma atenção especial a esse aspecto, definindo os riscos possíveis, o quanto eles podem afetar o produto, os meios para tratá-los e corrigi-los, além de poder definir quais os riscos que podem ser aceitáveis de existirem e não haver um tratamento específico para eles.

2.1 OWASP

Open Web Application Security Project[5] é um projeto de código aberto que visa estudar e prevenir as diversas vulnerabilidades possíveis em software, disponibilizando metodologia, documentação e tecnologia apropriadas para cumprir o seu objetivo.

[5] Disponível em: https://owasp.org/

2.1.1 Lista de vulnerabilidades da OWASP

Dentre os projetos que a OWASP gestiona, o OWASP *Top* 10[6] é um dos mais importantes, pois ele cataloga as 10 vulnerabilidades de segurança de aplicações mais comuns reportadas pela comunidade. A lista de vulnerabilidades mais conhecida da organização é em relação aos softwares desenvolvidos para a *World Wide Web* (Internet), atualizada em 2021. Existe também a OWASP *Mobile Top* 10[7], que traz a lista das 10 vulnerabilidades relacionadas a aplicações *Mobile* (aplicativos para dispositivos móveis), com sua última atualização datando de 2016.

As vulnerabilidades listadas no OWASP *Mobile Top* 10 estão organizadas na Tabela 1, numeradas com as suas respectivas posições na lista em relação ao número de casos reportados para essa vulnerabilidades:

Tabela 1 – Lista das vulnerabilidades listadas pelo OWASP *Mobile Top* 10

Número	Vulnerabilidade
1	Uso inadequado da plataforma
2	Armazenamento de dados inseguro
3	Comunicação insegura
4	Autenticação insegura
5	Criptografia insuficiente
6	Autorização insegura
7	Qualidade do código do cliente
8	Adulteração de código
9	Engenharia reversa
10	Funcionalidade estranha

Fonte: elaborado pelo autor

Uso inadequado da plataforma (1) se refere às falhas na utilização dos controles de segurança ou o uso indevido de algum recurso da plataforma. Chamadas de *Application Programming Interface* (API) são os principais pontos de entrada para esse tipo de vulnerabilidade, podendo incluir componentes *Intents* do *Android*, abordadas na Seção 2.4.

[6] Disponível em: https://owasp.org/www-project-top-ten/
[7] Disponível em: https://owasp.org/www-project-mobile-top-10/

Essas chamadas de API são problemáticas se codificadas com técnicas inseguras ou ultrapassadas, fazendo com o que o atacante possa enviar entradas maliciosas ou sequências de eventos inesperados, além de poder fazer essas chamadas da API fora do contexto em que ela é utilizada na aplicação.

Armazenamento de dados inseguro (2) pode ser definido por informações confidenciais (nomes de usuário, senhas, tokens de autenticação e dados de localização) armazenadas localmente no dispositivo que podem ser obtidas e lidas inapropriadamente, seja por um atacante que tenha acesso físico ao dispositivo ou por um malware instalado. Com isso, prejuízos como roubo de identidade, fraudes e perda material podem acontecer a partir do acesso indevido de dados, oriundo de uma má proteção do local do armazenamento. Não utilização de criptografia adequada ou a permissão de leitura por outras aplicações ativas são possíveis causas para a vulnerabilidade.

Comunicação insegura (3) pode se relacionar com a não utilização dos protocolos *Secure Sockets Layer* (SSL) e/ou *Transport Layer Security* (TLS), sendo mais utilizados em conjunto como SSL/TLS, em alguma parte da comunicação na rede do dispositivo alvo. Isso é uma porta de entrada para que um atacante possa monitorar a rede, podendo obter informações confidenciais do usuário. O atacante pode está utilizando a mesma rede *Wi-Fi* que a vítima como também pode estar sendo ajudado por um malware instalado no dispositivo que faz com que o tráfego de rede seja capturado e analisado. Com a possibilidade de capturar os dados, o atacante pode utilizá-los para se passar pela vítima em algum serviço de autenticação, por exemplo.

A autenticação é o processo para verificar a identidade e/ou veracidade de uma pessoa ou objeto. Feita de forma fraca, faz com que exista a possibilidade do atacante possa executar alguma funcionalidade oculta no aplicativo ou no *backend* por meio da própria aplicação, além de burlar o esquema de autenticação do aplicativo. Esse problema se relaciona com a vulnerabilidade autenticação insegura (4). Com a autenticação sendo burlada, também será possível explorar falhas de autorização, já que não houve a identificação correta do usuário no passo da autenticação e, consequentemente, a identificação das permissões delegadas a cada usuário.

A utilização de uma criptografia fraca, criptografia insuficiente (5), e que não esteja de acordo com as recomendações atuais pode dar um acesso relativamente fácil aos dados que se quer proteger. Quando ela é quebrada (decriptografada), pode resultar em roubo de informações, de código e de propriedade intelectual. Tais prejuízos são resultados de processos de gerenciamento de chaves ruins, criação e uso de protocolos de criptografia

personalizados e/ou uso de algoritmos inseguros e obsoletos, como *Rivest Cipher* 2 (RC2), *Message-Digest* 4 (MD4), MD5 e *Secure Hash Algorithm* 1 (SHA1).

Autorização insegura (6), que pode ser confundida com a autenticação insegura, pode ser observada se o usuário conseguir acessar uma funcionalidade privilegiada a qual ele não deveria ter acesso. Vale-se, também, de manipulações no binário da aplicação para poder identificar esse problema. Pode resultar em danos à reputação do desenvolvedor/responsável pela aplicação e dos usuários, fraude e roubo de informações.

A utilização de más práticas de programação pode trazer problemas relevantes, como estouros de buffer e vazamento de memória. Qualidade do código do cliente (7) é vulnerabilidade decorrente da não utilização de padrões de codificação consistentes e padronizados da linguagem utilizada, como utilização de APIs erradas ou inseguras e/ou obsoletas, bem como ter um código de difícil leitura e mal documentado.

A modificações de partes do aplicativo por meio de alterações no binário ou substituições de chamadas de API pode trazer informações valiosas da aplicação, da estrutura em si do aplicativo e/ou até mesmos dados confidenciais para o atacante. Isso se relaciona com a vulnerabilidade de adulteração de código (8). O ataque pode acontecer por meio de modificações no aplicativo e, após isso, induzir a vítima a instalar a aplicação adulterada em seu dispositivo que não seja por meio da loja de aplicativos oficial *Android*.

Engenharia reversa (9) de aplicativos móveis consiste no processo analisar o aplicativo compilado para obter informações sobre o funcionamento do código-fonte. Pode-se obter com essa análise, por exemplo, tabela de string original, código-fonte, bibliotecas, algoritmos e recursos incorporados ao aplicativo. Isso tudo pode revelar informações sobre servidores *backend*, constantes criptográficas e cifras, além do entendimento suficiente para realizar modificações de código subsequentes.

Funcionalidades e opções ocultas de forma proposital ou deixadas no código acidentalmente podem dar brechas para um atacante explorar a aplicação, se relacionando com a vulnerabilidade de funcionalidade estranha (10). Normalmente, esses desleixos acontecem em momentos de teste da aplicação, como deixar uma senha como um comentário em um aplicativo híbrido, que é um tipo de aplicativo e será abordado na Seção 2.4, e desativação da autenticação de dois fatores durante o teste, por exemplo.

2.1.2 Projeto MAS

Outro projeto interessante da OWASP é o *Mobile Application Security* (MAS[8]), que fornece uma avaliação de segurança bastante completa e com resultados convincentes em relação a aplicativos móveis com a intenção de ser um padrão industrial para a segurança desse tipo de aplicação. O projeto é formado por 2 guias metodológicos para ajudar a comunidade: o *Mobile Application Security Verification Standard* (MASVS[9]) e o *Mobile Application Security Testing Guide* (MASTG[10]).

O primeiro estabelece requisitos de segurança para aplicações móveis, podendo ser utilizado como orientação na fase de desenvolvimento e como uma lista de verificação de segurança. Além disso, o MASVS apresenta um modelo de segurança separado por níveis: L1, que é o nível básico de segurança que o projeto considera e que todos as aplicações devam cumprir os requisitos esse nível; L2, que é um nível de segurança mais elevado, em que contempla tanto os requisitos do L1 e alguns requisitos mais avançados; R, que já é considerado um nível paralelo aos outros dois por se tratar de requisitos relacionados à Engenharia Reversa e adulteração de código. Pode-se, também, combinar esses níveis para fornecer uma segurança mais reforçada, ou seja, “L1 + R” e “L2 + R”.

O segundo guia trata da execução dos testes de segurança, apresentando de forma detalhada as técnicas, os processos e as ferramentas utilizadas durante a análise de segurança dos aplicativos. Cada teste tem a identificação de um ou mais requisitos apresentados no MASVS, e o resultado do teste servirá para garantir se o requisito está sendo cumprido ou não na aplicação móvel. Por exemplo, o teste “Verificando a criptografia de dados na rede (MSTG-NETWORK-1)” faz a ligação com o requisito 1 da categoria de comunicação de rede, referenciada como *Network*, do MASVS. Além disso, o guia ainda descreve, com bastante clareza, características importantes acerca das arquiteturas das plataformas *Android* e iOS, que são as plataformas cobertas pelos testes. É válido ressaltar que ainda há requisitos do MASVS que a OWASP ainda não registrou oficialmente um teste direcionado no MASTG para poder cobrir o requisito. Gradativamente, esses testes vão sendo adicionados em versões futuras do guia.

---

[8] Disponível em: https://mas.owasp.org/
[9] Disponível em: https://mas.owasp.org/MASVS/
[10] Disponível em: https://mas.owasp.org/MASTG/

2.2 *Secure Software Development Life Cycle* (SSDLC)

Primeiramente, o conceito de *Software Development Life Cycle* (SDLC), que envolve o planejamento, implementação, testes e a implantação da aplicação, foi ganhando corpo e sendo bastante modificado em sua estrutura principal ao longo do tempo com o surgimento de novas metodologias de desenvolvimento de software. Originalmente, sua estrutura seguia uma linha bastante sequencial de processos desde o planejamento até a implantação, como a metodologia *Waterfall* (Mastg). Posteriormente, esses modelos foram se tornando defasados e problemáticos para certos tipos de projetos de software seguirem as diretrizes, já que não se podia voltar muitas etapas do desenvolvimento devido ao tempo e pelo fato de o produto só poder ser testado ou suas componentes serem executadas no fim do projeto. Com isso, novas metodologias foram surgindo, as chamadas metodologias ágeis, como o SCRUM, que consistem em ciclos de trabalho mais curtos, com a possibilidade de já poder ter uma visão das componentes e funções da aplicação executando à medida que os ciclos vão passando, tornando mais rápidas as correções de possíveis erros (Tung *et al.*, 2016; Khari *et al.*, 2016).

Contudo, a priori, essas metodologias não abordaram o quesito de segurança como um fator essencial na aplicação. Somente quando se tornou bastante significativo o número de incidentes relacionados à segurança que começou-se a pensar em integrar procedimentos de segurança no SDLC. Com isso, um novo termo foi criado e vem ganhando força com o passar do tempo, que é o *Secure* SDLC, ou SSDLC (SDLC Seguro).

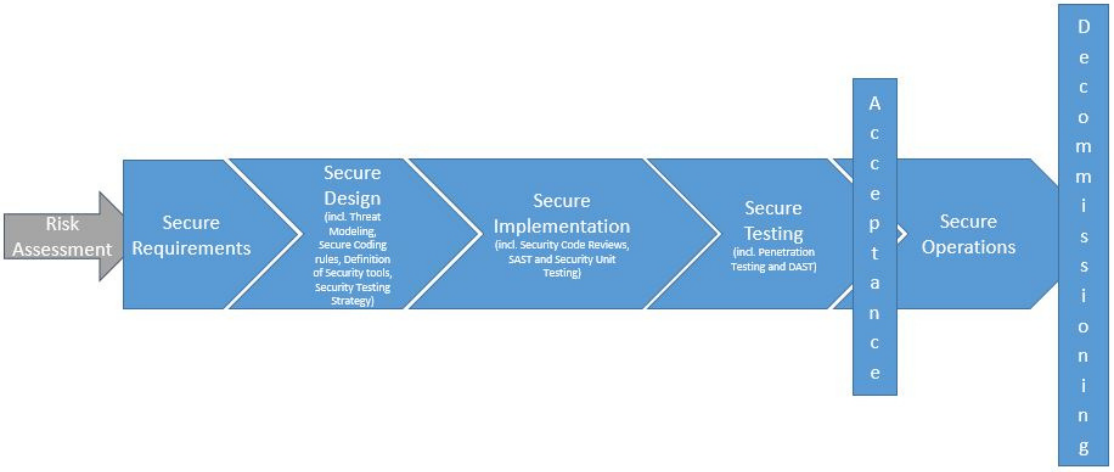
Os SSDLCs utilizados pelas diferentes equipes de desenvolvimento de software nas diferentes organizações sempre consistem nas mesmas etapas, sendo o processo geral no formato sequencial, lembrando as metodologias sequencias, como *Waterfall*, ao mesmo tempo que integra características dos modelos ágeis utilizados atualmente:

- Fase 1: Avaliação de risco (*Risk Assessment*): tenta identificar os riscos que a aplicação pode apresentar, seja no sentido financeiro, industrial e de marketing;
- Fase 2: Requisitos de segurança (*Security Requirements*): coletados junto com os requisitos funcionais no início do projeto;
- Fase 3: Modelagem de ameaças (*Secure Design*): que identifica e prioriza medidas para o tratamento de ameaças e torna clara a estratégia de testes que será adotada;
- Fase 4: Desenvolvimento seguro de software (*Secure Implementation*): fase em que realmente ocorre a implementação do código-fonte junto com revisões de código e testes de segurança estáticos;

- Fase 5: Teste de intrusão (*Secure Testing*): com um uma versão da aplicação, fazem-se os testes de penetração manual e com ferramentas automatizadas, em que os testes de segurança dinâmicos estão envolvidos, para verificar se o aplicativo está apto para entrar em produção ou se volta para o desenvolvimento para correções de bugs;
- Fase 6: Fase de produção (*Secure Operations*): após ser aceita na fase anterior, a aplicação é passada para a equipe de operações e posta em produção.

A figura a seguir exemplifica bem as fases principais listadas acima:

Figura 1 - Fases do SSDLC



Fonte: MASTG

Nesse sentido, ainda é válido ressaltar a relação entre a equipe de desenvolvimento de *software* (dev) com a equipe de operações (ops), que culminou no surgimento do modelo DevOps, que se relaciona com abordagens que tornam mais rápidos os processos necessários da implementação até a implantação em um ambiente de produção que gere valor ao usuário final. Esse modelo também precisou ganhar uma abordagem relacionada à segurança da aplicação e do processo como um todo para se adequar com as necessidades que foram surgindo (Izurieta et al., 2019; MASTG). Logo, a equipe de segurança (sec) foi integrada ao modelo, formando o DevSecOps. Vale ressaltar que essas equipes se comunicam continuamente durante todo o processo e, às vezes, é uma equipe só com membros destinados a cada uma das 3 áreas (Rangnau *et al.*, 2020; Sun *et al.*, 2021).

2.3 Teste de intrusão (*Pentesting*)

O teste de intrusão, mais conhecido como *Pentesting*, é uma atividade de segurança em que o testador que executa vai tentar identificar vulnerabilidades na aplicação alvo e tentar explorá-las, seja para fins profissionais, para reportar ao cliente os possíveis problemas encontrados, seja para fins maliciosos de hackers. Os principais tipos de *Pentesting* são: *black-box*, em que o testador tem conhecimento zero sobre a aplicação e sua estrutura, se comportando como um invasor real; *white-box*, em que o testador tem total conhecimento da aplicação, até mesmo do código-fonte às vezes; e *gray-box*, que é o meio-termo dos 2 primeiros, em que o testador tem acesso a algumas informações sobre a aplicação, como credenciais de acesso, e as outras informações devem ser descobertas (MASTG).

2.3.1 Teste de intrusão e SSDLC

Os SDLCs, que envolvem o planejamento, implementação, testes e a implantação da aplicação, mesmo que adaptados para cada tipo de projeto, seguem uma linha principal de etapas que, diferente de épocas passadas, se preocupam com a segurança atualmente, transformando-se em SSDLCs: avaliação de riscos, levantamento de requisitos (bem como requisitos de segurança), modelagem do tratamento de ameaças, implementação segura, teste de intrusão e implantação da aplicação (Junmei et al., 2021).

Na fase de levantamento de requisitos de segurança, fase 2, o MASVS pode ser utilizado para elencar os requisitos essenciais para aplicação. A fase de teste de intrusão, fase 5, se dá quando há uma versão candidata para lançamento e, então, o testador ou a equipe de testadores irá testar as vulnerabilidades, baseando-se nos requisitos de segurança apontados na 2º fase, e definir se a aplicação está pronta no sentido de estar segura ou se deve voltar para a implementação para corrigir *bugs*. Nesse momento, o MASTG é um ótimo guia para decidir as técnicas a serem utilizadas, bem como a forma de utilização das ferramentas, para averiguar se os requisitos de segurança estão sendo seguidos.

2.3.2 Testes estáticos e dinâmicos

O processo de procurar vulnerabilidades é muito importante para a segurança de uma aplicação, sendo possível ser feito manualmente ou por ferramentas e scanners automatizados. Duas dessas análises são as análises estática e dinâmica. A análise estática lê e

analisa o código-fonte e os componentes da aplicação sem executá-la, enquanto a análise dinâmica verifica o aplicativo em tempo de execução, normalmente verificando a entrada e saída de dados (MASTG). A análise dinâmica é mais utilizada com base nos resultados da análise estática para melhorar a mitigação das vulnerabilidades, pois, normalmente, não traz descobertas novas que a análise estática já não tenha confirmado, mas é uma forma interessante de identificar recursos e características desses recursos do ponto de vista do usuário. Os testes de intrusão utilizam esses 2 tipos de análise, que é a maneira mais adequada para se ter uma cobertura de segurança maior no aplicativo testado (Chao *et al.*, 2020; Li Li *et al.*, 2017).

2.4 Aplicações *Android*

Se tratando de aplicações móveis, elas podem ser diferenciadas em 3 tipos principais:

- Nativos, desenvolvidos para um sistema operacional específico, *Android* ou iOS, por meio de *Software Development Kit* (SDK);
- Para *Web*, que são executados no navegador do dispositivo como uma página web moderna, com as limitações dos componentes gerais impostas por serem executados dentro do navegador;
- Híbridos, que integra os outros 2 tipos, sendo que uma parte deles são executadas num navegador incorporado, que é chamado de *WebView*.

Os aplicativos para *Android* são compilados em *bytecode Dalvik*, uma versão otimizada do *bytecode Java*, e executados pela Máquina Virtual *Dalvik*, semelhante à JVM. O *bytecode*, então, é armazenado em arquivos DEX, enquanto arquivos na linguagem *Java* tradicionais são compilados em arquivos CLASS, e compactados em um único arquivo APK ou AAB. Por fim, a decompilação de um aplicativo *Android* consiste em descompactar o arquivo APK, por exemplo, e obter os arquivos DEX, para a partir destes obter novamente o código-fonte *Java*. Ferramentas automatizadas utilizam desse procedimento para fazer análises estáticas e dinâmicas de aplicação no formato APK (Sarkar *et al.* 2019; Mateless *et al.*, 2020).

Todo aplicativo no formato APK contém um arquivo no diretório raiz chamado "*AndroidManifest.xml*", o arquivo Manifesto, o qual descreve toda a estrutura do aplicativo, descrevendo permissões e componentes utilizados pela aplicação. No contexto de *Android*, uma componente comum é a *Activity*, que é responsável por todas as interações das telas do

aplicativo, ou seja, a parte visível para o usuário, gerenciando os eventos que possam ocorrer. Para cada tela, existe somente uma única *Activity* (MASTG).

Outro componente importante é o *Intent*, que faz a solicitação da ação de algum outro componente do mesmo aplicativo ou de outro instalado no dispositivo, ou seja, é um objeto de mensagem. Existem os *Intents* explícitos, que são indicados para solicitar ações de outro componente da mesma aplicação, e os *Intents* implícitos, que indicam uma ação geral para realizar e podem ser processados por componentes de outros aplicativos.

Intents têm a ver com um conceito denominado *Inter-Process Communication* (IPC), que tem a ver com o mecanismo de comunicação entre processos dos aplicativos no geral, que pode enviar dados e sincronizar ações. Os *Intents* são um desses mecanismos de comunicação.

No que se refere à conexão e comunicações de rede, o protocolo HTTPS deve ser utilizado em toda e qualquer comunicação que ocorra envolvendo o aplicativo, pois irá proteger com mais eficiência os dados que são transmitidos na rede. Nesse contexto, a conexão com um *host* exige a utilização de um certificado assinado por uma autoridade certificadora confiável para garantir a segurança na comunicação, fazendo com que a aplicação rejeite qualquer comunicação que não esteja envolvendo o certificado ou que ele esteja mal configurado. O tipo mais comum de certificado considerado seguro é o X.509 (MASTG).

No tocante à tradução do binário final da aplicação para linguagem mais natural, no formato de APK, vale ressaltar 2 conceitos importantes: desmontadores e decompiladores. Ambos envolvem o processo de traduzir o código binário ou *bytecode* para uma linguagem mais perto da humana. Os desmontadores estão ligados à conversão de código de máquina em código *assembly*, que é conhecido como "*smali*" para os arquivos DEX no contexto de *Java* para *Android*. Já os decompiladores utilizam esse código *assembly* para traduzir de volta para o código *Java*, que é mais fácil de ler do que o *assembly*, mas pode ser um pouco mais impreciso. Essas 2 técnicas são utilizadas para análise de código fonte a partir de uma APK (Gonzalez *et al.*, 2015; Ziadia *et al.*, 2020).

3 METODOLOGIA

As ferramentas de testes, o emulador de *Android* e as aplicações *Android* vulneráveis utilizados neste trabalho são listadas. Note que as ferramentas de testes, tanto para análises estáticas quanto dinâmicas, são indicadas no MASTG citado na seção 2.1.2. As

configurações utilizadas para cada ferramenta podem ser visualizadas com mais atenção nas referências indicadas.

3.1 MobSF

MobSF[11] é uma ferramenta automatizada utilizada para *pentesting*, análise de malware e fornece uma avaliação de testes estáticos e dinâmicos em aplicações móveis, ou seja, faz a busca por vulnerabilidades, suportando alguns binários como o APK.

Algumas informações que a análise da ferramenta pode trazer são: análise das permissões da aplicação, bem como o impacto delas; avaliação de ameaças a nível de código binário; análise do arquivo Manifesto, focando nas componentes da aplicação; e a análise do código fonte, que é convertido para *Java* após a decompilação da APK e analisando as possíveis vulnerabilidades existentes, além de indicar em que arquivos elas ocorrem. Para este trabalho, esta ferramenta será utilizada na abordagem de testes estáticos.

3.2 adb

O adb[12] é uma ferramenta de linha de comando bastante prática e que permite a comunicação com um dispositivo *Android*. Por meio dela, são permitidas variadas ações no dispositivo, como instalação e depuração de aplicativos, visualização das mensagens de *log* e acesso a um terminal *Unix*, já que a plataforma *Android* é baseada em *Linux*, e, com isso, mais opções são possíveis de serem executadas no dispositivo. Portanto, é uma ferramenta que é utilizada numa abordagem de análise dinâmica.

Fazendo parte do pacote *Android* SDK, a ferramenta é uma forma de preencher o espaço entre o ambiente de desenvolvimento/testes local e o dispositivo, que pode se conectar via USB ou *Wi-Fi*. Para este trabalho, a ferramenta será utilizada para instalar as aplicações, visualizar mensagens de *log* e acessar o terminal *Unix* no dispositivo.

[11] Disponível em: https://mobsf.github.io/docs/#/
[12] Disponível em: https://developer.android.com/studio/command-line/adb?hl=pt-br

3.3 *Apktool*

É uma ferramenta voltada para engenharia reversa de aplicativos *Android* binários, na forma de APK, feitos por terceiros. Basicamente, o *apktool*[13] decompila a aplicação do formato binário e recupera os recursos para um formato próximo ao original. Ela tanto pode fazer a decompilação do binário APK como pode realizar o processo inverso, reconstruindo-os após o usuário fazer alterações à sua maneira. Logo, é uma ferramenta que deve ser usada em uma abordagem estática.

Dentre os arquivos oriundos da decompilação, podem ser citados: "*AndroidManifest.xml*", decodificado do XML binário para o XML em texto e que pode ser lido e editado por um editor de texto; "original", pasta com o arquivo "*MANIFEST.MF*", que mantém as informações acerca dos arquivos presentes no arquivo JAR; "res", diretório contendo todos os recursos da aplicação, em arquivos com sintaxe da linguagem *Java*; e "*smali*", que é o diretório contendo o *bytecode Dalvik* desmontado.

3.4 *Burp Suite*

O *Burp Suite*[14] é uma plataforma integrada utilizada para realizar testes de segurança em aplicativos móveis e *Web*. Ela contém diversas ferramentas para mapeamento de vulnerabilidades, mas, no escopo deste trabalho, será utilizado apenas o *Burp Proxy*, que funciona como um *proxy* na *Web* e é posicionado entre o servidor e o cliente que, no caso, é o dispositivo *Android* e as aplicações contidas nele. Ele é administrado pela interface principal do *Burp Suite*. Com isso, é possível interceptar, ler e modificar o tráfego HTTP(S) de entrada e saída do dispositivo.

Considerando o tráfego HTTP de um dispositivo *Android*, o *Burp Suite* pode interceptar todas as comunicações que estão acontecendo de todos os processos e aplicações que precisam de acesso à rede. Contudo, para averiguar tráfego HTTPS, é necessária a instalação de um certificado de uma autoridade certificadora (CA) no dispositivo, exceto em casos em que a aplicação não esteja verificando corretamente a integridade do certificado utilizado na conexão.

---

[13] Disponível em: https://ibotpeaches.github.io/Apktool/
[14] Disponível em: https://portswigger.net/burp/documentation/desktop

3.5 Emulador de dispositivo

Ao invés de se utilizar um dispositivo físico para instalar as aplicações e fazer a análise delas, é mais conveniente e seguro utilizar um emulador de dispositivo *Android*. Desta forma, evita-se danificar o dispositivo físico acidentalmente, além de ser mais fácil de gerenciar configurações e recursos.

Com isso, se introduz o conceito de *sandbox*, que é um ambiente de teste, mais especificamente um sistema operacional, isolado que permite a execução de programas e aplicações com mais segurança, em que outros programas em execução fora desse ambiente não interferem com a execução do que está na *sandbox* e nem vice-versa. Qualquer falha ou vulnerabilidade presente em uma aplicação executada nesse ambiente só poderá afetar, no máximo, o ambiente em que ela está inserida, não afetando o dispositivo. O emulador *Android* funcionará com o ambiente descrito.

Existem diversos emuladores que podem ser utilizados, tanto pagos como gratuitos, e, para este trabalho, será utilizado uma máquina virtual do *VirtualBox*[15] que irá executar a imagem oficial do *Android*[16] [17]. Logo, quando a ferramenta adb for utilizada, ela se conectará com o dispositivo por meio da rede local.

3.6 APKs utilizadas

Este trabalho, como dito na Seção 1.2, tem por objetivo principal servir como tutorial para realização dos testes e, por questões de privacidade e praticidade, as aplicações utilizadas estão disponíveis publicamente. Ao todo, foram utilizadas 5 aplicações, no formato APK, para a realização dos experimentos, sendo 3 referenciadas no próprio MASTG para realização e práticas dos testes (Diva, *AndroGoat* e *InsecureBankv2*) e 2 advindas de pesquisa externa (*InsecureShop* e *VulnApp*). A escolha desses aplicativos se deve pelo fato deles, reunidos, conterem as 10 vulnerabilidades listadas pela OWASP *Mobile Top Ten*, Tabela 1, e que serão exploradas ao longo da Seção 4.

---

[15] Disponível em: https://www.virtualbox.org/
[16] Disponível em: https://www.android-x86.org/
[17] Tutorial de como instalar e configurar imagem *Android* no *VirtualBox*. Disponível em: https://www.youtube.com/watch?v=QTVz1oH_heU

3.7 Descrição da realização dos testes

Primeiramente, foram utilizadas as ferramentas de análise estática em cada uma das 5 APKs para verificar que vulnerabilidades da OWASP *Mobile Top* 10 elas apresentavam. Após isso, nas vulnerabilidades que a análise estática não apresentava um resultado tão comprobatório da existência dela, as ferramentas de análise dinâmica eram utilizadas para deixar mais evidente que a vulnerabilidade realmente estava presente na aplicação.

Em relação às fases do SSDLC, de acordo com a Seção 2.2 e a Figura 1, quando uma ferramenta de análise estática for utilizada, ela estaria se referenciando à fase de desenvolvimento seguro, fase 4, ao passo que a utilização de uma ferramenta de análise dinâmica estaria relacionada com a fase de teste de intrusão, fase 5.

4 EXECUÇÃO DOS TESTES

Nesta seção serão abordadas as 10 vulnerabilidades listadas no OWASP *Mobile Top Ten*, exibidas na Tabela 1 e abordadas uma a uma na Seção 2.1.1, que foram encontradas nas APKs citadas na Seção 3.6. Para cada APK citada na Seção 3.6, serão evidenciadas as vulnerabilidades da OWASP *Mobile Top Ten*, exibidas na Tabela 1 e descritas na Seção 2.1.1, que ela contém. Logo, com a avaliação das 5 APKs, serão totalizadas as 10 vulnerabilidades da lista.

Para cada vulnerabilidade, serão mostradas as ferramentas estáticas e/ou dinâmicas utilizadas para atestar a existência dela, fazendo menção ao requisito do MASVS, citado na Seção 2.1.2, que está sendo afetado e ao procedimento de teste apropriado descrito no MASTG, citado na Seção 2.1.2, para identificação da vulnerabilidade. Além disso, sugestões de como corrigir o caso de uso e outras recomendações também são indicadas.

4.1 Diva

(*Damn insecure and vulnerable App*) é uma aplicação que foi projetada para ser insegura com o objetivo de ensinar falhas oriundas de codificação ruim ou insegura. Recebeu atualizações em 2016 e contém 13 desafios diferentes.

4.1.1 Uso inadequado da plataforma (Posição 1)

Utilizando a ferramenta MobSF, Figura 2, e observando o arquivo Manifesto, Figura 3, como mencionado na Seção 2.4, foi detectada a presença de um *intent-filter*, que reúne as *intents* implícitas a serem utilizadas, na componente *Activity* "*jakhar.aseem.diva.APICredsActivity*", à qual chama uma das telas da aplicação, tornando-a explicitamente exportável e fazendo com que outra aplicação instalada no mesmo dispositivo possa acessar, além de também poder capturar, as possíveis informações contidas na componente sem ter que executar de fato a aplicação.

A detecção da ferramenta e o trecho do código onde ocorre o problema podem ser vistos nas Figuras 2 e 3, respectivamente, a seguir:

Figura 2 - Detecção da ferramenta MobSF da presença de uma *Activity* exportada publicamente na aplicação Diva

3	**Activity** (jakhar.aseem.diva.APICredsActivity) is not Protected. An intent-filter exists.	warning	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Activity is explicitly exported.

Fonte: elaborado pelo autor.

Figura 3 - Trecho do código onde mostra a presença perigosa de uma *Activity* exportada publicamente na aplicação Diva

```
<activity android:label="@string/apic_label" android:name="jakhar.aseem.diva.APICredsActivity">
    <intent-filter>
        <action android:name="jakhar.aseem.diva.action.VIEW_CREDS" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Fonte: elaborado pelo autor.

Isso afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-PLATFORM-4: O aplicativo não exporta funcionalidades sensíveis através do IPC, a menos que esses mecanismos estejam devidamente protegidos" e o teste do MASTG para averiguar esse requisito é denominado "Teste para exposição de funcionalidade sensível por meio de IPC (MSTG-PLATFORM-4)"[18]. Se a lógica da aplicação indicar que esse componente só deva ser executado pelo próprio aplicativo, não se usa *intent-filter*, e sim define o atributo "*exported*" como "*false*" para a componente e utiliza-se *intents* explícitas.

[18] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05h-Testing-Platform-Interaction/#testing-for-sensitive-functionality-exposure-through-ipc-mstg-platform-4

4.1.2 Armazenamento de dados inseguro (Posição 2)

Primeiramente, utilizando a ferramenta MobSF, foi detectado que os dados movidos na aplicação eram registrados no *log* do dispositivo, mostrado na Figura 4, e o trecho de código onde acontece o problema é exibido na Figura 5. O problema acontece se dados confidenciais estiverem sendo registrados em texto plano, de modo que alguém que esteja monitorando o *log* possa identificar esses dados.

Na Figura 4, também, é visto que a análise da ferramenta citou uma referência a *Common Weakness Enumeration* (CWE), que é uma lista de tipos de vulnerabilidades em software e hardware criada e gerida pela MITRE, organização norte-americana que gerencia centros de pesquisa e desenvolvimento financiados pelo governo dos EUA. CWE-532[19] remete a "Inserção de informações confidenciais no arquivo de *log*". As figuras 4 e 5 são exibidas a seguir:

Figura 4 - Detecção da ferramenta MobSF que as informações são registradas nas mensagens de *log* na aplicação Diva

NO	ISSUE	SEVERITY	STANDARDS
1	The App logs information. Sensitive information should never be logged.	info	**CWE:** CWE-532: Insertion of Sensitive Information into Log File **OWASP MASVS:** MSTG-STORAGE-3

Fonte: elaborado pelo autor.

[19] CWE-532: https://cwe.mitre.org/data/definitions/532.html

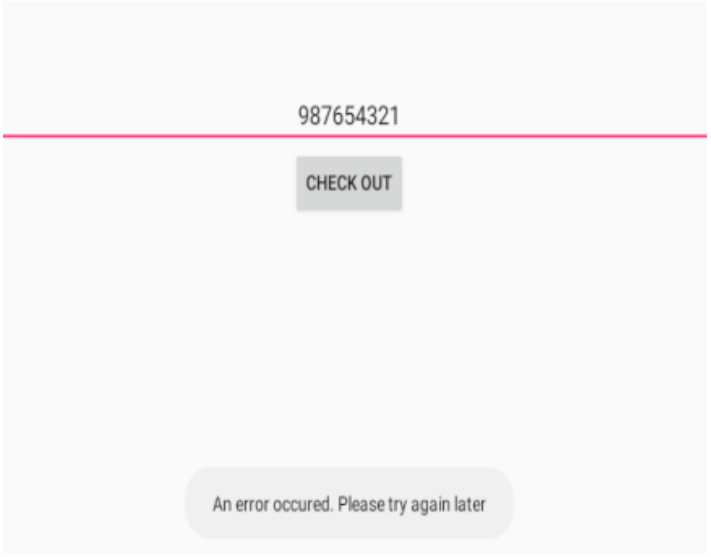
Figura 5 - Trecho de código que mostra que a informação está sendo registrada em mensagens de *log* na aplicação Diva

```
public void checkout(View view) {
    EditText cctxt = (EditText) findViewById(R.id.ccText);
    try {
        processCC(cctxt.getText().toString());
    } catch (RuntimeException e) {
        Log.e("diva-log", "Error while processing transaction with credit card: " + cctxt.getText().toString());
        Toast.makeText(this, "An error occured. Please try again later", 0).show();
    }
}
```

Fonte: elaborado pelo autor.

Fazendo uso da aplicação na funcionalidade de cadastro de um cartão de crédito, Figura 6, e com a utilização da ferramenta adb e o seu utilitário *logcat* - monitoramento das mensagens de *log* do dispositivo -, é possível verificar que dados confidenciais estavam aparecendo em texto plano entre as mensagens de *log*, Figura 7. Nela, é possível ver que o dado inserido está contido na mensagem de erro gerada, o que é inapropriado. As Figuras 6 e 7 são mostradas a seguir:

Figura 6 - Visualização da utilização da aplicação em que é detectado que informações sensíveis estão sendo vazadas na aplicação Diva



Fonte: elaborado pelo autor.

Figura 7 - Visualização da mensagem de *log* que representa um erro na aplicação em que é exibida a informação digitada na figura 6

```
10-12 19:39:36.612  3573  3573 E diva-log: Error while processing
transaction with credit card: 987654321
```

Fonte: elaborado pelo autor.

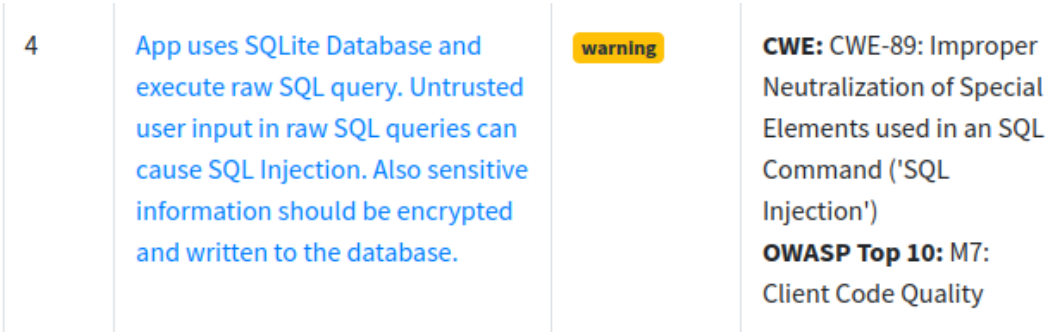
Esse problema afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-STORAGE-3: Dados sensíveis não podem aparecer nos '*logs*' de aplicação". O teste do MASTG para averiguar esse requisito é denominado "Testando *logs* para dados confidenciais (MSTG-STORAGE-3)"[20]. Considere bem o que deve ser passado nas mensagens de *log* e não registre informações confidenciais como parte de uma operação normal e por conta própria - que não seja mensagem gerada pelo próprio sistema *Android* e sem intervenção do programador - a não ser que seja estritamente importante para a aplicação.

4.1.3 Qualidade do código do cliente (Posição 7)

Utilizando a ferramenta MobSF para analisar o aplicativo, foram detectadas que estão sendo feitas consultas SQL de forma bruta com os dados inseridos pelo usuário. Isso pode causar o problema de injeção SQL, que consiste em uma consulta SQL imprópria ao banco de dados a partir dos campos de entrada de dados da aplicação.

A detecção da ferramenta é mostrada na Figura 8, enquanto o trecho do código onde ocorre o problema é indicado na Figura 9. A CWE é citada novamente, sendo a "CWE-89[21]: Neutralização imprópria de elementos especiais usados em um comando SQL ('SQL Injection')", que remete ao tratamento incorreto ou inexistente dos dados de entrada, podendo deixar uma consulta SQL imprópria ser feita.

Figura 8 - Detecção da ferramenta MobSF do tratamento incorreto de dados de entrada do usuário do aplicativo Diva



Fonte: elaborado pelo autor.

[20] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05d-Testing-Data-Storage/#testing-logs-for-sensitive-data-mstg-storage-3

[21] CWE-89: https://cwe.mitre.org/data/definitions/89.html

Figura 9 - Trecho de código onde mostra o tratamento incorreto de dados de entrada do usuário, mais especificamente na linha 35, do aplicativo Diva

```
32.     public void search(View view) {
33.         EditText srchtxt = (EditText) findViewById(R.id.ivi1search);
34.         try {
35.             Cursor cr = this.mDB.rawQuery("SELECT * FROM sqliuser WHERE user = '" + srchtxt.getText().toString() + "'", null);
36.             StringBuilder strb = new StringBuilder("");
37.             if (cr == null || cr.getCount() <= 0) {
38.                 strb.append("User: (" + srchtxt.getText().toString() + ") not found");
39.             } else {
40.                 cr.moveToFirst();
41.                 do {
42.                     strb.append("User: (" + cr.getString(0) + ") pass: (" + cr.getString(1) + ") Credit card: (" + cr.getString(2) + ")\n");
43.                 } while (cr.moveToNext());
44.             }
45.             Toast.makeText(this, strb.toString(), 0).show();
46.         } catch (Exception e) {
47.             Log.d("Diva-sqli", "Error occurred while searching in database: " + e.getMessage());
48.         }
49.     }
```

Fonte: elaborado pelo autor.

Esse problema afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-PLATFORM-2: Todas as entradas de fontes externas e do usuário são validadas e, se necessário, sanitizadas. Isso inclui dados recebidos através da UI, mecanismos de IPC como intenções, URLs personalizados e origens pela rede". O teste do MASTG para averiguar esse requisito é denominado "Teste de falhas de injeção (MSTG-PLATFORM-2)"[22]. Para evitar que um usuário possa fazer uma consulta SQL inadequada e mal intencionada, deve-se, por exemplo, criar listas de entradas aceitáveis, fazer sempre a verificação do dado inserido e utilizar as bibliotecas mais indicadas pela documentação oficial da linguagem utilizada para seguir os padrões de segurança da linguagem.

4.1.4 Engenharia reversa (Posição 9)

Analisando o aplicativo visualmente, pode-se perceber uma funcionalidade de autenticação em que o usuário deve digitar uma senha de acesso à qual não se tem nenhuma informação à princípio. A interface é mostrada na Figura 10 a seguir:

[22] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05h-Testing-Platform-Interaction/#testing-for-injection-flaws-mstg-platform-2

Figura 10 - Funcionalidade de acesso do aplicativo Diva

Fonte: elaborado pelo autor.

Utilizando a ferramenta MobSF e observando o código fonte da aplicação após a decompilação, o arquivo referente a "*HardCodeActivity*" contém o procedimento referente à funcionalidade de acesso descrita. Nesse procedimento, pode-se verificar a comparação do dado inserido pelo usuário com a *string* "*vendorsecretkey*", em texto simples, que é a senha secreta e que pode ser descoberta observando o código. A visualização do código, mais especificamente na linha indicada pela seta vermelha, pode ser vista na Figura 11 a seguir:

Figura 11 - Trecho de código referente ao procedimento de acesso do aplicativo Diva

```
public void access(View view) {
    EditText hckey = (EditText) findViewById(R.id.hcKey);
    if (hckey.getText().toString().equals("vendorsecretkey")) {
        Toast.makeText(this, "Access granted! See you on the other side :)", 0).show();
    } else {
        Toast.makeText(this, "Access denied! See you in hell :D", 0).show();
    }
}
```

Fonte: elaborado pelo autor.

Esse problema afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-AUTH-1: Se o aplicativo fornecer aos usuários acesso a um serviço remoto, alguma forma de autenticação, como autenticação de nome de usuário e senha, será executada no terminal remoto". O teste do MASTG para averiguar esse requisito é denominado "Testando

credenciais de confirmação (MSTG-AUTH-1 e MSTG-STORAGE-11)"[23]. Toda chave de identificação e/ou dados confidenciais devem ser retirados do código fonte, além de utilizar a técnica de ofuscação em partes importantes do código para não revelar a lógica principal do aplicativo, equilibrando com a perda de desempenho que a técnica promove à aplicação.

4.2 *AndroGoat*

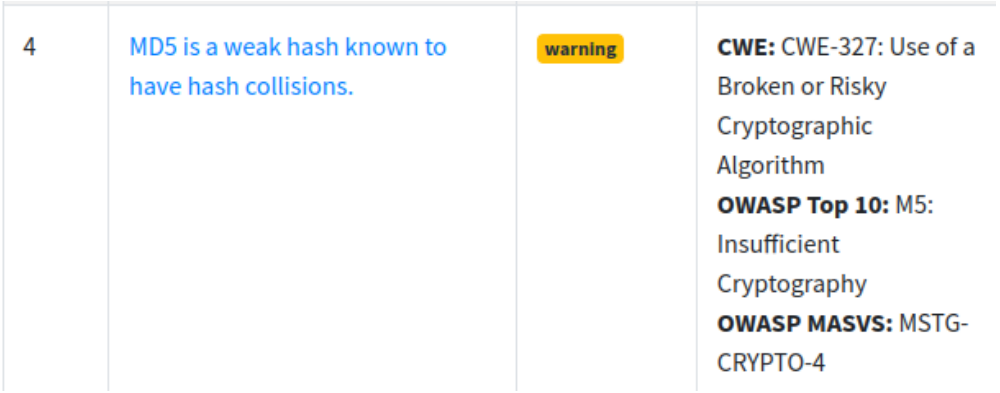
É um aplicativo inseguro que foi desenvolvido em *Kotlin* e contém mais de 20 vulnerabilidades para serem testadas.

4.2.1 Criptografia insuficiente (Posição 5)

Utilizando a ferramenta MobSF para analisar o aplicativo, foi detectado que está sendo utilizada uma função hash criptográfica MD5, que já é considerada problemática pelo risco de colisões entre os códigos *hash*.

A Figura 12 apresenta o *print* da detecção da ferramenta, enquanto a Figura 13 mostra o trecho do código onde ocorre o problema, mais especificamente na linha 112. Mais uma vez a CWE é referenciada, mais especificamente a "CWE-327[24]: Uso de um Algoritmo Criptográfico Quebrado ou Arriscado", referente a algoritmos defasados e que não representam tanta segurança atualmente.

Figura 12 - Detecção da ferramenta MobSF da utilização de função criptográfica problemática do aplicativo *Androgoat*



Fonte: elaborado pelo autor.

[23] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05f-Testing-Local-Authentication/#testing-confirm-credentials-mstg-auth-1-and-mstg-storage-11

[24] CWE-327: https://cwe.mitre.org/data/definitions/327.html

Figura 13 - Trecho do código que mostra a utilização da função criptográfica problemática do aplicativo *Androgoat*

```
110.      public final String hashPIN(String pinValue) {
111.          Intrinsics.checkParameterIsNotNull(pinValue, "pinValue");
112.          MessageDigest messageDigest = MessageDigest.getInstance("MD5");
113.          byte[] bytes = pinValue.getBytes(Charsets.UTF_8);
114.          Intrinsics.checkExpressionValueIsNotNull(bytes, "(this as java.lang.String).getBytes(charset)");
115.          byte[] digest = messageDigest.digest(bytes);
116.          Intrinsics.checkExpressionValueIsNotNull(digest, "MessageDigest.getInstanc…t(pinValue.toByteArray())");
117.          String md = ArraysKt.joinToString$default(digest, (CharSequence) "", (CharSequence) null, (CharSequence) nu
118.          return md;
119.      }
120.  }
```

Fonte: elaborado pelo autor.

Esse problema está afetando o requisito de segurança do MASVS de nível L1 denominado "MSTG-CRYPTO-4: O aplicativo não utiliza protocolos criptográficos ou algoritmos que são considerados amplamente obsoletos para uso em segurança". O teste do MASTG para averiguar esse requisito é denominado "Testando a configuração de algoritmos padrão criptográficos (MSTG-CRYPTO-2, MSTG-CRYPTO-3 e MSTG-CRYPTO-4)"[25]. Esse problema pode ser solucionado modificando a função hash para SHA-2, que suporta tamanho de mensagens hash até 512 bits, enquanto a MD5 só suporta 128 bits.

Ademais, se tratando de algoritmos criptográficos, é uma boa prática sempre buscar aqueles que sejam classificados ou publicados pela NIST[26] mais recentemente e seguir os padrões da linguagem utilizada e da plataforma *Android*. Por exemplo, o módulo *Conscrypt*[27] otimiza e melhora a segurança do dispositivo *Android*, utilizando código *Java* e fornecendo uma biblioteca nativa para desenvolvimento em plataforma *Android*, além de conter diversos algoritmos atualizados.

4.3 *InsecureBankv2*

É uma atualização de um projeto mais antigo, o *InsecureBank*, o qual foi projetado para ser explorado por apreciadores de segurança de aplicações, podendo encontrar diversas vulnerabilidades nele. O *backend* foi projetado em *Python*.

[25] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05e-Testing-Cryptography/#testing-the-configuration-of-cryptographic-standard-algorithms-mstg-crypto-2-mstg-crypto-3-and-mstg-crypto-4
[26] Algoritmos recomendados: https://www.keylength.com/en/4/
[27] Mais sobre Concrypt: https://source.android.com/docs/core/architecture/modular-system/conscrypt

4.3.1 Autenticação insegura (Posição 4)

Utilizando a ferramenta MobSF para analisar o aplicativo e observando o arquivo Manifesto, mencionado na Seção 2.4, foi observado que 4 *Activities* relacionadas com o procedimento de *login* estavam sendo exportadas explicitamente ("*PostLogin*", "*DoTransfer*", "*ViewStatement*" e "*ChangePassword*"), por meio do atributo "*exported*" com valor "*true*". A Figura 14 a seguir mostra o ocorrido:

Figura 14 - Trecho do código onde mostra as *Activities* exportadas na aplicação *InsecureBankv2*

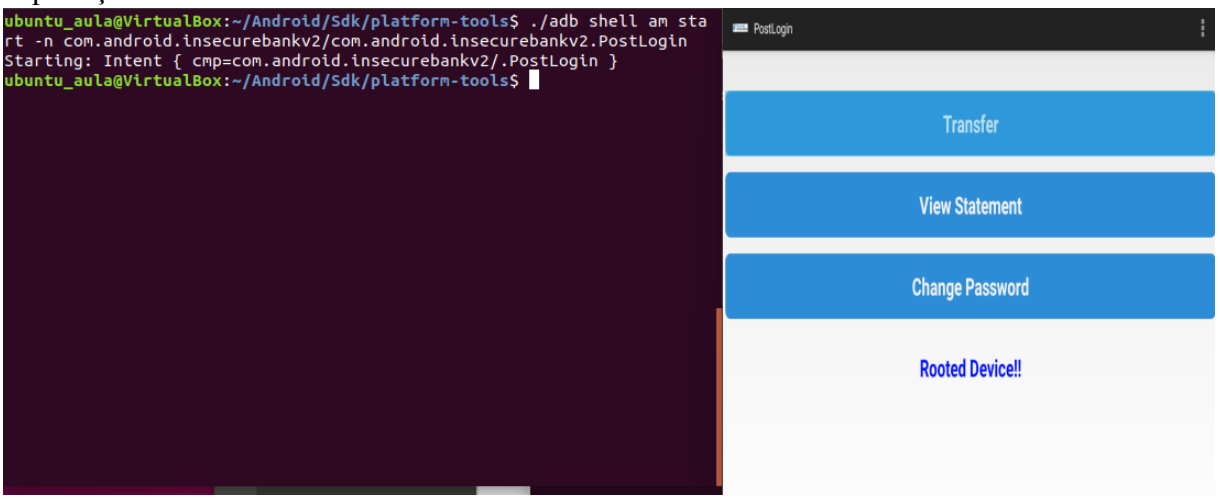
```
    <activity android:exported="true" android:label="@string/title_activity_post_login"
android:name="com.android.insecurebankv2.PostLogin"/>
    <activity android:label="@string/title_activity_wrong_login"
android:name="com.android.insecurebankv2.WrongLogin"/>
    <activity android:exported="true" android:label="@string/title_activity_do_transfer"
android:name="com.android.insecurebankv2.DoTransfer"/>
    <activity android:exported="true" android:label="@string/title_activity_view_statement"
android:name="com.android.insecurebankv2.ViewStatement"/>
    <provider android:authorities="com.android.insecurebankv2.TrackUserContentProvider" android:exported="true"
android:name="com.android.insecurebankv2.TrackUserContentProvider"/>
    <receiver android:exported="true" android:name="com.android.insecurebankv2.MyBroadCastReceiver">
        <intent-filter>
            <action android:name="theBroadcast"/>
        </intent-filter>
    </receiver>
    <activity android:exported="true" android:label="@string/title_activity_change_password"
 android:name="com.android.insecurebankv2.ChangePassword"/>
```

Fonte: elaborado pelo autor.

Perceba que é mais um problema envolvendo *Activities*, assim como na Seção 4.1, mas em uma abordagem diferente. Assume-se, primariamente, que a *Activity* "*PostLogin*" está relacionada com a exibição de uma tela específica que só deveria ser acessada após o *login* feito com sucesso. Contudo, com a utilização da ferramenta adb e seu comando "*shell*", a *Activity* é invocada com sucesso pela linha de comando, já que ela é exportável. Com isso, a tela de dados e funções do usuário autenticado, que só seria apresentada após o *login* efetuado com sucesso, é acessada, mostrando que a autenticação foi completamente ignorada.

Isso pode ter um impacto bem considerável se tratando de informações importantes, como os dados do usuário. O procedimento descrito é visto na Figura 15 a seguir:

Figura 15 - Resultado no aplicativo ignorando o procedimento de *login* na aplicação *InsecureBankv2*



Fonte: elaborado pelo autor.

O problema afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-PLATFORM-4: O aplicativo não exporta funcionalidades sensíveis através do IPC, a menos que esses mecanismos estejam devidamente protegidos". O teste do MASTG para averiguar esse requisito é denominado "Teste para exposição de funcionalidade sensível por meio de IPC (MSTG-PLATFORM-4)"[28]. Como recomendações, atentar-se aos componentes exportados da aplicação, não deixando nenhum que seja referente a eventos privados na lógica da aplicação, como procedimento de autenticação, de forma pública. Além disso, as autenticações devem ser feitas pelo lado do servidor e deve-se evitar guardar e compartilhar dados de autenticação localmente no dispositivo.

4.3.2 Autorização insegura (Posição 6)

Utilizando a ferramenta MobSF para analisar e acessando o arquivo de código fonte referente a "*LoginActivity*", observa-se o trecho de código que, aparentemente, tem uma funcionalidade de administrador oculta baseada no valor da string "*is_admin*". Caso o valor seja "*no*", o botão é ocultado da tela, como mostra a Figura 16 a seguir:

[28] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05d-Testing-Data-Storage/#testing-logs-for-sensitive-data-mstg-storage-3

Figura 16 - Funcionalidade de administradores oculta do aplicativo *InsecureBankv2*

```
@Override // android.app.Activity
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_log_main);
    String mess = getResources().getString(R.string.is_admin);
    if (mess.equals("no")) {
        View button_CreateUser = findViewById(R.id.button_CreateUser);
        button_CreateUser.setVisibility(8);
    }
```

Fonte: elaborado pelo autor.

Utilizando a ferramenta apktool para decompilar apk, e acessando o arquivo "*strings.xml*", localizado em "*/res/values/*", onde ficam armazenados todos os recursos envolvendo *strings*, é possível localizar "*is_admin*" atribuída com o valor "*no*", como mostra a Figura 17 a seguir:

Figura 17 - Visualização da string "is_admin" do aplicativo *InsecureBankv2*

```
<string name="hello_world">Hello world!</string>
<string name="is_admin">no</string>
<string name="loginscreen_password">Password:</string>
<string name="loginscreen_username">Username:</string>
```

Fonte: elaborado pelo autor.

Trocando o valor para "*yes*", compilando com o *apktool* o projeto da aplicação com essa alteração no arquivo e instalando novamente no dispositivo, tem-se o aparecimento do botão "*Create User*", como mostram as figuras a seguir. A Figura 18 mostra a tela antes da alteração no código e a Figura 19 mostra após a alteração:

Figura 18 - Tela antes da alteração no aplicativo *InsecureBankv2*

Fonte: elaborado pelo autor.

Figura 19 - Tela depois da alteração no aplicativo *InsecureBankv2*

Fonte: elaborado pelo autor.

Esse problema afeta os requisitos de segurança do MASVS de nível L1 e R, respectivamente, denominados “MSTG-ARCH-1: Todos os componentes do aplicativo são identificados e reconhecidos como necessários” e “MSTG-RESILIENCE-3: O aplicativo

detecta e responde para a manipulação de executáveis e dados críticos do próprio aplicativo". O teste do MASTG para averiguar esse segundo requisito é denominado "Testando verificações de integridade de arquivo (MSTG-RESILIENCE-3)"[29]. Para o primeiro requisito, ainda não há um teste registrado no guia MASTG direcionado a ele.

É indicado que as permissões e funções que requerem um certo nível de autorização utilizem as informações para tal diretamente do sistema *backend*, como o servidor responsável pela autenticação dos usuários. É importante também verificar se todos os endpoints de API e funcionalidades necessárias estão disponíveis publicamente, além de que o aplicativo deve ser capaz de fazer verificação em tempo de execução para detectar que o código foi alterado a partir das informações que são conhecidas sobre a integridade da aplicação em tempo de compilação.

4.3.3 Funcionalidade estranha (Posição 10)

Utilizando a ferramenta MobSF para analisar o aplicativo, como também para decompilar a aplicação para obter seu código fonte *Java*, e acessando o arquivo de código fonte referente a "*LoginActivity*", observa-se o trecho de código referente ao procedimento de *login*, onde se identifica a utilização da *Activity* "*DoLogin*" para tratar do dados de *login* inseridos pelo usuário. A Figura 20, mais especificamente na linha indicada pela seta vermelha, a seguir mostra a análise:

Figura 20 - Trecho de código onde mostra a utilização de "*DoLogin*" do aplicativo Diva

```
protected void performlogin() {
    this.Username_Text = (EditText) findViewById(R.id.loginscreen_username);
    this.Password_Text = (EditText) findViewById(R.id.loginscreen_password);
    Intent i = new Intent(this, DoLogin.class);
    i.putExtra("passed_username", this.Username_Text.getText().toString());
    i.putExtra("passed_password", this.Password_Text.getText().toString());
    startActivity(i);
}
```

Fonte: elaborado pelo autor.

Conferindo o arquivo de código fonte de "*DoLogin*", é possível notar que há um usuário padrão denominado "*devadmin*", que tem um tratamento especial e é mostrado na

[29] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05j-Testing-Resiliency-Against-Reverse-Engineering/#testing-file-integrity-checks-mstg-resilience-3

Figura 21: não informando uma senha como, também, informando uma senha qualquer, o *login* será realizado com sucesso para esse usuário, indicado das linhas 19 a 22. Além disso, é utilizado um endpoint diferente para realizar o *login* deste usuário, indicado na linha 15.

Figura 21 - Trecho de código onde mostra o diferente tratamento para o usuário "*devadmin*" no aplicativo Diva

```
9  public void postData(String valueIWantToSend) throws ClientProtocolException, IOException, JSONException,
10 InvalidKeyException, NoSuchAlgorithmException, NoSuchPaddingException, InvalidAlgorithmParameterException,
11 IllegalBlockSizeException, BadPaddingException {
12         HttpResponse responseBody;
13         DefaultHttpClient defaultHttpClient = new DefaultHttpClient();
14         HttpPost httppost = new HttpPost(DoLogin.this.protocol + DoLogin.this.serverip + ":" + DoLogin.this.serverport + "/login");
15         HttpPost httppost2 = new HttpPost(DoLogin.this.protocol + DoLogin.this.serverip + ":" + DoLogin.this.serverport + "/devlogin");
16         List<NameValuePair> nameValuePairs = new ArrayList<>(2);
17         nameValuePairs.add(new BasicNameValuePair("username", DoLogin.this.username));
18         nameValuePairs.add(new BasicNameValuePair("password", DoLogin.this.password));
19         if (DoLogin.this.username.equals("devadmin")) {
20             httppost2.setEntity(new UrlEncodedFormEntity(nameValuePairs));
21             responseBody = defaultHttpClient.execute(httppost2);
22         } else {
23             httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
24             responseBody = defaultHttpClient.execute(httppost);
25         }
26         InputStream in = responseBody.getEntity().getContent();
27         DoLogin.this.result = convertStreamToString(in);
28         DoLogin.this.result = DoLogin.this.result.replace("\n", "");
29         if (DoLogin.this.result == null) {
30             return;
31         }
32         if (DoLogin.this.result.indexOf("Correct Credentials") != -1) {
33             Log.d("Successful Login:", " account=" + DoLogin.this.username + ":" + DoLogin.this.password);
34             saveCreds(DoLogin.this.username, DoLogin.this.password);
35             trackUserLogins();
```

Fonte: elaborado pelo autor.

Esse problema afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-ARCH-1: Todos os componentes do aplicativo são identificados e reconhecidos como necessários" e ainda não há um teste registrado no MASTG direcionado a esse requisito. Recomenda-se verificar se todos os endpoints de API estão disponíveis publicamente, retirar códigos de teste antes da compilação da fase de produção e *log* de mensagens para garantir que nenhuma informação descritiva do *backend* fique registrada.

4.4 *InsecureShop*

É um aplicativo projetado em *Kotlin* especificamente para ser vulnerável, contendo cerca de 20 vulnerabilidades.

4.4.1 Comunicação insegura (Posição 3)

Utilizando a ferramenta MobSF (ver Figura 22), foi detectado que uma *WebView*, que faz parte da *Activity* "*WebViewActivity*", mencionada na Seção 2.4, foi implementada com a falta de verificação do certificado SSL, sem verificar erros e a validade do mesmo. Na

Figura 23, é possível ver o trecho do código onde a variável que representa o possível erro, "*SslError error*", e prossegue com a solicitação para o caso de o certificado ser válido, indicada pela chamada de função "*handler.proceed()*". Isso pode acarretar em aceitar comunicações com certificados defasados e o usuário ser suscetível ao ataque MITM (ataque do homem no meio), em que um atacante pode monitorar a comunicação e roubar dados confidenciais que possam ser transferidos na comunicação.

Pela Figura 22, é possível verificar que a ferramenta indica com o maior grau de risco (*high*) na métrica da ferramenta para a vulnerabilidade encontrada. Além disso, mais uma vez é citada uma referência a CWE, remetendo à vulnerabilidade "CWE-295[30]: Validação de certificado imprópria", ou seja, que o software não valida ou valida incorretamente um certificado.

Figura 22 - Detecção da ferramenta MobSF da falta de verificação de erros de certificados SSL na aplicação InsecureShop

4	Insecure WebView Implementation. WebView ignores SSL Certificate errors and accept any SSL Certificate. This application is vulnerable to MITM attacks	high	**CWE:** CWE-295: Improper Certificate Validation **OWASP Top 10:** M3: Insecure Communication **OWASP MASVS:** MSTG-NETWORK-3

Fonte: elaborado pelo autor.

Figura 23 - Trecho do código onde está a falta de verificação de erros de certificados SSL na aplicação *InsecureShop*

```
public final class CustomWebViewClient extends WebViewClient {
    @Override // android.webkit.WebViewClient
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
        if (handler != null) {
            handler.proceed();
        }
    }
}
```

Fonte: elaborado pelo autor.

Pode-se ainda verificar esse problema na aplicação InsecureShop utilizando uma análise dinâmica com as ferramentas *Burp Suite* e adb.

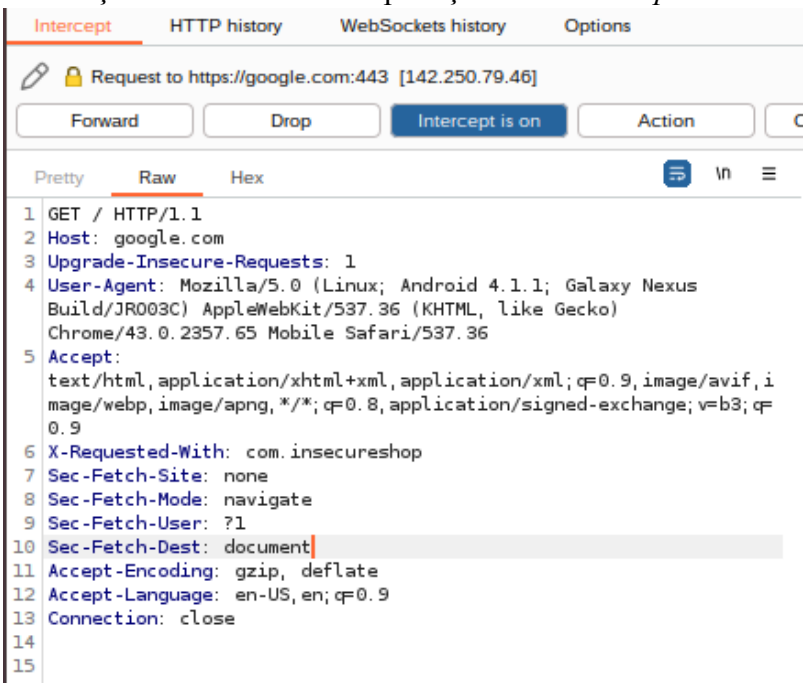
[30] CWE-295: https://cwe.mitre.org/data/definitions/295.html

Primeiramente, o *Burp Suite* foi utilizado para interceptar a comunicação que utiliza o protocolo HTTP. Nesse caso em específico, diferente da recomendação de instalar no dispositivo um certificado confiável do *Burp Suite* para interceptar comunicações com HTTPS, será utilizado um dispositivo sem certificado instalado para averiguar a falta de validação do certificado na aplicação. Normalmente, se o certificado do *Burp Suite* não estiver instalado no dispositivo, a ferramenta não conseguirá interceptar tráfego HTTPS das aplicações, pois elas, corretamente, estão fazendo a verificação da utilização de um certificado. Se o certificado não estiver instalado e mesmo assim for possível observar comunicação HTTPS, indica que a aplicação em questão está com uma falha nessa verificação, e é o que será testado na *InsecureShop*.

Com o adb, foi utilizado o comando "*shell*" para se ter acesso ao dispositivo em que a aplicação está instalada e com o acesso, a *WebView* da aplicação que contém a vulnerabilidade é invocada por linha de comando, a partir da utilização do comando "*shell*" citado, para acessar a url "*google.com*", que é um host que utiliza HTTPS para comunicação.

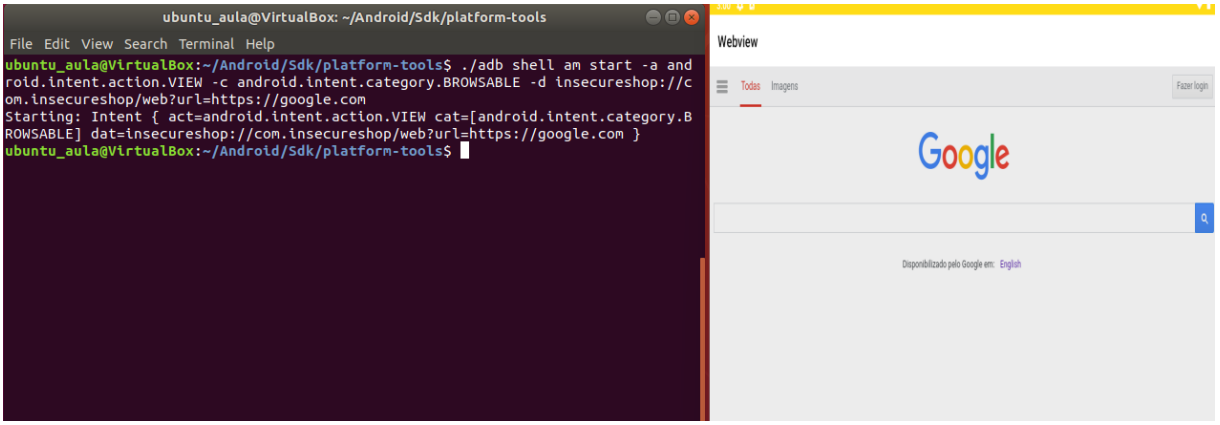
Esse procedimento é resumido nas imagens a seguir, em que a Figura 24 mostra uma comunicação interceptada pelo *Burp Suite* que utiliza HTTPS invocada a partir do aplicativo InsecureShop, indicado na linha 6, e que é acessada por meio da linha de comando, por meio do adb, que invoca a WebView e não pela aplicação diretamente acessada pelo usuário, como mostra a Figura 25. Por ela, é vista a invocação da WebView e a aplicação sendo executada acessando a url indicada.

Figura 24 - Visualização do *Burp Suite* que exemplifica o problema de validação de certificado na aplicação *InsecureShop*



Fonte: elaborado pelo autor.

Figura 25 - Acessando "*google.com*" a partir da aplicação *InsecureShop*



Fonte: elaborado pelo autor.

Esse problema afeta o requisito de segurança do MASVS de nível L1 denominado "MSTG-NETWORK-3: O aplicativo verifica o certificado X.509 do terminal remoto quando o canal seguro é estabelecido. Apenas certificados assinados por uma CA confiável são aceitos". O teste do MASTG para averiguar esse requisito é denominado "Testando para verificação de identificação do endpoint (MSTG-NETWORK-3)"[31]. A verificação de certificados SSL deve averiguar se a CA que assinou o certificado é confiável, se o certificado

[31] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05g-Testing-Network-Communication/#testing-endpoint-identify-verification-mstg-network-3

ainda não expirou e se ele é auto assinado[32]. Uma maneira simples é utilizar a interface *TrustManager*, que contém os meios necessários para fazer as verificações de validade de um certificado.

4.5 *VulnApp*

É um aplicativo desenvolvido especificamente para explorar a adulteração de código.

4.5.1 Adulteração de código (Posição 8)

Analisando a funcionalidade de acesso simplificada do aplicativo, observa-se que é necessário digitar uma senha que, a princípio, o usuário não sabe. A interface referente a essa descrição com uma senha incorreta digitada pode ser vista na Figura 26 a seguir:

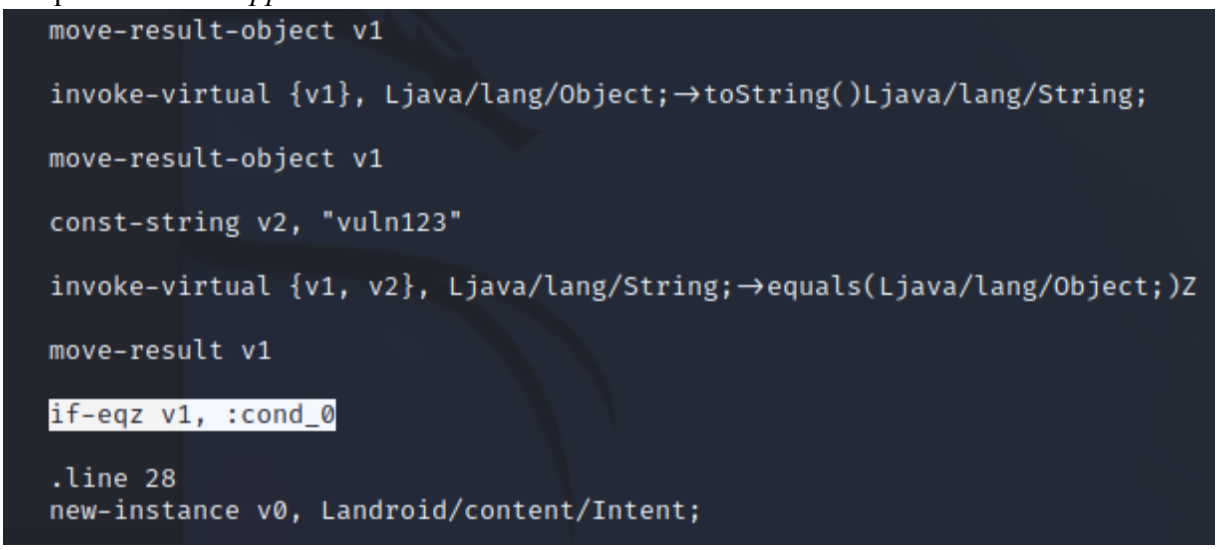
Figura 26 - Funcionalidade de acesso do aplicativo *VulnApp*

Fonte: elaborado pelo autor.

[32] Segurança com HTTPS e SSL: https://developer.android.com/training/articles/security-ssl

Utilizando a ferramenta apktool para decompilar a apk referente ao aplicativo *VulnApp*, e analisando o código *smali*, mais precisamente no arquivo localizado em "*smali/openssecurity/vulnapp/MainActivity$1.smali*", é possível perceber, com um certo esforço, que, no momento em que se clica no botão, uma comparação entre o dado inserido e a *string* "*vuln123*" é feita, dando indício de que essa é a senha correta. O trecho de código apresentado na Figura 27 mostra isso:

Figura 27 - Trecho de código onde mostra a comparação com a *string* "*vuln123*" no aplicativo *VulnApp*

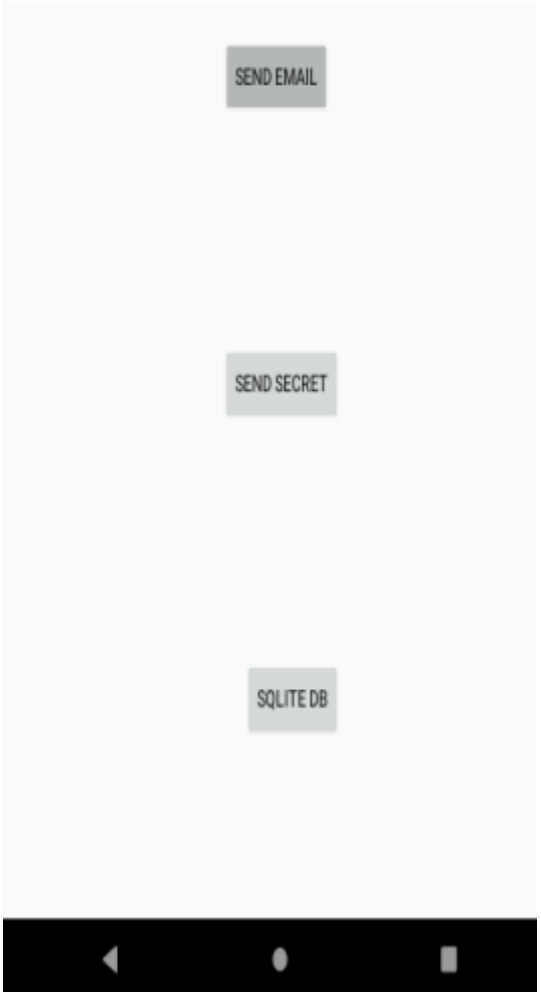


Fonte: elaborado pelo autor.

A função "*if-eqz*" está comparando 2 variáveis, a que contém o dado do usuário e a que contém a string da senha correta, e retorna verdadeiro se forem iguais e falso se forem diferentes. A função "*if-nez*" faz o oposto da função "*if-eqz*": retorna falso se forem iguais e verdadeiro se forem diferentes.

Fazendo essa modificação no código *smali* (trocar a função "*if-eqz*" por "*if-nez*"), recompilando a estrutura do aplicativo com o *apktool*, e instalando o novo aplicativo, percebe-se que digitando qualquer coisa diferente de "vuln123", o procedimento para acessar a aplicação é feito com sucesso, como mostra a Figura 28 a seguir:

Figura 28 - Funcionalidade de acesso após a manipulação no código do aplicativo *VulnApp*



Fonte: elaborado pelo autor.

Esse problema afeta o requisito de segurança do MASVS de nível R denominado "MSTG-RESILIENCE-3: O aplicativo detecta e responde para a manipulação de executáveis e dados críticos do próprio aplicativo". O teste do MASTG para averiguar esse requisito é denominado "Testando verificações de integridade de arquivo (MSTG-RESILIENCE-3)"[33]. Para se proteger dessa vulnerabilidade é indicado que o aplicativo seja capaz de fazer verificação em tempo de execução para detectar que o código foi alterado. Essa verificação acontece com a aplicação "sabendo" de informações em tempo de compilação sobre a integridade da própria aplicação, por meio de verificação *Cyclic Redundancy Check* (CRC) no *bytecode* do aplicativo, bibliotecas nativas e arquivos de dados importantes. CRC é um código de detecção de erro que serve para analisar a integridade de dados.

[33] Teste do MASTG disponível em: https://mas.owasp.org/MASTG/Android/0x05j-Testing-Resiliency-Against-Reverse-Engineering/#testing-file-integrity-checks-mstg-resilience-3

5 CONCLUSÃO

Na Seção 4, onde foram apresentados os testes de segurança, mostrou que as vulnerabilidades identificadas podem ser bastante perigosas e que não devem ser ignoradas. Com isso, é confirmado que uma metodologia de desenvolvimento de software incorporando os testes de segurança em aplicações *Mobile*, principalmente para plataformas *Android*, foco desse trabalho, se faz necessária. A explicação para isso é baseada no número de pessoas utilizando *smartphones*, mencionado na introdução, ser bastante considerável, e a existência de aplicações vulneráveis instaladas neles pode gerar prejuízos em escala mundial tanto para o usuário quanto para a equipe que detém os direitos das aplicações. Se essas aplicações tratam de dados sensíveis de usuários, o prejuízo pode ser ainda maior.

As más práticas de programação, pelo que foi mostrado nos testes, são as principais vilãs no contexto das vulnerabilidades enumeradas pelo OWASP *Mobile Top Ten*, tanto que as recomendações para se contornar esses problemas, em maioria, envolviam seguir os padrões atestados pela linguagem ou *framework* utilizada. A vulnerabilidade de uso inadequado da plataforma, que está diretamente ligada com o mal uso da linguagem de programação, é a primeira da lista, mostrando a necessidade de se ter atenção a esse aspecto.

Além disso, uma padronização da maneira como atestar a existência ou não de vulnerabilidades é importante para se ter resultados convincentes. Foi mostrado, para cada vulnerabilidade, que um ou mais requisitos de segurança elencados pelo MASVS não estavam sendo seguidos. Com a utilização dele na fase de análise de requisitos e riscos do projeto, e a utilização das ferramentas indicadas pelo MASTG, bem como a maneira de realizar os testes, são instrumentos bastante completos e confiáveis, que trazem uma maior garantia de que o número de vulnerabilidades críticas no projeto será minimizada.

Se tratando dos objetivos definidos na introdução, pode considerar que o objetivo geral foi parcialmente alcançado, pelo fato de não haver, ainda, um conjunto de procedimentos de testes registrados no MASTG que contemplem todos os requisitos de segurança elencados pelo MASVS. Isso foi observado nos casos das Seções 4.6 e 4.10. Mesmo com esses testes ainda pendentes de registros, ainda é notório o quão benéfico o MASTG é para servir como direcionamento na execução de testes de segurança em aplicativos. Fora isso, os objetivos específicos definidos na introdução foram totalmente alcançados, ressaltando as recomendações de solução dos problemas ou de como evitá-los.

Para trabalhos futuros, pode-se citar a tentativa de formular procedimentos de teste para os requisitos do MASVS que ainda não foram cobertos pelos testes do MASTG.

REFERÊNCIAS

Strategy Analytics. **Strategy Analytics: Half the World Owns a Smartphone**, 2021. Disponível em: https://news.strategyanalytics.com/press-releases/press-release-details/2021/Strategy-Analytics-Half-the-World-Owns-a-Smartphone/default.aspx

PELTIER, T. R. **Information security risk analysis**, 3.ed. Auerbach Publication, 2010. 331 p. v. 1

OWASP. **OWASP Mobile Application Security Verification Standard (MASVS)**. Disponível em: https://mas.owasp.org/MASVS/

OWASP. **OWASP Mobile Application Security Testing Guide (MASTG)**. Disponível em: https://mas.owasp.org/MASTG/

Tung, Y. H.; Lo, S. C.; Shih, J. F.; Lin, H. F. (2016, Outubro). An integrated security testing framework for secure software development life cycle. In 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS) (pp. 1-4). IEEE.

Khari, M.; Kumar, P. (2016, Março). Embedding security in software development life cycle (sdlc). In 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 2182-2186). IEEE.

Junmei, W.; Chengkang, Y. (2021, Agosto). Automation Testing of software security Based on Burpsuite. In 2021 International Conference of Social Computing and Digital Economy (ICSCDE) (pp. 71-74). IEEE.

Izurieta, C.; Prouty, M. (2019, Maio). Leveraging secdevops to tackle the technical debt associated with cybersecurity attack tactics. In 2019 IEEE/ACM International Conference on Technical Debt (TechDebt) (pp. 33-37). IEEE.

Rangnau, T.; Buijtenen, R. V.; Fransen, F.; Turkmen, F. (2020, Outubro). Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC) (pp. 145-154). IEEE.

Sun, X.; Cheng, Y.; Qu, X.; Li, H. (2021, Junho). Design and Implementation of Security Test Pipeline based on DevSecOps. In 2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC) (Vol. 4, pp. 532-535). IEEE.

Chao, W.; Qun, L.; XiaoHu, W.; TianYu, R.; JiaHan, D.; GuangXin, G.; EnJie, S. (2020, Junho). An android application vulnerability mining method based on static and dynamic analysis. In 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC) (pp. 599-603). IEEE.

Li, L.; Bissyandé, T. F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Octeau, D.; Traon, L. (2017). Static analysis of android apps: A systematic literature review. Information and Software Technology, 88, 67-95.

Sarkar, A.; Goyal, A.; Hicks, D.; Sarkar, D.; Hazra, S. (2019, Dezembro). Android application development: a brief overview of android platforms and evolution of security systems. In 2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC) (pp. 73-79). IEEE.

Mateless, R.; Rejabek, D.; Margalit, O.; Moskovitch, R. (2020). Decompiled APK based malicious code classification. Future Generation Computer Systems, 110, 135-147.

Gonzalez, H.; Kadir, A. A.; Stakhanova, N.; Alzahrani, A. J.; Ghorbani, A. A. (2015, Agosto). Exploring reverse engineering symptoms in Android apps. In Proceedings of the Eighth European Workshop on System Security (pp. 1-7).

Ziadia, M.; Fattahi, J.; Mejri, M.; Pricop, E. (2020). Smali+: An operational semantics for low-level code generated from reverse engineering Android applications. Information, 11(3), 130.