

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

18. Informatika a výpočetní technika

Zranitelnosti mobilních aplikací

Michal Himer

Jihomoravský kraj

Židlochovice, 2023

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

18. Informatika a výpočetní technika

Zranitelnosti mobilních aplikací Vulnerabilities in mobile applications

Jméno: Michal Himer

Škola: Gymnázium Zidlochovice, příspěvková organizace

Kraj: Jihomoravský kraj

Konzultant: doc. Ing. Zdeněk Martinásek Ph.D.

Školní konzultant: Mgr. Dana Pauková

Židlochovice, 2023

Disclaimer

Tento dokument je zkrácená verze původní práce SOČ. Pro účely jejího užití jakožto zdroje informací uživatelům Kybernetické arény VUT z ní byla vyňata část pojednávající o tvorbě scénáře. Při plnění scénáře i mimo něj můžete tuto práci využít k doplnění informací o vybraných zranitelnostech v mobilních aplikacích a jejich vyzkoušení ve vlastním prostředí.

Poděkování

Chtěl bych poděkovat paní konzultantce Mgr. Daně Paukové za trpělivost, motivaci k práci a za cenné rady a připomínky ke psaní a formátování. Také bych chtěl poděkovat panu konzultantovi doc. Ing. Zdeňku Martináskovi Ph.D. za námět práce, odborné vedení a bližší přístup ke Kybernetické aréně VUT. Také mu děkuji, že mne přesvědčil ke psaní práce v programu \LaTeX , což mi usnadnilo práci v mnoha ohledech. Nakonec bych chtěl poděkovat i Ing. Tomáši Stodůlkovi za ochotu a pomoc při implementaci hry do Kybernetické arény VUT.

Anotace

Práce se zabývá zabezpečením mobilních aplikací v operačním systému Android. V prvních kapitolách se čtenář dozví o organizaci OWASP, bezpečnostních mechanismech systému Android a metodě penetračního testování. Dále se práce soustředí na tři vybrané zranitelnosti mobilních aplikací a detailně rozebírá příčiny jejich vzniku, způsob jejich zneužití a možnosti ochrany proti nim. V praktické části jsou tyto zranitelnosti demonstrovány ve virtuálním prostředí a je navržena výuková hra společně s prototypem aplikace, která bude sloužit k výuce o zranitelnostech v mobilních aplikacích zábavnou formou. Tato hra byla naimplementována do virtuálního prostředí kybernetické arény na Fakultě elektrotechniky a komunikačních technologií Vysokého učení technického v Brně.

Klíčová slova

Android, aplikace, zranitelnost, kyberbezpečnost.

Abstract

This thesis is about mobile application security in the Android operating system. In the first chapters, the reader will learn about the OWASP organization, Android security mechanisms and penetration testing method. Next, the thesis focuses on three selected mobile-devices vulnerabilities and discusses in detail the causes of vulnerabilities, how they are exploited and how to protect against them. In the practical part, those vulnerabilities are demonstrated in a virtual environment and a tutorial game is designed along with a prototype application to teach about the vulnerabilities in mobile applications in a fun way. This game is then implemented into a virtual cyber range environment at the Faculty of electrical engineering and communication at Brno university of technology.

Keywords

Android, application, vulnerability, cybersecurity.

OBSAH

Úvod	2
1 Organizace OWASP	4
2 Zabezpečení OS	5
2.1 Aplikační sandbox	5
2.2 Oprávnění aplikací	6
3 Penetrační testování	8
3.1 Průběh testování	8
3.2 Výsledky	11
4 Analýza zranitelností	12
4.1 Nezabezpečená úložiště	13
4.2 Exportovaná aktivita	19
4.3 Binární protekce	26
5 Demonstrace zranitelností	32
5.1 Příprava prostředí	33
5.2 Nezabezpečená úložiště	39
5.3 Exportovaná aktivita	42
5.4 Binární protekce	45
Závěr	49

ÚVOD

V již téměř čtvrtině jednadvacátého století se mobilní telefony staly nedílnou součástí životů naprosté většiny lidí na planetě. Chytrý mobilní telefon v roce 2022 vlastnilo přibližně 91 % světové populace [50]. Současně s tím se ve světě technologií začíná stále více a více řešit otázka kyberbezpečnosti, a jak bychom na ni jako lidstvo měli reagovat. Tato dvě horká témata se rychle a jistě spojují v jedno, a najednou je velmi důležité klást důraz na bezpečnost mobilních telefonů a aplikací na nich nainstalovaných. Žijeme dnes ve společnosti, kdy našim zařízením a jejich vývojářům svěřujeme nepřeborné množství jak veřejně známých informací, tak i citlivých a soukromých údajů. S touto důvěrou ale do zařízení vkládáme také velkou moc.

Technologie, zejména pak chytré telefony, ovládají dnešní svět. Každou druhou myčku nádobí spustíte přes váš mobilní telefon a stejně tak snadno z něj například odemknete své vozidlo. Již dnes existují autonomní auta, která jsou za pomoci množství senzorů a umělé inteligence schopny pasažéry převézt z jednoho místa na druhé. Dokonce i řízení dopravních letadel je z velké části přenecháváno autonomním systémům. S touto mocí udílenou zařízením ale přichází také velká zodpovědnost řádně je zabezpečovat. Zodpovědnost tak velká, jako touha některých členů naší společnosti získat nad těmito systémy kontrolu.

Kdyby někdo dokázal prolomit kritické bezpečnostní infrastruktury, mohlo by to mít nedozírné následky. Od publikování citlivých informací o známé celebritě nebo krádeže auta, přes způsobení autonehody až po pád letadla. Otázka kyberbezpečnosti je tedy palčivější než kdy dříve, a nutnost se jí zabývat se v budoucnosti bude velmi pravděpodobně bude zvyšovat. Je proto nezbytné dbát na dostatečné zabezpečení, skrze dobře provedená testování.

Každá produkční aplikace by měla před tím, než je vypuštěna na trh, projít testovacím procesem, který aplikaci detailně analyzuje a objeví co nejvíce zranitelností. To s sebou bohužel často nese nejen vysoké náklady, ale také snížení uživatelského komfortu. Z těchto důvodů firmy často přehlížejí nutnost zabezpečení jejich aplikací a věnují jim jen částečnou, nebo dokonce žádnou pozornost. V praxi je ale vždy důležité zvážit, které zranitelnosti by mohly být pro aplikaci, či pro vývojářské firmy kritické, a těmito zranitelnostem věnovat pozornost i za cenu vyšších nákladů. Zranitelnostmi jsou koneckonců ohroženi i uživatelé, a pokud je jim skrze zranitelnou aplikaci nějakým způsobem poškozeni, firma vyvíjející tuto aplikaci by měla převzít zodpovědnost.

Na druhou stranu bezpečnost nikdy není absolutní. Je nemožné vytvořit systém naprosto a zcela odolný vůči veškerým útokům. I při velké snaze vývojářů nakonec míra zabezpečení systému odpovídá pouze míře jejich znalostí a schopností. Pro jakéhokoliv útočníka s pokročilejšími dovednostmi nebude problém zabezpečení překonat. Aby byla zachována integrita a bezpečnost systémů, testeři musí být vždy o krok před hackery. Pokud nejsou, systémy, které denně využíváme, a kterým tolik věříme, se pro nás stávají hrozbou. A jelikož se možnosti využití mobilních telefonů v dnešním světě neustále zvětšují, zabezpečování by mělo začít právě u nich.

Cílem mé práce je obeznámit čtenáře srozumitelným způsobem se zranitelnostmi mobilních aplikací a bezpečností mobilních zařízení, případně v něm vzbudit o toto téma zájem. Tato práce také může výrazně pomoci vývojářům při zabezpečování jejich vlastních aplikací. V neposlední řadě je důležitým cílem vytvořit způsob, jakým se lze o tématu dozvědět hravou a zábavnou formou určenou pro studenty všech věkových kategorií. Součástí toho je i vytvoření demonstrační aplikace, která by byla nejen funkční, ale zároveň by na ní bylo možné zranitelnosti otestovat a ukázat jejich dopad v praxi. Kybernetická aréna VUT je k tomuto účelu vhodné prostředí. Jde o tzv. „cyber range“ platformu, ve které studenti trénují dovednosti týkající se kybernetické bezpečnosti formou „hry o vlajku“. Kybernetická aréna VUT zatím žádnou hru zaměřenou na testování mobilních aplikací neobsahuje, a já bych rád možnost takového způsobu učení nabídl i studentům, kteří se více soustředí na mobilní aplikace. Podat informace o tomto aktuálním tématu zábavným mi bylo při psaní velkou motivací.

Analýza jednotlivých zranitelností bude zahrnovat detailní informace o příčinách, možnostech zneužití a ochraně proti těmto zranitelnostem. Každá z nich pak bude demonstrována ve virtuálním prostředí za využití zranitelných aplikací. Budou také popsány veškeré postupy při tvorbě virtuálního prostředí a demonstraci zranitelností tak, aby byl uživatel schopen všechny úkony provést sám a zranitelnost si tak vyzkoušet na vlastní kůži. Celá práce bude doprovázena velkým množstvím obrázků, z nichž téměř všechny jsou autorské. Součástí práce je také elektronická příloha dostupná v repozitáři na platformě Github.com na adrese https://github.com/MichalHimer/SOC_22_23_Himer_Michal. Příloha obsahuje aplikační balíčky aplikací RootDetector a DataGuard, textový soubor links.txt s odkazy pro stažení nástrojů využívaných v práci a program mainactivityhook.js využívaný při demonstraci jedné ze zranitelností. Práce byla vysázena pomocí programu L^AT_EX.

Chtěl bych zdůraznit, že práce je určena výhradně ke vzdělávacím účelům. V žádném případě nepodporuje zneužívání informací obsažených v této práci za účelem poškození jiné osoby nebo za účelem vlastního zisku. Autor práce nenese žádnou zodpovědnost za chování lidí, kteří by práci zneužili k páchání jakékoliv trestné činnosti. Práce nikoho k ničemu podobnému nenabádá, každý je za své chování zodpovědný sám.

1 ORGANIZACE OWASP

V oblasti zabezpečení mobilních aplikací udělala velký krok kupředu nezisková organizace OWASP (The Open Web Application Security Project). Jedná se o projekt založený v roce 2001 Markem Curpheyem a jeho cílem je umožnit firmám vytvářet a udržovat bezpečné a důvěryhodné systémy [2]. V roce 2003 organizace poprvé představila seznam OWASP Top 10, což je žebříček pojednávající o nejběžnějších a nejzávažnějších zranitelnostech webových aplikací. Tento žebříček je průběžně aktualizován, poslední verze pochází z roku 2022 [22]. V roce 2012 pak poprvé publikovali žebříček Mobile Top 10, který se zaměřoval na nejčastější zranitelnosti mobilních aplikací [41]. V roce 2018 jej rozšířili o dokument MASTG (Mobile Application Security Testing Guide), který obsahuje požadavky na bezpečnost mobilních aplikací rozdělených dle kategorií (L1, L2, R), příručku k jejich otestování a také kontrolní seznam, podle kterého lze zkontrolovat, zda testovaná aplikace obsahuje závažné zranitelnosti [21].

Kategorie L1 a L2 určují, jak silná opatření aplikace vyžaduje. Požadavky kategorie L1 se vztahují na každou vyvíjenou aplikaci a jejich naplnění vyústí v bezpečnou aplikaci, která splňuje základní bezpečnostní standardy a neobsahuje běžné zranitelnosti. Do kategorie L2 spadají požadavky směřující na aplikace, které spravují vysoce citlivé informace (např. bankovní aplikace) a zahrnují dodatečné zabezpečení proti více složitým útokům. Kategorie R zahrnuje požadavky na ochranu aplikace před reverzním inženýrstvím, modováním, upravováním kódu apod. Tyto požadavky by měly naplňovat zejména aplikace, které jsou pro zabránění těmto technikám specificky navrženy [20].

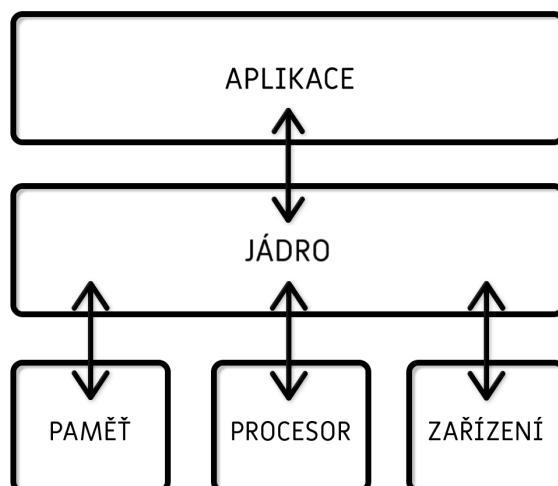
2 ZABEZPEČENÍ OPERAČNÍHO SYSTÉMU ANDROID

Základ zabezpečení mobilního zařízení leží již v zabezpečení samostatného operačního systému. Ten totiž je stavebním kamenem celého virtuálního prostředí, a nainstalované aplikace slouží již jen jako jakési rozšíření omezené funkcionality systému. Proto k tomu, abychom pochopili jak správně přistupovat k zabezpečení aplikací, je nutné nejprve pochopit, jaké možnosti nám operační systém dává, a jak do něj aplikace správně implementovat.

Operační systém Android byl postavený okolo linuxového jádra. Jádro (neboli také *kernel*) je v podstatě počítačový program, který tvoří skutečný základ systému a slouží jako prostředník komunikace mezi hardwarovými komponentami a jednotlivými aplikacemi (viz obr. 2.1). Výhodou tohoto jádra je, že je používáno již po velmi dlouhou dobu, a prošlo si velkou řadou testů. Můžeme jej tak považovat za důvěryhodné a bezpečné. To ale zdaleka není vše, čeho můžeme na Linuxu využít. Mimo mnoha jiných vlastností a bezpečnostních opatření, kterými Linux disponuje, byl Linux navržen jako víceuživatelský systém, primárně zaměřený na fungování na počítači. Jeho úkolem, jakožto systém pro více uživatelů, je izolovat každého uživatele od ostatních tak, aby žádný uživatel neměl přístup k datům jiného uživatele nebo nemohl využívat jeho hardwarové zdroje (například přidělenou operační paměť). Vývojáři Androidu však cíleně použili Linux jako jednouživatelský systém a princip více uživatelů využili na jiném místě. Právě tak vznikly dva základní a pro běžné uživatele neviditelné, ale přesto klíčové bezpečnostní mechanismy – aplikační sandbox a oprávnění aplikací [27].

2.1 APLIKAČNÍ SANDBOX

S víceuživatelskými systémy se již setkal snad každý, i když si to třeba neuvědomuje. Prakticky všechny běžné operační systémy, které běží na počítačích (Windows, MacOS, Linux), jsou víceuživatelské – v zařízení se nachází jeden nebo více uživatelů, přičemž každý z nich má v počítači své vlastní prostředí. Lze si jej představit jako plochu, kterou vidíte po přihlášení do zařízení. Máte z ní přístup ke svým vlastním souborům a programům nebo si ji můžete přizpůsobit dle svého. Změny, které provedete ve svém účtu, neovlivní účet nikoho jiného.



Obrázek 2.1: Znázornění funkce jádra v rámci systému [29].

Jak již bylo zmíněno, Android je systém pro pouze jednoho uživatele, a tak je zbytečné mít pro něj jakýkoliv mechanismus pro izolaci. Místo toho roli uživatelů v Androidu hrají jednotlivé aplikace. Každá běží jako samostatný uživatel ve svém vlastním sandboxu¹. Tím pádem je kompletně izolovaná od ostatních aplikací, nemá přístup ani k jejich datům, ani jejich hardwarovým zdrojům. To mimochodem výrazně zlepšuje tzv. multitasking zařízení, což je schopnost systému udržovat v běhu více aplikací najednou. Každá aplikace běží pod svým přiděleným unikátním uživatelským identifikátorem (tzv. UID – parametr, který je jedinečný pro každého uživatele a na jeho základě je lze mezi sebou rozpoznat) a je jí zároveň přidělen vlastní datový prostor, ve kterém může zapisovat nebo číst soubory pouze tato aplikace. Díky tomuto vcelku jednoduchému mechanismu je aplikace téměř kompletně izolována od ostatních a pokud by došlo k jejímu napadení a úspěšnému prolomení jejího zabezpečení, útočník by neměl mít možnost získat více dat nebo přístup k více možnostem, než kterými disponuje samotná aplikace [5].

2.2 SYSTÉM OPRÁVNĚNÍ APLIKACÍ

Aplikační sandboxing představuje skvělou bariéru mezi aplikací a zbytkem systému. Ta bohužel ale funguje oběma směry, a kromě toho, že nepouští žádná data dovnitř, tak žádná nepouští ani ven. Aplikace i samotní její vývojáři jsou tak omezeni na funkcionalitu, která jim umožňuje velmi malý přístup k systémovým funkcím a hardwaru. Vývojáři Androidu tedy hledali způsob, jak se tomuto omezení vyhnout a zároveň zachovat onu bezpečnostní bariéru. To se jim podařilo pomocí vytvoření systémů oprávnění.

Systém oprávnění si lze snadno představit jako malé tunely, které vedou z místnosti, kde je samotná aplikace, do různých částí systému a umožňují přístup k jeho funkcím.

¹Sandbox je označení pro prostředí, které běží v jiném vnějším prostředí samostatně v izolaci. Vše, co se stane uvnitř sandboxu, nijak neovlivní vnější prostředí.

Tím ale logicky dochází k narušení bezpečnostní bariéry a každé další oprávnění, pokud není naimplementováno v naprostém souladu se zabezpečením operačního systému, představuje riziko a další způsob, jak může útočník získat přístup i k jiným datům a funkcím, než poskytuje samotná aplikace. Každý vývojář by měl tak při implementaci jednotlivých oprávnění důkladně zvážit, zda je konkrétní oprávnění pro danou funkcionalitu opravdu nezbytné, a pokud ano, dbát na jeho správnou implementaci.

Systém oprávnění má ale i své přínosy. Všechna oprávnění musí být nějakým způsobem prezentovány uživateli v závislosti na jejich riziku. Zároveň může uživatel oprávnění vždy zamítnout, anebo zrušit již dříve udělená oprávnění. To přináší několik zásadních výhod:

- Kontrola uživatele nad chováním aplikace – uživatel si sám může vybrat, která oprávnění aplikaci udělí, a která ne. Pokud konkrétní oprávnění není zásadní pro běh aplikace, aplikace dále funguje jen s omezenou funkcionalitou.
- Transparentnost chování aplikace – uživatel si také může nechat zobrazit veškerá oprávnění, které má aplikace udělená a může si tak vytvořit vcelku jasnou představu o tom, jak aplikace s jednotlivými oprávněními zachází, a zda nepředstavují potenciální riziko.
- Minimalizace dat – aplikace jsou díky tomuto systému nuceny využívat jen oprávnění, které skutečně potřebují a je tak zmenšeno riziko neoprávněného přístupu k datům jiných aplikací skrze aplikaci napadenou [24].

Díky oprávněním aplikací již vývojář není tolik limitován systémem a má větší možnosti ve smyslu funkcionality aplikace. Ve spojení s aplikačním sandboxingem tvoří základ zabezpečení aplikace, které nám poskytuje systém samotný a pomáhají zmírnit dopady případného útoku. K němu by ale nemělo za žádnou cenu dojít, o to se však již musí postarat programátor při tvorbě konkrétní aplikace a také její uživatel.

3 PENETRAČNÍ TESTOVÁNÍ V OPERAČNÍM SYSTÉMU ANDROID

Penetrační testování (zkráceně *pentesting*) je metoda zhodnocení bezpečnosti systémů stejnými metodami, které používá útočník. Jeho cílem je objevit slabiny dříve, než je objeví útočníci, a opravit je. Zpravidla se provádí ještě před vydáním aplikace a je součástí tzv. *Software Development Life Cycle (SDLC)* – životního cyklu softwarového vývoje [44].

Člověk, který testování provádí se nazývá *pentester*, a může se jednat jak o externího člověka mimo systém (např. vývojářskou firmu) s malým množstvím dostupných informací, tak i o někoho zevnitř, kdo o vyvíjeném softwaru ví téměř všechno. V každém případě je velmi důležité, aby pentester měl povolení testování provést. Pokud jej má a dodržuje stanovená hranice a pravidla, jde o takzvaný *white-hat hacking*, který je zcela legální a často i finančně ohodnocený. Oproti tomu stojí *black-hat hacking*, kdy tester (v tomto případě jej lze označit za skutečného útočníka) se snaží překonat zabezpečení neoprávněně a s nečistým úmyslem (tedy za účelem nějakého osobního zisku, například krádeží dat, vydíráním nebo přeprodáváním informací o chybě dalším útočníkům). Mezi těmito dvěma póly se nachází ještě tzv. *grey-hat hacking*, kdy tester narušuje bezpečnost objektu neoprávněně, avšak jeho cílem není objekt zdiskreditovat, ale podat mu hlášení o chybě a zamezit tak útokům dalších lidí. V některých případech takovým lidem společnosti nabízejí finanční odměnu, není to ale jejich povinnost vzhledem k faktu, že tester k činnosti neměl povolení [28].

3.1 PRŮBĚH TESTOVÁNÍ

Při oficiálním externím testování, které poskytuje společnost nebo osoba třetí strany, můžeme testovací proces rozdělit na celkem pět fází:

1. vymezení rozsahu testování,
2. sběr a analýza informací,
3. využití zranitelností (exploitace),
4. závěrečná zpráva [28].

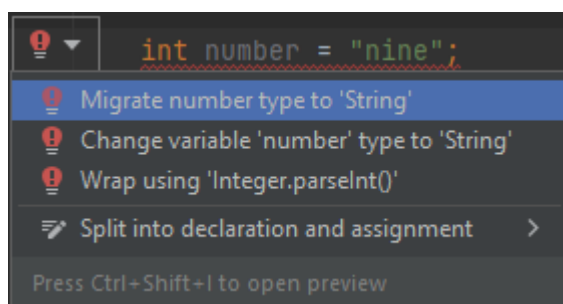
Toto je oficiální byznysový model, kterým se řídí zejména lidé, kteří si pentestingem vydělávají. Pro zaměření této práce jsou však podstatné pouze body 2 a 3, které se zaměřují na samotný průběh testování. Za objekt testování můžeme považovat mobilní

aplikaci. Vezmeme-li tedy v úvahu tyto úpravy, můžeme říci, že se proces penetračního testování skládá ze dvou částí: **Sběr a analýza informací** a **exploitace**.

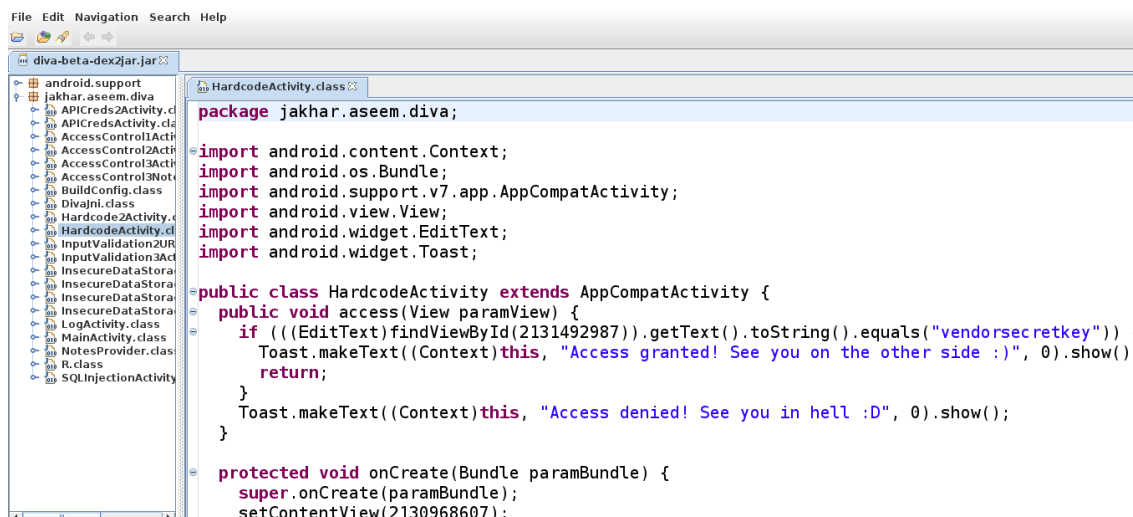
3.1.1 Sběr a analýza informací

Cíl sběru informací a jejich analýzy je vcelku jednoduchý: naším úkolem je zjistit, zda je v aplikaci nějaká zranitelnost a pokud ano, kde se nachází a jak ji můžeme využít. K tomuto úkolu se dá přistoupit mnoha způsoby, obecně je ale lze rozdělit na statickou analýzu a dynamickou analýzu. Statická analýza je ve své podstatě prozkoumávání aplikace do hloubky bez jejího vlastního spuštění. Může fungovat na principu kontroly kvality kódu, nebo ji můžeme provést zjišťováním informací o vývojáři, verzích aplikace apod. Jejím cílem je shromáždit o aplikaci co nejvíce informací. Eventuálně můžeme i prozkoumat kód, zjistit jak aplikace funguje a zda například v kódu nejsou přímo vepsány citlivé údaje. Ke zkoumání kódu se zpravidla využívá automatizovaných nástrojů. Z pohledu vývojáře, který testuje svoji vlastní aplikaci, se může jednat například o tzv. *linter*, který programátorovi zvýrazní chyby v kódu nebo upozorní na nevhodná řešení konkrétních problémů (viz obr. 3.1) [1]. Linter bývá běžně obsažen v softwaru vývojového prostředí, nebo si jej lze doinstalovat. Kromě linteru existují další podobné nástroje, jako je například PMD (Programming Mistake Detector), který kontroluje, zda v kódu nejsou nějaké nevyužité proměnné, prázdné bloky pro vypořádávání se s chybami, aj. [25]. Pro externí penetrační testery je vhodný třeba nástroj dex2jar, který dokáže převést zkomprimovaný `.apk` soubor na čitelný kód (viz obr. 3.2).

alicm!



Obrázek 3.1: Ukázka využití linteru ve vývojovém prostředí Android studio; poukazuje na chybu špatného datového typu a navrhně její řešení [29].



Obrázek 3.2: Ukázka zdrojového kódu dekompilevaného z .apk souboru pomocí nástroje dex2jar a zobrazeného v rozhraní nástroje jd-gui [29].

Proti statické analýze stojí analýza dynamická, která probíhá za běhu aplikace. Zaměřuje se na chování aplikace, jestli nenastávají nějaké chyby, které by vyústily v pád aplikace, či neunikají data a také zkoumá jak na případné chyby reaguje. Také je možné díky různým nástrojům jako je například Frida nebo Drozer vkládat do aplikace vlastní kód a měnit její chování. Dobře zabezpečená aplikace by měla být vůči všem takovým možnostem imunní a disponovat protiopatřeními. Dynamická analýza je mnohem účinnější, pokud je provedena až po statické analýze, poněvadž tehdy už tester ví, jak aplikace funguje a na jaká místa se má zaměřit. Jednoduše by se dalo říci, že aplikaci „pitváme“ tak dlouho, dokud nenarazíme na nějakou zranitelnost. Pokud se tak stane a my o ní máme dostatek informací, můžeme naplánovat útok a postoupit k druhé části, a to její exploitaci.

3.1.2 Využití zranitelností

Exploitace je část, kde využijeme získané informace o zranitelnosti a využijeme jich k získání přístupu k zařízení prolomením zabezpečení aplikace, či změnou jejího chování v náš prospěch. Jedná se o důležitou část testování, jelikož je potřeba ověřit, zda je nalezená zranitelnost skutečně relevantní a lze díky ní získat citlivé informace, ale také k jejímu opravení je často důležité plné pochopení jejího fungování. Také je třeba se podívat na věc z druhé strany – strany útočníka. Ten se snaží prolomit zabezpečení s cílem získat co nejvíce užitečných informací z cílené aplikace. Ať už jsou v jakékoliv podobě (může se jednat o hesla, fotografie nebo výpisy z bankovníctví), ve výsledku vlastně nejde o nic více, než je získání informací, které by neměly vyjít na povrch. To je opět důležité při hodnocení závažnosti zranitelnosti. Pokud jediný výstup, který její exploitace může nabídnout je například verze aplikace nebo operačního systému, zpravidla se jí nemusíme zabývat, protože se nejedná o nic, co by útočník mohl zneužít proti nám (výjimkou je, pokud je na zařízení nainstalovaná stará či nezabezpečená verze např. operačního systému, kterou útočník zná a může na základě této informace provádět další útoky).

Průběh exploitace výrazně závisí na povaze zranitelnosti, a je těžké jej charakterizovat obecně. Dá se ale sledovat jakási linka postupu, jak tato fáze probíhá. Uvažujeme, že tester již provedl dostatečnou analýzu a má potřebné množství informací k provedení útoku. Aby mohl zahájit útok, nejprve musí nějakým způsobem získat přístup k zařízení, na kterém je cílená aplikace nainstalována. Buď ho musí mít u sebe fyzicky (nejjednodušší je stáhnout si aplikaci na svoje vlastní mobilní zařízení či virtuální emulátor), anebo k němu získat přístup vzdáleně. Poté se k němu připojí, nejčastěji přes příkaz `adb connect`, využívající tzv. *Android debug bridge*, který umožňuje na zařízení spouštět příkazy. Poté může tester spustit aplikace a správnou volbou příkazů a nástrojů postupovat dle svého připraveného plánu. Mohou mu k tomu dopomoci tzv. *exploity*, což jsou programy vytvořené na míru známým zranitelnostem, aby skrz ně prolomily zabezpečení. Tyto exploity jsou volně dostupné na internetu, například na webu *www.exploit-db.com* [18]. Pokud tester nalezne dostatečně vhodný exploit, může jej použít jak je, anebo si jej dopracovat tak, aby vyhovoval jeho potřebám. Pokud se nám po provedení plánu podaří dosáhnout cíle, můžeme fázi pokládat za úspěšnou. Pokud ne, můžeme zranitelnost přezkoumat znovu a rozhodnout se pro provedení dalšího pokusu exploitace či uzavření zranitelnosti jako irelevantní. Naším cílem je mít aplikaci zabezpečenou co nejvíce, takže prohlášení zranitelnosti jako irelevantní musí být dostatečně podloženo.

3.2 VÝSLEDKY PENETRAČNÍHO TESTOVÁNÍ

Pokud byl pentesting proveden správně a byla dodržována všechna pravidla, měli bychom mít jasnou představu o zabezpečení aplikace. Víme, kde se nachází jaká zranitelnost, jak funguje a co ji způsobuje. Pokud jsme testovali vlastní aplikaci, můžeme ji na základě výsledků opravit a znovu otestovat. Pakliže byl test proveden externě, pod proces testování spadá ještě tzv. report, tj. zpráva o nalezených zranitelnostech seřazených podle závažnosti. Tester by měl podrobně popsat postup testování, detaily jednotlivých zranitelností, jejich dopady a návrh jejich řešení. Tato zpráva by měla být poté dodána zadavateli a ten by na jejím základě měl aplikaci zabezpečit.

Při ochraně proti nalezeným zranitelnostem je ale nutné myslet na to, že ne každá zranitelnost musí být nutně závažný problém, zatímco zabezpečení proti každé slabíně stojí čas a prostředky. Ideálně by aplikace měla být zabezpečena proti všem druhům útoků na všechna místa, ovšem reálně je velmi těžké se takovému stavu přiblížit. Proto je nutné zvažovat závažnost každé zranitelnosti (při externím testování by měla být uvedena v reportu) a nutnost její opravy. Případně se vývojář může rozhodnout opravit ty nejzásadnější zranitelnosti, aby aplikace byla relativně bezpečná a zároveň funkční a drobnější zranitelnosti může opravit v pozdějších verzích.

4 ANALÝZA VYBRANÝCH ZRANITELNOSTÍ

Při zabezpečování aplikace může dojít k nepřehlednému množství chyb na různých místech, které vyústí v různé zranitelnosti. Mohou být způsobeny mnoha faktory: špatně napsaný kód, který v určitém případě selže, nevyužití bezpečnostních mechanismů operačního systému, nebo například nedostatečná míra kryptografie (šifrování). Průběh vývoje aplikace je poměrně složitý a je velmi pravděpodobné, že někde nastane nějaká chyba. Ne všechny zranitelnosti jsou vážné, některé nemají prakticky žádný dopad na uživatele, ani na aplikaci, avšak některé mohou zásadně ovlivnit obojí. Pro mou práci jsem si vybral celkem tři zranitelnosti:

1. zranitelnost nezabezpečených datových úložišť,
2. zranitelnost exportované aktivity,
3. absence binární protekce.

Tyto zranitelnosti jsem podrobně popsal z mnoha úhlů v následujících sekcích.

4.1 NEZABEZPEČENÁ DATOVÁ ÚLOŽIŠTĚ

Zranitelnost nezabezpečených datových úložišť je jakákoliv zranitelnost, při které jsou citlivé informace uloženy jako prostý text na místech globálně přístupných celému systému. Obecně za citlivé informace považujeme veškeré personální informace, na jejichž základě je útočník schopný způsobit uživateli škodu anebo narušit jeho soukromí. Taková data mohou být například bankovní údaje, kreditní karty anebo přihlašovací údaje. Nebezpečná místa pro ukládání takových dat mohou být soubory s logy, XML soubory, SQL databáze nebo externí úložiště (SD karta) [16]. Většinou ale tato zranitelnost nemůže být zneužitá sama o sobě, je k tomu potřeba nějaké další chyby v zabezpečení.

4.1.1 Příčiny zranitelnosti

Zranitelnost nezabezpečeného datového úložiště je spíše kategorií různých zranitelností, které vedou k neúmyslnému úniku dat. Jejich konkrétních příčin může být proto mnoho. Obecně lze říci, že tato zranitelnost vzniká zpravidla ve chvíli, kdy vývojáři nepočítají s cizím přístupem k datům aplikace, ať už ze strany útočníka nebo malwaru¹. Pokud je taková možnost nenapadne, pravděpodobně se nebudou příliš zabírat tím, kam a jak data aplikace uloží. Uchýlí se tedy k běžným postupům, jako je ukládání dat například do sdílených preferencí, anebo do již výše zmíněných volně přístupných databází anebo XML souborů. V takovém postupu chyby nastávají hned na několika místech. Především by každý vývojář měl myslet na bezpečnost své aplikace a jejích uživatelů a to nehledě na její povahu. Mimo jiné by se měl tedy zamyslet nad všemi daty, které ukládá, zhodnotit, jak moc jsou citlivá a podle toho s nimi nakládat [16].

Konkrétní příčina vzniku zranitelnosti může být například spoléhání se na zabezpečení, které lze snadno překonat. Například díky aplikačnímu sandboxu Androidu jsou veškerá data privatizována pouze pro danou aplikaci, neurčí-li vývojář jinak, a proto lze ukládat data v podobě prostého textu v souborech aplikace, aniž by hrozil jejich únik. Aplikační sandbox ale bohužel představuje pouze základní úroveň zabezpečení. Pokud aplikace obsahuje zranitelnost, která by umožnila útočníkovi spouštět na zařízení vlastní kód anebo procházet soubory aplikace, získá tak přístup do jejího prostředí a aplikační sandbox již netvoří překážku. Druhý způsob, jak bariéru sandboxu překonat, je spustit zařízení s administrátorskými právy, tedy provést tzv. *root* zařízení. Takové zařízení má pak přístup k veškerým datům v něm uloženým a tedy procházet například sdílené preference anebo jiné soubory [16].

Další příčinou mohou být špatně nastavené parametry datových úložišť. Například ve

¹Malware, zkratka pro „škodlivý software“, označuje jakýkoli škodlivý software vyvinutý kyberzločinci (často označovanými jako *hackeři*) za účelem krádeže dat a poškození nebo zničení počítačů a počítačových systémů. Příklady běžného malwaru jsou viry, červi, trojské viry, spyware, adware a ransomware [36]

starších verzích Androidu bylo možné soubor uložit s parametrem `MODE_WORLD_READABLE`, nebo `MODE_WORLD_WRITEABLE`. Toto je parametr říkající, že soubor je možné číst či do něj zapisovat v rámci celého systému. Umožňuje tedy přístup k souboru všem ostatním aplikacím v zařízení. Tato funkce měla usnadnit vývojářům sdílení dat s jinými aplikacemi, pro vysoké bezpečnostní riziko však tato možnost byla v Androidu 4.2 (API úroveň 17) prohlášena za zastaralou a přestala být podporována. Od verze Android 7.0 (API úroveň 24) její použití vyústí v chybovou hlášku [10]. Další z parametrů, který může způsobit neúmyslný únik dat je parametr `android:allowBackup`. Ten je definován v souboru `AndroidManifest.xml`, který je základem každé Android aplikace a ve výchozím stavu je nastaven na hodnotu `true`. Znamená, že aplikace automaticky zálohuje data na cloudovém úložišti a slouží k jejich obnovení v případě reinstalace aplikace. Pokud tak například vaše aplikace uložila nějaká data na jednom zařízení a vy ji poté nainstalujete do druhého, měla by se nainstalovat již s daty z jiného zařízení. Proces obnovení ale může být vyvolán manuálně z ovládací konzole, a vrácená data dostane útočník přímo do rukou. Je proto vhodné zvážit, zda je automatická záloha dat potřebná a jak velké riziko představuje. Obecně se ale doporučuje mít tento parametr nastavený na hodnotu `false`, tedy mít zálohu vypnutou [10, 7].

Chyba také nemusí být v každém případě přímo na straně vývojáře. Může se třeba stát, že nějaká z použitých knihoven obsahuje bezpečnostní slabinu, díky které je útočník schopen například spouštět kód, získat šifrovací klíč apod. Osoba zodpovědná za výběr knihoven je nicméně stále vývojář a měl by používat pouze ověřené knihovny. V neposlední řadě je častou příčinou zranitelnosti nedostatečné šifrování. Ačkoliv toto je zranitelnost spadající do kategorie nedostatečné kryptografie, která je rovněž obsažena v seznamu Mobile top 10 organizace OWASP, stále jde o velmi běžnou příčinu neúmyslného úniku dat. Dešifrovat získaná data totiž pro útočníka tvoří mnohem větší překážku než jejich samotné získání a silná kryptografie může zabránit nezkušenému hackerovi data zužítkovat. Oproti tomu použití žádných nebo slabých šifrovacích knihoven útočníkovi umožní bez problémů soubory dešifrovat a číst jejich obsah.

4.1.2 Prevalence a zjistitelnost

Co se prevalence (běžnosti výskytu) týče, zranitelnost nezabezpečených datových úložišť je jednou z nejběžnějších zranitelností v mobilních aplikacích. Také proto se umístila na druhém místě v žebříčku zranitelností Mobile Top 10 od organizace OWASP [16]. Ten je řazený jak podle běžnosti výskytu, tak podle vážnosti dopadu. Výskyt zranitelnosti je zde označený jako běžný a dopad vážný. Konkrétní čísla se velmi liší v závislosti na zdroji a datu, například podle článku stránky *positive technologies* [48] z roku 2019 byla tato zranitelnost objevená v až 76 % testovaných aplikací. Jiné průzkumy zase odhadují okolo 40 %, stále se však jedná o velmi vysoké číslo. Obtížnost zjistitelnosti takové zranitelnosti závisí na míře šifrování aplikace, nicméně organizace OWASP ji v MT10 žebříčku charakterizovala jako průměrnou [16].

K jejímu objevení je potřeba zjistit, kam se jaké informace ukládají. Preventivně je dobré prozkoumat nejčastější používaná úložiště, jako jsou sdílené preference a databáze, pokud v rámci aplikace existují. Další užitečné informace mohou být také zapsány v souborech s logy. Pokud se nám podaří přečíst jejich obsah a naleznout citlivé informace, máme vyhráno. Může se ale také stát že soubory budou zašifrované, nebo jejich obsah nebude nijak hodnotný. Pak musíme hledat dále anebo data dešifrovat, což s sebou přináší značné komplikace.

Nejefektivnější způsob, jak zjistit, kam jsou data aplikace ukládána, je prozkoumat její zdrojový kód. Ten, pokud jej nemáme k dispozici, můžeme získat použitím dekompilačních nástrojů, jako je například JADX či již zmíněný dex2jar [45, 46]. Pokud vývojář nepoužil příliš silnou obfuskaci (viz sekce 4.1.5) a kód je čitelný, můžeme z něj vyčíst, kam jsou jaká data uložena. Pokud je kód obfuskován, získat z něj potřebné informace se stane velmi obtížným úkolem. Existují ale i tzv. deobfuskační nástroje, které mohou, ale také nemusí s tímto problémem pomoci.

Pokud po důkladném prozkoumání aplikace dospějeme k názoru, že aplikace žádná citlivá data neukládá na snadno dostupných úložištích, ale například na externím serveru, můžeme se domnívat, že aplikace tuto zranitelnost neobsahuje. Pokud však pojmeme podezření, že nějakou takovou aktivitu provádí (například nalezením kódu podobným kódu na obrázku 4.1), můžeme tuto možnost blíže prozkoumat.

```
public void saveCredentials(View paramView) {  
    SharedPreferences.Editor editor = PreferenceManager.getDefaultSharedPreferences((Context) this).edit();  
    EditText editText1 = (EditText) findViewById(2131493000);  
    EditText editText2 = (EditText) findViewById(2131493001);  
    editor.putString("user", editText1.getText().toString());  
    editor.putString("password", editText2.getText().toString());  
    editor.commit();  
    Toast.makeText((Context) this, "3rd party credentials saved successfully!", 0).show();  
}
```

Obrázek 4.1: Ukázka kódu, který viditelně ukládá uživatelské jméno a heslo do sdílených preferencí. Kód pochází z aplikace DIVA (Damn Insecure and Vulnerable App), byl dekompilován pomocí nástroje dex2jar a zobrazen v rozhraní jd-gui [29].

4.1.3 Exploitate

Jestliže jsme našli potenciální úložiště citlivých dat (ve zmíněném příkladu jsou to sdílené preference), proces exploitate spočívá v podstatě pouze v získání přístupu k tomuto úložišti a přečtení jeho obsahu. Ovšem i na cestě k tomuto jednoduchému cíli se mohou objevit různé překážky. Úplně první, na kterou můžeme narazit je nemožnost procházet data aplikace kvůli nedostatečným oprávněním. To je způsobeno aplikačním sandboxem. Ten, jak již bylo zmíněno (viz. sekce 2.1), izoluje data aplikace od zbytku systému a ostatní uživatelé (aplikace či konzole) k nim nemají přístup. Aplikační sandbox však přestane fungovat v momentě, kdy zařízení běží s administrátorskými právy. Administrátorská práva lze snadno využít, má-li tester k dispozici zařízení, ke kterému má plný přístup (může se jednat o jeho vlastní fyzické zařízení, nebo o virtuální zařízení). Na cizím zařízení

se to však provádí složitě, což případným útočníkům komplikuje cestu zjistit konkrétní data od cizích lidí. Uvažujeme-li však, že je aplikace pouze testována a nikoliv napadána, tester si může vytvořit virtuální zařízení, které spustí s administrátorskými právy a bariéru aplikačního sandboxu tím zboří. Poté má již volný přístup ke všem lokálně uloženým datům aplikace.

Další problém může nastat, pokud tester otevře soubor např. sdílených preferencí a zjistí, že je nečitelný. (viz. obr. 4.2). V takovém případě vývojář nejspíše využil šifrování a soubor je pro přečtení nutné dešifrovat. K tomu již tester potřebuje znalosti používaných šifrovacích metod a zdrojový kód. Záleží také na výběru šifrování vývojářem. Pokud například použil slabou nebo neověřenou knihovnu, úkol se pro testera stává mnohem snazším, kdežto pokud vývojář použil dobré šifrování, jeho dešifrování již od testera vyžaduje expertízu a množství práce, což jej může odradit. Pokud se mu podařilo úspěšně otevřít soubor s uloženými citlivými daty, potvrdil tím existenci zranitelnosti a také by již měl mít představu o tom, co ji může způsobovat. Toto byl pouze obecný postup zneužití této zranitelnosti, demonstrace na konkrétním příkladě je popsána v kapitole 5.2.

```
▼<map>
  <string name="password">supersecretpassword123</string>
  <string name="username">user8526</string>
</map>
```

(a)

```
▼<map>
  <string name="__androidx_security_crypto_encrypted_prefs_key_keyset__">12a9013356a0f184d15d2954aa85bfb2a26a80fa18cfa3938eab3d2cbcd2e2c8b6d73d93319eb5456d77e3a3</string>
  <string name="__androidx_security_crypto_encrypted_prefs_value_keyset__">128801a7f1dc14224d99c1e2dd9a56c6ac5f120151ba83ce64bb342ac629c340e07a46eb53f0398920e71b</string>
  <string name="AVVnxe2VuyRoborrcH3s9XmLr6Q05UZH1SCago=">ATF6uTt8cjioWt6jsk2w4QybXSI8s54+qYbFVN3LWzK2qm6YBs4Cg1h1ex7+tt5iTo=</string>
  <string name="AVVnxfATJ3sSC0hanUWvRvIuhIKWrsuA0nugMU=">ATF6uTE6nU1Sw167oFNOCHMb1241TAJTET+QE196QouhNMqGQA2Ae11ihVsPgyfZ9SKms4398K1jNBXfC6gW</string>
</map>
```

(b)

Obrázek 4.2: Ukázka souboru nešifrovaných sdílených preferencí (a) a souboru sdílených preferencí šifrované klíčem (b). V obou souborech jsou uloženy stejné informace a tyto data je možné z obou souborů získat zpět. Soubor šifrovaných sdílených preferencí navíc obsahuje zašifrovaný klíč potřebný k dešifrování [29].

4.1.4 Dopad zranitelnosti

Dopad zranitelnosti velmi záleží na charakteristice uložených dat, nicméně jakákoliv citlivá data mohou vést přinejmenším k zisku informací o uživateli nebo i celé firmě, které může útočník dále zúžitkovat a narušení jejich soukromí. Pokud jde ale o významnější citlivá data a útočník získá přístup například k účtům uživatelů, jejich mobilním číslům nebo e-mailovým schránkám, může dojít ke krádeži identity či vydírání. Tím také může útočník poškodit jejich reputaci a v nejhorších případech zneužití této zranitelnosti může vyústit i v materiální ztráty [16].

4.1.5 Ochrana před zranitelností

V momentě, kdy aplikace ukládá jakákoliv data uvnitř zařízení, je téměř nemožné je učinit zcela nepřístupná pro kohokoliv nepovolaného. Nejlepším způsobem, jak se tomuto problému vyhnout je ukládat data na externím serveru, ke kterému by se aplikace přihlašovala. Bohužel takové řešení není vždy možné, pokud vývojář potřebuje, aby aplikace například pracovala i offline či nemá prostředky na provozování serveru. Pokud se tedy rozhodneme ukládat data přímo uvnitř zařízení, je potřeba cestu k jejich získání případnými útočníky pořádně zkomplikovat, především pokud se jedná o citlivá data. V úvodu kapitoly již bylo zmíněno jaké údaje můžeme obecně považovat za citlivé, avšak skutečně to velmi záleží na povaze aplikace. V prvé řadě je pro zabezpečení třeba zvážit, která data mohou být považována za citlivá a která ne. Tato data je poté potřeba uložit uvnitř aplikace, aby byla chráněna aplikačním sandboxem. Také není dobré je ukládat v podobě prostého textu, ale použít šifrování [9].

Pro šifrování je důležité zvolit správnou knihovnu. V současnosti existuje mnoho typů šifrování, ať už se ale vývojář rozhodne pro jakýkoliv a zvolí jakoukoliv konkrétní knihovnu, je vhodné ověřit si, zda je skutečně bezpečná. Také je důležité dát si pozor, aby klíč k dešifrování byl náhodně vygenerovaný a nebyl nikde dostupný. Pokud by se k němu útočník mohl snadno dostat, jakékoliv šifrování pomocí tohoto klíče ztrácí smysl.

Na konci vývoje aplikace či jedné její verze je dobré před vydáním provést tzv. obfuskaci kódu. Obfuskace znamená snížení čitelnosti kódu tak, aby se v něm útočník nemohl snadno orientovat a pochopit jej. To zahrnuje například přejmenování metod a proměnných, vložení nefunkčního kódu či zpřeházení pořadí jednotlivých metod [43]. Silně obfuskovaný kód je těžké přechytit a ačkoliv existují některé deobfuskací nástroje, nemusí být vždy efektivní a v každém případě se jedná o velkou komplikaci pro útočníka. Cílem obfuskace je natolik ztížit útočnickovi práci, aby mu nestála za získanou odměnou a vzdal se (více k obfuskaci v sekci 4.3.1).

Nejen u zranitelnosti nezabezpečených datových úložišť je velmi důležitá myšlenka, že ať je použito sebelepší šifrování a sebevíc obfuskace, pokud je útočník opravdu odhodlaný uložená data získat za jakoukoliv cenu, není téměř možné mu v tom zcela zabránit. S největší pravděpodobností se tak nestane, pokud aplikace nenabízí data, která by opravdu stála za množstvím probdělých nocí nad kódem, a tak je metoda odrazení útočníka vcelku účinná.

4.1.6 Příklad zranitelnosti ve skutečném světě

Jelikož je zranitelnost nezabezpečených datových úložišť poměrně běžná, příkladů, kdy vedla k nějakým závažnějším důsledkům, by se nejspíše dalo najít více. Zvolil jsem případ z roku 2016, kdy společnosti Quest Diagnostic kvůli zranitelnosti v mobilní aplikaci MyQuest unikla zdravotní data přibližně 34 tisících osob. Uniklá data podle společnosti

obsahovala jména, data narození, laboratorní výsledky a v některých případech i telefonní čísla. Žádná finanční data obětí útoku však obsažena nebyla. Naštěstí není známa informace o zneužití těchto dat. Ačkoliv nešlo o žádné výrazně citlivé osobní informace, informace o zdravotním stavu jsou čistě soukromé. Společnost Quest Diagnostics tohle věděla či měla vědět, avšak jejich aplikace tato data přesto ukládala nebezpečným způsobem [38, 37].

4.2 ZRANITELNOST EXPORTOVANÉ AKTIVITY

Jako každá jiná aplikace se i aplikace pro Android sestává z několika komponent a tyto komponenty mohou být definovány mnoha parametry. Ty jsou uvedeny v souboru `AndroidManifest.xml`. Některé z těchto komponent mají mimo jiné parametr `android:exported`. Ten obsahuje booleovskou hodnotu, tedy buď `true` (pravda), nebo `false` (nepravda). Pokud je nastaven na kladnou hodnotu, znamená to, že komponenta je přístupná jakékoliv další aplikaci a může jej spustit kdokoli skrze název třídy této komponenty. Pokud je hodnota záporná, může být spuštěna pouze svou aplikací, aplikací se stejným uživatelským ID či privilegovanými komponenty systému [3].

Komponenty, které mohou mít tento parametr definovaný, jsou:

- aktivity,
- služby,
- broadcast receivery,
- content providery.

Všechny tyto komponenty jsou základní stavební složkou Android aplikací. V rámci své práce však pracují pouze s aktivitami, protože jejich zranitelnost považují za nejzajímavější a tento parametr funguje u všech komponent stejně, tedy i zranitelnosti spočívající v jejich nesprávně nastaveného `android:exported` parametru fungují na stejném principu. Ještě před tím, než se pustíme do detailní analýzy této zranitelnosti, je třeba vysvětlit některé zásadní pojmy použité v této části.

Co je to aktivita?

Jako první klíčovou věc je třeba pochopit, co je a k čemu se využívají v aplikaci aktivity. Jedná se o vstupní bod uživatelské interakce s aplikací. Každá aktivita aplikace představuje jednu obrazovku se svými vlastními prvky a jejich vlastní funkcionalitou. Jedna aktivita například může implementovat obrazovku s nastavením a jiná zase obrazovku s výběrem fotografie z galerie [13].

Pro snadnější pochopení si lze aktivitu představit jako webové stránky v prohlížeči. Pokud se nacházíme na nějakém webovém sídle s různými podstránkami a odkazy, můžeme se mezi nimi pohybovat. Při návštěvě každé stránky se nám objeví stránka jiná – s jiným textem, jinou grafikou, atd. Všechny podstránky spadají pod jedno webové sídlo (hraje roli aplikace), avšak každá podstránka má svůj vlastní kód, svou vlastní vizáž a funkcionalitu (hrají roli aktivity). Toto přirovnání pochopitelně není zcela přesné, avšak pro snazší pochopení konceptu aktivit postačí.

Většina aplikací obsahuje více obrazovek - tedy více aktivit. Zpravidla je jedna aktivita označena jako hlavní a jedná se o tu první, která se uživateli objeví při spuštění aplikace.

Každá aktivita má schopnost spustit jinou aktivitu za účelem provedení různých akcí. Například v e-mailové aplikaci může být hlavní aktivitou obrazovka s doručenými e-maily. Odtud pak aktivita může spustit třeba aktivitu k napsání nové zprávy nebo otevřít aktivitu s obsahem konkrétního e-mailu [13].

Co je to intent?

Intent slouží jako prostředek ke komunikaci mezi jednotlivými komponentami. Vyjadřuje záměr učinit nějakou operaci a zároveň slouží jako popis, co se po dané operaci vyžaduje. Může být využit například ke spuštění aktivity nebo ke komunikaci se službami aplikace. Například pokud chce aplikace zobrazit webovou stránku, vytvoří intent s požadovanou akcí a URL adresou a předá jej systému. Ten vyhledá prostředek, který je schopen žádosti vyhovět (v tomto případě webový prohlížeč) a spustí jej [11, 6].

Co je to Intent filter?

Intent filter je parametr, který slouží k regulaci intentů, jež může aktivitu přijímat. Intenty mají několik možných parametrů a na jejich základě je možné je filtrovat. Systém k cílené komponentě doručí pouze takový intent, který projde alespoň jedním filtrem [12]. Pro příklad pokud má být spuštěna aktivita která zobrazí fotografii, může mít intent filter, který povolí aktivitu spustit pouze v případě, že zaslaná položka k otevření je opravdu pouze fotografie, veškeré jiné soubory odmítne.

Důležité je také zmínit, že jakmile aktivita obsahuje intent filter, parametr `android:exported` se automaticky nastaví na hodnotu `true`. Pokud aktivita intent filter nemá a programátor explicitně neurčil jinak, jeho hodnota je naopak automaticky `false`. Od verze Androidu 12 (API úroveň 31) musí mít každá komponenta aplikace využívající intent filter explicitně tento parametr nastavený ať už na kladnou, či zápornou hodnotu. Aplikace, které toto nesplňují nemohou být na zařízeních s touto anebo vyšší verzí systému nainstalovány [3, 12].

4.2.1 Příčiny zranitelnosti

Příčina této zranitelnosti se může zdát zřejmá; aktivita je exportovaná, tedy může ji spustit jakákoliv jiná aplikace, což přirozeně vede k potenciální zranitelnosti, kdy je aktivita spuštěna bez patřičných oprávnění. To však nutně nemusí vždy vyústit ve vznik zranitelnosti, přece jen přítomnost tohoto parametru v systému má svůj důvod. Pakliže je tento parametr explicitně nastaven na kladnou hodnotu, počítá se s tím, že odpovídající komponenta bude přístupná ostatním aplikacím, což by mělo být opodstatněné. Například e-mailová aplikace může chtít, aby po kliknutí na e-mailovou adresu v jakékoli jiné aplikaci bylo možné spuštění aktivity pro napsání zprávy adresátovi, jemuž původní adresa náleží a zjednodušit tím uživateli používání. Jako jiný příklad může třeba firemní aplikace

umožnit po kliknutí na odkaz profilu zaměstnance zobrazit jeho uživatelský profil v rámci firmy. Možnost sdílet své komponenty s jinými aplikacemi může být užitečná, problém nastává teprve při špatné implementaci této funkce.

Vývojáři zmíněné firemní aplikace mohou záměrně vytvořit takovouto funkci a už nemusí myslet na to, že nějaký takový profil může potenciálně obsahovat citlivé údaje jak o zaměstnanci, tak o firmě, a měl by se zobrazit pouze uživatelům, kteří jsou do aplikace také přihlášení jako zaměstnanci. Pokud tedy nějak neomezí (např. pomocí intent filterů) možnost aktivitu s profilem spustit, nebo pokud není uživatel vyvolávající tuto operaci ověřen, může si profil zobrazit kdokoli.

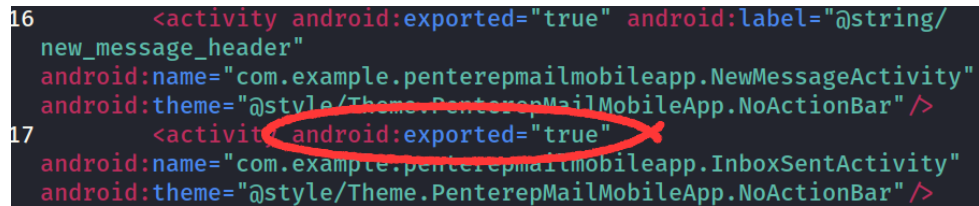
Další příčinou může být i špatné zpracovávání dat předávaných mezi aktivitami. Intent je objekt tzv. typu `parcelable`, tedy umožňuje přidání dalších informací do jeho struktury. Vývojáři toho využívají k vytváření přechodných (proxy) aktivit, které informace získané intentem dále předávají do metod jako je například `startActivity(...)`, `sendBroadcast(...)`, a další. To může při zneužití pomocí vnořených intentů (jeden intent nese jako extra data jiný intent) vést ke spuštění neexportované aktivity, jenž nebyla zamýšlena k účelu být spouštěna jinými aplikacemi skrze jiné zranitelné exportované aktivity.

Použití `WebView` (třída umožňující zobrazení webových stránek v rámci aplikace) může také podobným způsobem skrze exploitovatelnou aktivitu vést k přesměrování uživatele, která jej může dále značně poškodit.

4.2.2 Prevalence a zjistitelnost

Exportované aktivity se v mobilních aplikacích vyskytují vcelku běžně, tudíž i výskyt zranitelnosti vyplývající z jejich špatné implementace není nikterak. Již zmiňovaný žebříček *Mobile Top 10* organizace OWASP uvedl prevalenci zranitelností kategorie *M1: Nesprávné použití platformy*, do níž spadá i tato zranitelnost, jako běžnou [14]. Konkrétní data se opět mohou výrazně lišit, nicméně podle článku blogu společnosti *Oversecured* [31] jejich statistiky vypovídají o 80 % aplikací. Tento výsledek je ale ovlivněn mnoha faktory a je spíše orientační.

Zjistitelnost této zranitelnosti není nikterak složitá. Stačí rozbalit soubor aplikace a prozkoumat soubor `AndroidManifest.xml` (Zde již nepostačí zmiňovaný `dex2jar`, zato nám ale může pomoci např. nástroj `Apktool` či `jadx`). Zde najdeme informace o všech aktivitách aplikace a snadno se dozvíme, zda je některá aktivita exportovaná a zda byl použit nějaký intent filter (viz. obr. 4.3). Informace o přístupných aktivitách lze také získat pomocí různých nástrojů dynamické analýzy. Pokud takovou aktivitu objevíme, je dobré si ještě přečíst její kód a pokusit se pochopit jak aktivita funguje a zda by se skrze ni nedala například otevřít jiná neexportovaná aktivita nebo z ní získat užitečné informace. Případně můžeme zkontrolovat, zda aktivita přijímá nějaký intent nebo nějaké informace z něj, podívat se jaké parametry takový intent musí splňovat a jak s ním aktivita nakládá.



```
16 <activity android:exported="true" android:label="@string/
new_message_header"
android:name="com.example.penterepmailmobileapp.NewMessageActivity"
android:theme="@style/Theme.PenterepMailMobileApp.NoActionBar" />
17 <activity android:exported="true"
android:name="com.example.penterepmailmobileapp.InboxSentActivity"
android:theme="@style/Theme.PenterepMailMobileApp.NoActionBar" />
```

Obrázek 4.3: Ukázka parametru `android:exported` nastaveného na hodnotu `true` v souboru `AndroidManifest.xml` aplikace `PenterepMail` [29].

To, že je aktivita exportovaná, samo o sobě neznamena zranitelnost. Aplikace si může dovolit mít aktivitu exportovanou, pokud je dostatečně ochráněná intent filtry anebo ověřuje obsah intentu, který aktivita zpracovává. Také se může jednat o aktivitu jež je dostupná, avšak neobsahuje žádné užitečné informace. Pokud není možnost toto ověřit z kódu aplikace, relevantnost zranitelnosti se prokáže či neprokáže při fázi exploitace.

4.2.3 Exploitate

Konkrétní proces exploitace pochopitelně opět závisí na povaze exploitované aktivity. Zde popíší opět jen obecný postup, ukázky na konkrétních příkladech jsou v kapitole 5.3.

Pokud jsme našli exportovanou aktivitu, nyní jen potřebujeme způsob, jak ji spustit. K tomu se nám nabízí různé možnosti:

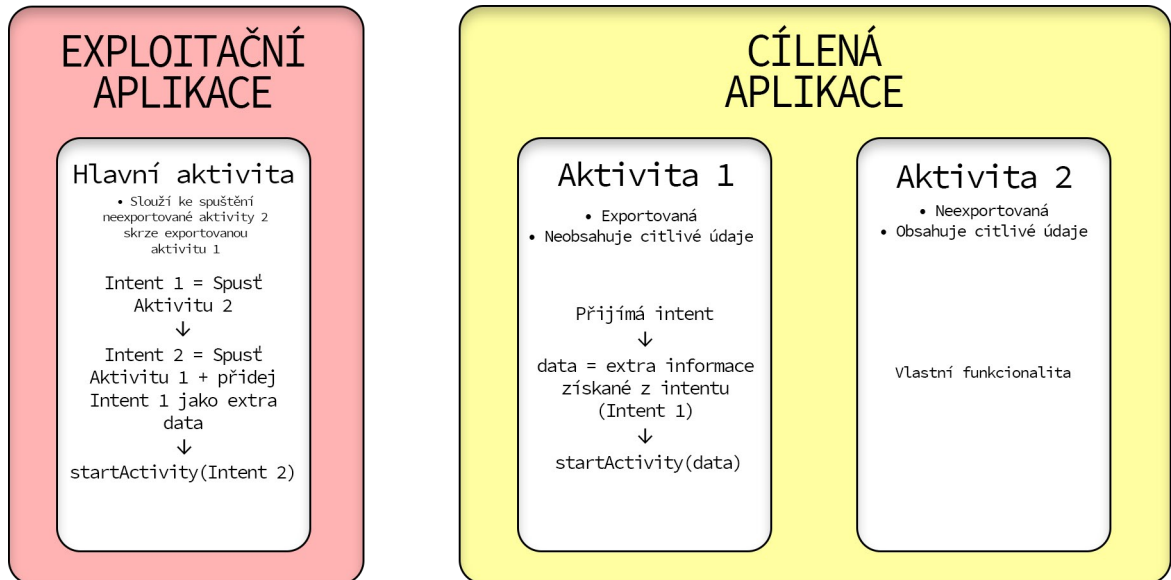
Jako první můžeme využít tzv. ADB (Android Debug Bridge). Jedná se o nástroj, jenž je součástí Android SDK (Software development kit – balíček pro vývoj Android aplikací) a umožňuje komunikaci s Android zařízením skrze příkazový řádek. Součástí ADB je také nástroj *Activity manager* (`am`), který mimo jiné může posloužit ke spuštění aktivity. Po připojení ADB k zařízení stačí zadat příkaz s názvem balíčku aplikace a konkrétní aktivity a ADB se postará o zbytek.

Druhá metoda, jak lze exportovanou aktivitu spustit, je skrze testovací framework, jako je například Drozer. Ten obsahuje široké spektrum různých nástrojů a po připojení k zařízení je vše opět jen na volbě správných příkazů. Drozer nám dokonce může i přímo vypsat aktivity, k nimž je povolený přístup bez jakýchkoliv oprávnění a jiný nástroj je dokáže zase spustit.

Zajímavá je také ještě varianta spuštění aktivity skrze externí aplikaci. Ve vývojovém prostředí, jako je např. Android Studio, můžeme vytvořit jednoduchou miniaplikaci, která třeba po kliknutí na tlačítko vyvolá intent ke spuštění dané aktivity. Jedná se o nejsložitější variantu, avšak je o něco více realistická vzhledem k tomu, že v praxi útočníci nemají vždy možnost vzdáleného přístupu k cíleným zařízením. V takovém případě se mohou uchýlit ke snaze přesvědčit uživatele ke stažení škodlivé aplikace a spuštění aktivity skrze ni [51].

Pro exploitaci exportované proxy aktivity skrze intent je nejlepší použít vlastní miniaplikaci pro skládání vlastního intentu. Pokud chceme spustit aktivitu, která nabízí

citlivé údaje, ale není exportovaná, pokus o její spuštění skrze metodu `startActivity()` z naší aplikace by vyústil v chybu. Pokud je ale v aplikaci přítomna exportovaná aktivita která skrze přijímaný intent spouští jinou aktivitu, můžeme v naší aplikaci sestavit vlastní intent, který spustí uvažovanou exportovanou aktivitu. Do něj jako extra data přibalíme název cílené neexportované aktivity. Ta je poté spuštěna skrze metodu `startActivity()` v proxy aktivitě (viz obr. 4.4) [31]. Podobný postup lze použít i při provádění XSS (Cross-site scripting útoků), kdy lze do prvního intentu zabalit URL adresu, a ta při využití WebView uživatele odkáže na škodlivou stránku.



Obrázek 4.4: Grafické znázornění jednotlivých aktivit a jejich funkcionality při exploitaci exportované proxy aktivity skrze vnořený intent [29].

4.2.4 Dopad zranitelnosti

Dopad této zranitelnosti opět silně záleží na vlastnostech exportované aktivity a dosahu jejího exploitu (např. zda je možnost spustit jiné aktivity a útok dále rozvádět). Potenciální zisk pro útočníka však může tvořit přístup k datům, jež by měly být dostupné pouze autorizovaným uživatelům. Velké nebezpečí spočívá i v možnosti provádění XSS útoků skrze exportovanou aktivitu, kdy cílená aktivita využívá rozhraní WebView a otevírá webové stránky z uživatelského vstupu bez jejich většího ověřování. Tyto stránky jsou poté považovány za důvěryhodné a uživatel vůbec nemusí rozpoznat, že se jedná o škodlivé stránky a může bez rozpaků odevzdat své soukromé údaje do rukou útočníka [8]. Tyto útočník může dále zpracovat a v horším případě to může vést i k útočnickovu peněžnímu zisku.

4.2.5 Ochrana před zranitelností

Nejlepším a nejjistějším způsobem, jak zabránit vzniku této zranitelnosti je exportované aktivity vůbec nepoužívat. Pokud to funkcionality aplikace vyžaduje, pak by měl být

parametr `android:exported` nastavený na kladnou hodnotu pouze u aktivit, které to potřebují a všechny ostatní aktivity by tento parametr měly mít explicitně nastavený na hodnotu `false`. Toto je koneckonců také vyžadováno u všech aplikací mířených pro zařízení se systémem Android verze 12 (API úroveň 31) a vyšší [39]. Dobrým zvykem je také vyhýbat se využívání vnořených intentů a deep linků, je-li to možné. U aktivit, které jsou exportované, je pak důležité omezit možnosti jejich spuštění. První možností je použít zmíněné intent filtry (viz. kapitola 4.2). Každý filter specifikuje typ akceptovaných intentů na základě jejich akce, kategorií a typu dat. Ty mají mnoho podparametrů, díky kterým lze požadovaný intent nastavit vcelku explicitně. Aktivitu poté nebude možné spustit pomocí jiných intentů, než které projdou alespoň jedním intent filterem [12].

Další nástroj který můžeme využít k ochraně před zranitelností exportované aktivity je komponenta `<permission>`. Ta je využívána v souboru `AndroidManifest.xml` k definování vlastních druhů povolení. Ty mohou být definovány více parametry, klíčovým je zde parametr `android:protectionLevel`. Pokud naší aplikaci stačí sdílet své aktivity pouze s jinými aplikacemi, které jsme vytvořili také my, lze použít atribut `signature`. Jeho použitím bude oprávnění uděleno pouze aplikacím, které jsou podepsány stejným certifikátem. Dále je zde atribut `signatureOrSystem`, který tento výběr rozšiřuje i na aplikace, které byly nainstalovány v rámci systému. Mimo jiné ještě zmíním atribut `dangerous`, který indikuje velké riziko při spuštění této aktivity a může si vyžádat uživatelské potvrzení ke spuštění [23].

Pro ochraně proti XSS útokům a vnořených intentů je dobré řídit se pravidlem „Nikdy nevěřte vstupu uživatele“. Pokud aktivita zpracovává jakýmkoliv způsobem vstup, jenž může ovlivnit uživatel (například zadá URL adresu webové stránky), měla by vždy ověřit, zda se jedná opravdu o vstup, který aktivita očekává, a zda-li porušením této podmínky nemůže dojít k nějaké nežádané akci.

Do exportované aktivity může také být zabudovaná nějaká ověřovací funkce, která například ověří, zda je uživatel přihlášený, či jinak sama zkontroluje legitimitu volání nezávisle na zabezpečení systému. Metod, kterými lze omezit přístup k aktivitě nebo jejím datům je mnoho, avšak vytváří komplikace v momentě, kdy má všemi filtry projít žádaný intent, který musí splňovat velké množství podmínek. To činí aplikaci komplikovanější a hůře udržitelnou. Je proto opět důležité zvážit míru rizika a rozmyslet si, zda nějaké zabezpečovací mechanismy nejsou již nadbytečné a kontraproduktivní.

4.2.6 Příklad zranitelnosti ve skutečném světě

Díky běžnému výskytu této zranitelnosti mezi aplikacemi je závažných zranitelností u produkčních aplikací mnoho. Zde jsem vybral zranitelnost popsanou v článku na platformě www.hackerone.com. Uživatel *bagipro* zde popisuje zranitelnost nalezenou v mobilní aplikaci Slack. Uvnitř hlavní aktivity, která byla exportovaná a dostupná všem ostatním aplikacím byl kód, který umožňoval skrze vnořený intent spustit jakoukoliv

aktivitu, exportovanou i neexportovanou. Dále se tato zranitelnost dala zneužít k otevření WebView aktivity, které mohla být pomocí vnořeného intentu předložena vlastní URL adresa, a ta by uživatele mohla přenést na škodlivé stránky. Bylo také možné provádět telefonáty reálným lidem, zobrazovat uživateli falešný obsah či zachytit a případně zaměnit soubory přijaté aplikací. Platforma slack tuto zranitelnost označila jako závažnou a uživatele, jenž tuto zranitelnost odhalil odměnila finanční odměnou ve výši 1000 \$. Nalezená zranitelnost byla samozřejmě opravena [49].

4.3 ABSENCE BINÁRNÍ PROTEKCE

Při napadání jakékoliv mobilní aplikace je analýza zdrojového kódu jedna z prvních věcí, kterou útočník provede. Také může prohlédnout její soubory, pakliže je nainstalována na rootovaném zařízení. Někteří útočníci také dokonce stahují známé aplikace, analyzují jejich kód, upraví jej způsobem, který v nich vytvoří bezpečnostní slabiny či si vytvoří zadní vrátka a poté aplikaci distribuují jako originální. Uživatel, který si takovou aplikaci stáhne může používat nebezpečnou aplikaci v domnění, že se jedná o originál, zatímco útočníci mají přístup ke všem jeho údajům. Binární protekce je způsob jak omezit možnosti takové aktivity provádět. Jedná se o obecnou ochranu kódu tak, aby byl co nejnečitelnější, nemohl být spuštěn na rootovaném zařízení nebo aby kód nemohl být snadno zmanipulován. Metod, jak zvýšit bezpečnost kódu je mnoho. Pokud jsou dobře implementovány, zkomplikuje to útočníkům analýzu a pokud aplikace obsahuje nějaké další zranitelnosti, nemusí je díky ochráněnému kódu vůbec odhalit. Je však důležité, že ochrana kódu pomocí všemožných metod útočníka může velmi zpomalit, zkomplikovat mu práci nebo jej odradit, nicméně zcela zabránit v útoku mu nedokáže [15].

Opatření proti této zranitelnosti v dokumentu MASVS organizace OWASP spadají pod kategorii MASVS-R (Sada požadavků pro odolnost vůči reverznímu inženýrství). Jak bylo zmíněno v kapitole 1, opatření spadající pod tuto kategorii neslouží jako bezpečnostní mechanismy pro aplikaci, ale pouze jako dodatečné zabezpečení proti reverznímu inženýrství, pokud je to jedním z cílů vývojářů. Jejich použití tedy není nutné v každé aplikaci a jejich nepoužití by nemělo vyústit v příliš závažný problém, nicméně jedná se o další vrstvu ochrany, která posílí zabezpečení u zvláště citlivých aplikací [20].

Pro lepší pochopení zranitelnosti jsem si dovilil pozměnit lehce strukturu kapitoly. Upravil jsem pořadí jednotlivých podsekcí tak, aby byly nejprve vysvětleny jednotlivé metody ochrany kódu a poté chyby, které mohou při zabezpečování nastat. Také jsem jednotlivé podsekcce rozdělil do částí pro lepší orientaci.

4.3.1 Ochrana před zranitelností

Obfuskace

Pokud naše aplikace obsahuje opravdu citlivá data, nebo z jakéhokoliv jiného důvodu nechceme aby někdo mohl zkoumat její kód nebo jím manipuloval, můžeme použít vícero různých metod. Jedním z nich je obfuskace, která byla již zčásti vysvětlena v sekci 4.1.5. Ta, jak bylo zmíněno, snižuje čitelnost kódu a útočník musí buď použít deobfuskační nástroje, nebo se pokusit v něm sám vyznat, pokud má s obfuskací již zkušenosti. Nejjednodušší je použít nástroj Android ProGuard nebo R8, které jsou součástí prostředí Android Studio. Tyto nástroje kromě obfuskace také odstraní nepotřebné části kódu a zmenšují celkovou velikost aplikace. Z těchto dvou nástrojů je však více doporučovaný R8, poněvadž je efektivnější a jeho používání je jednodušší [19]. Tím, že se ale jedná o často používaný způsob, který je prováděný přímo při kompilaci kódu do `.apk` souboru, může

být snadno deobfuskovaný i některými dekompilátory, nebo jinými externími nástroji. Také ne všechny obfuskátory mění i hodnoty jednotlivých proměnných (například string), a tak některé hodnoty v něm mohou být čitelné i po dekompilaci (viz obr. 4.5).

```
/**
 * Logs you into your SIP provider, registering this device as the location to
 * send SIP calls to for your SIP address.
 */
public void initializeLocalProfile() {
    if (manager == null) {
        return;
    }

    if (me != null) {
        closeLocalProfile();
    }

    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
    String username = prefs.getString("namePref", "");
    String domain = prefs.getString("domainPref", "");
    String password = prefs.getString("passPref", "");
}
```

(a)

```
public void b()
{
    if (this.b == null)
        return;
    if (this.c != null)
        c();
    SharedPreferences localSharedPreferences = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
    String str1 = localSharedPreferences.getString("namePref", "");
    String str2 = localSharedPreferences.getString("domainPref", "");
    String str3 = localSharedPreferences.getString("passPref", "");
}
```

(b)

Obrázek 4.5: Ukázka originálního kódu (a) a dekompilovaného kódu po použití obfuskace nástrojem ProGuard (b). Názvy proměnných jsou změněné, avšak vepsané string hodnoty jsou ve stejném stavu [32].

Root detekce

Obfuskace tedy ztíží útočníkovi čtení kódu, avšak například zranitelnost nezabezpečeného úložiště (4.1), kdy mohou být citlivé informace uloženy třeba ve sdílených preferencích, může být stále odhalena prozkoumáním právě těchto úložišť. K tomu, aby se však útočník k těmto datům dostal bez využití nějaké další zranitelnosti, potřebuje spustit zařízení s administrátorskými právy, tedy provést takzvaný *root* telefonu. Administrátor má na zařízení neomezená oprávnění a mimo jiné může prohlížet veškeré soubory aplikací, které jsou jinak uzavřené v aplikačním sandboxu. Provést root fyzického zařízení není příliš složitý úkol, a ještě snadnější je vytvořit si virtuální zařízení, které již administrátorská práva má. Někteří lidé svá zařízení schválně rootují, protože jim to poskytuje možnost upravovat si vzhled systému, chování telefonu a další věci. Je to zároveň ale jak pro uživatele, tak i pro vývojáře aplikací na nich nainstalovaných bezpečnostní riziko. V případě, kdy by se útočník zmocnil takového zařízení anebo by do něj byl nainstalován malware, mohl by bez problému ukrást veškeré jeho údaje. Pro bezpečnost uživatele i aplikace je proto lepší zajistit, aby na takovém zařízení aplikace neběžela.

Nejčastějším a nejjednodušším způsobem, jak zjistit, zda aplikace běží na rootovaném zařízení je pokusit se provést nějakou operaci, k jejímuž provedení jsou zapotřebí administrátorská práva. Může se jednat například o pokus přistoupit k souborům, ke kterým by aplikace běžící bez administrátorských práv neměla mít přístup. Různých takových kontrol je mnoho, avšak s dnešními antidefekčními nástroji se tyto kontroly

dají snadno překonat. Kromě obecných kontrol může být také dobrým způsobem detekce implementace detekčních nástrojů, které jsou zaměřené přímo proti konkrétním nástrojům [33].

Například když testeři testují nějakou aplikaci, nejčastěji ji testují na virtuálním zařízení spuštěné pomocí tzv. *emulátoru*. Emulátor je program, který umožní spuštění jiného programu, který jinak na zařízení spustit nelze [52]. V našem případě ho lze vnímat jako program na počítači, jenž na něm spustí virtuální mobilní zařízení. Máme tak k dispozici plně funkční mobilní zařízení k testování aplikace, které si navíc můžeme do značné míry upravit. Implementace detekce emulátoru tak testerovi velmi zkomplikuje práci, poněvadž bude muset aplikaci testovat buď na fyzickém zařízení, které má ovšem velmi limitované možnosti oproti emulátoru, anebo přijít na způsob, jak detekci obejít [33].

V neposlední řadě je dobré zkontrolovat přítomnost různých rootovacích nástrojů i pokud je aplikace spuštěna na fyzickém zařízení. Jako příklad mohu uvést nástroj Magisk. Magisk je nástroj sloužící k rootování zařízení „bezsystemovým“ způsobem. To v jednoduchosti znamená, že při takovém postupu nejsou systémové soubory nijak modifikovány. V současné době se jedná o jeden z nejpopulárnějších způsobů rootování zařízení a proto zavést detekci proti tomuto nástroji je vhodné opatření v případě, že je opravdu nezbytné znemožnit aplikaci běh na zařízení běžícím s administrátorskými právy [35, 47]. Pokud některý z těchto mechanismů rozpozná, že aplikace běží na emulátoru či zařízením s administrátorskými právy, je potřeba na to adekvátně zareagovat v závislosti na povaze aplikace. Pokud obsahuje skutečně citlivá data, je bezpečnější ji vůbec znemožnit spuštění či jakoukoliv funkcionalitu. Pokud uvážíme, že by aplikace mohla fungovat, avšak některé funkce by představovaly bezpečnostní riziko, můžeme aplikaci pouze omezit.

Detekce změny kódu

Jako poslední příklad binární protekce kódu uvedu detekci změny kódu. Díky tomu, že operační systém Android je otevřený, a podporuje instalaci aplikací i z jiných zdrojů, než jen z oficiálního obchodu Google Play, je velikým lákadlem útočníků změnit kód k jejich prospěchu a distribuovat škodlivou aplikaci jako legitimní. Také při testování aplikace si tester může pozměnit kód ke svým potřebám a přimět aplikaci provádět operace, které mohou vést k zisku dat. Nejsnazším způsobem jak ověřit, zda je aplikace legitimní, je pomocí *podpisového klíče*. Podpisový klíč je atribut, který musí mít každá Android aplikace k tomu, aby mohla být nainstalována. Je pro každou aplikaci jedinečný, v průběhu vývoje aplikace se nemění a je chráněný heslem, které zná pouze vývojář. Bez tohoto hesla nelze aplikaci stejným klíčem podepsat, a pokud by kdokoliv cizí chtěl nainstalovat upravenou aplikaci, musel by ji podepsat jiným klíčem. Jelikož je navíc podpisový klíč atribut, jenž lze pomocí kódu z aplikace získat v zašifrované podobě, pro kontrolu legitimacy aplikace jej stačí pouze porovnat s tím správným. Pokud se klíče shodují, aplikace je legitimní a nebyla upravena cizím subjektem. Pokud ne, tak pravděpodobně byla, a je bezpečnější jí zabránit ve fungování [26, 4].

4.3.2 Příčiny zranitelnosti

Jelikož mluvíme o zranitelnosti spočívající v absenci binární protekce, její příčina je zřejmá již z jejího názvu, tedy není použita žádná nebo pouze nedostatečná míra ochrany kódu. Kde se nachází hranice toho, zda je ochrana dostatečná či nikoliv opět závisí na povaze aplikace. Zde musí vývojáři opět zvážit zda jejich aplikace může být zneužita např. k šíření malwaru, nebo zda upravení jejího kódu může vést k odhalení či zneužití dalších potenciálních zranitelností. I pokud vývojáři zhodnotí rizika správně a aplikují určité způsoby protekce, je důležité dbát na jejich správnou implementaci (tak jako u jakékoliv jiné funkce aplikace či formě jejího zabezpečení). Klíčové je také vrstvení těchto opatření, samy o sobě nejsou příliš funkční. Například detekce změny kódu je jednoduché a užitečné opatření, avšak stejně tak jako může útočník změnit kód k provádění nebezpečných aktivit, může také z kódu odebrat detekci změny podpisového klíče a aplikaci dále používat i po změně. Pokud je ale část kódu zařizující tuto detekci v kódu dobře uschována a obfuskována a případně zašifrována, lze šance na její odhalení útočníkem minimalizovat. Při testování aplikace je dobré tyto opatření dočasně vypnout, aby mohla být dobře zanalyzována a otestována funkcionalita aplikace. Binární protekce představuje spíše štít nad celou aplikací, avšak není neprolomitelný a pokud aplikace pod ním obsahuje zranitelnosti, tento štít ji nezachrání. Až poté, co je řádně zkontrolována aplikace samotná, mohou být aktivována binární opatření, která by měla opět projít ještě dalším testem.

4.3.3 Prevalence a zjistitelnost

Běžnost výskytu aplikací neobsahující dostatečnou binární protekci je extrémně velká, nicméně může to být odůvodněno tím, že se nejedná o nezbytně nutné zabezpečení [15].

Zjistitelnost absence binární protekce je pro útočníka velmi snadná, dá se říci, že téměř automatická. Spíše než že binární protekce chybí, útočník objeví, že nechybí, poněvadž to pro něj představuje komplikace. Pokud se dekompilovaná aplikace sestává z čitelného kódu, dovolí útočníkovi ji spustit s administrátorskými právy či provádět reverzní inženýrství, binární protekce buď nebyla implementována vůbec, anebo nedostatečným způsobem. Oproti tomu vývojář si zranitelnost uvědomí nejspíše až v momentě, kdy zjistí, že se někomu podařilo úspěšně upravit aplikaci a její modifikovanou verzi upravovat. Jedná se navíc zpravidla spíše o náhodu, než že by vývojář záměrně prohledával obchody s aplikacemi [15].

4.3.4 Exploitate

Pokud aplikace, kterou testujeme, nemá binární protekci, není zde co překonávat. Máme umožněný plný přístup ke kódu, který můžeme libovolně číst nebo měnit. Nicméně tato zranitelnost zahrnuje i stav, kdy aplikace binární protekci obsahuje, avšak na nedostatečné

úrovni. To v praxi znamená, že ochranná opatření lze snadno překonat. Například obfuskovaný kód se můžeme pokusit znovu deobfuskovat. Pokud tuto činnost nechceme provádět manuálně, což vyžaduje zkušenosti, znalost programovacího jazyka a spoustu času, můžeme zkusit nějaký automatizovaný nástroj, např. Simplify. Ten virtuálně spustí aplikaci, aby pochopil její fungování a poté optimalizuje kód tak, aby se choval stejně a zároveň byl čitelný pro člověka [40].

Další z ochranných mechanismů, který se lze pokusit překonat je root detekce. K tomu můžeme využít nástroj pro dynamickou analýzu Frida, který zahrnuje spoustu užitečných nástrojů, jako je právě script `fridantiroot.js`. Tento kód se pokouší různými způsoby zamaskovat informaci o tom, že je zařízení rootované tak, aby při zavolání této informace byla vrácena negativní hodnota. Pokud jej opět pomocí nástroje Frida vnoříme do cílené aplikace, je šance, že se nám podaří root detekci obejít [34]. Také, v případě, že aplikace nezahrnuje detekci změny kódu, můžeme jednoduše v dekompilovaném kódu nalézt mechaniku pro root detekci a přepsat ji tak, aby byla nefunkční [42]. Pokud aplikace detekci změny kódu obsahuje, poté se můžeme pokusit ji taktéž najít a odstranit ji z kódu. Poté už stačí jen aplikaci podepsat novým klíčem, zkompilovat ji a nainstalovat zpět na zařízení.

4.3.5 Dopad zranitelnosti

Obecně absence binární protekce způsobí, že aplikace může být snadno a rychle analyzována, reverzně sestavena a modifikována [15]. To může v závislosti na povaze aplikace dále vést ke množství negativních následků. Útočník může třeba odhalit v kódu další zneužitelné slabiny, ze kterých by měl vyšší zisk. Jinou možností je do kódu vložit malware, aplikaci zkompilovat a distribuovat ji přes různé obchody s aplikacemi jako původní legitimní aplikaci a získávat tím data uživatelů. Případně může kód být zkopírován a použit jako základ pro jinou aplikaci určenou k prodeji, což útočníkům zajistí rychlý a snadný výdělek. V neposlední řadě může být aplikace modifikována k osobním účelům, typicky se jedná o módy do videoher. To znamená, že se původní hra pozmění, odeberou se či přidají nové funkce a hráči tak získají nový požitek ze hry. Důsledků, ke kterým může kvůli absenci binární ochrany dojít je tedy mnoho, a pokud vývojáři explicitně nechtějí, aby k něčemu takovému došlo, je vhodné alespoň základní míru protekce do jejich mobilních aplikací implementovat.

4.3.6 Příklad zranitelnosti v reálném světě

Jako příklad zranitelnosti ve skutečném světě jsem se rozhodl uvést situaci, kdy v roce 2018 byla na různých obchodech s aplikacemi nalezena falešná kopie aplikace Uber, původně sloužící k objednání přepravy automobilem. Tato falešná aplikace byla velmi dobře zpracovaná a skutečně se tvářila jako originální aplikace. Měla v sobě ale zabudovaný malware, který uživatele čas od času vyzval k přihlášení. Poté, co uživatel zadal správné údaje, aplikace začala dále normálně pracovat. Data, která uživatel zadal byla zachycena

aplikací a předána hackerům, kteří se pokusili napadnout účty se stejnými údaji na jiných stránkách, a také je přeprodávali dalším hackerům skrze dark web.

Aplikace nicméně neprošla kontrolami na obchodu Google Play a byla k dispozici pouze na obchodech třetích stran. Z toho vyplývá, že není vhodné stahovat aplikace z jiných obchodů, obzvláště ty pocházející z Ruska, odkud byla i tato aplikace, protože představují vysoké bezpečnostní riziko [30].

5 DEMONSTRACE VYBRANÝCH ZRANITELNOSTÍ

Kromě analýzy je při snaze pochopit, jak zranitelnost funguje, důležité také vidět, případně si vyzkoušet proces jejího objevení a také exploitace. Proto jsou všechny z probíraných zranitelností v této kapitole demonstrovány na jednoduchých příkladech. K tomu slouží tzv. zranitelné aplikace (v angličtině *vulnerable apps*), což jsou aplikace záměrně obsahující různé typy zranitelností. Zpravidla se jedná o opravdu jednoduché aplikace a jejich zranitelnosti nejsou nikterak sofistikované, což umožňuje vyzkoušet si zranitelnosti na vlastní kůži téměř komukoliv. Aby si čtenář mohl vyzkoušet příklady zranitelností sám je zde také podrobně popsán instalační proces jednotlivých programů a nástrojů využívaných při demonstraci, stejně tak jako postup exploitace. Nutno opět podotknout, že veškeré techniky demonstrované v této kapitole i v celé práci jsou prováděny na mém vlastním zařízení, ke kterému mám plný přístup, a nikdo by neměl činit jinak. Použití hackerských technik na cizím zařízení bez oprávnění může být považováno za trestný čin (a každý je sám zodpovědný za své jednání).

5.1 PŘÍPRAVA PROSTŘEDÍ

Ke kompletní demonstraci zařízení je potřeba nejprve vytvoření testovacího prostředí. To bude tvořit naše fyzické zařízení, odkud budou spouštěny příkazy a bude probíhat komunikace se zařízením, a emulátor. Na něm poběží virtuální mobilní zařízení, na kterém budou nainstalovány zranitelné aplikace a se kterým bude manipulováno. Veškeré úkony je možné provést i s fyzickým zařízením, to s sebou však přináší omezení a bezpečnostní rizika. Použití emulátoru je jednodušší, bezpečnější a praktičtější. Všechny popsané postupy jsou aktuální k datu odevzdání této práce. Je možné, že později se již struktura webových stránek či instalačních balíčků změní, a postupy tedy nemusí být vždy zcela aktuální.

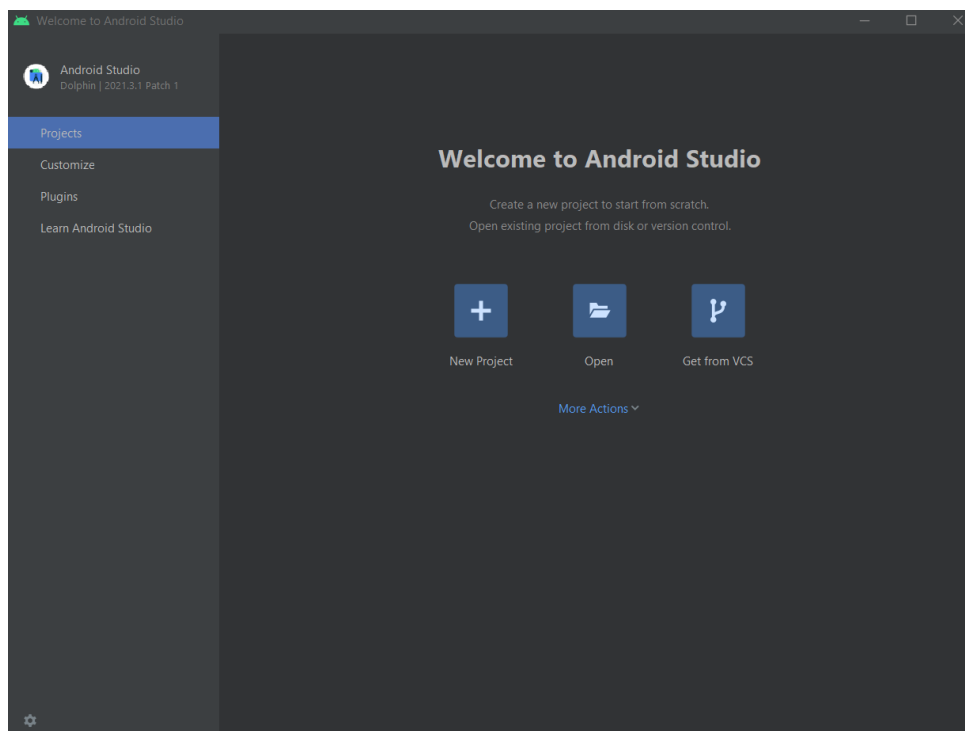
Android Studio

Android Studio je vývojové prostředí vytvořené společností Google sloužící pro vývoj aplikací pro Android. Je založeno na editoru IntelliJ IDEA, což je jeden z nejsilnějších a nejpopulárnějších editorů pro programovací jazyk Java [17]. Kromě možnosti vyvíjení aplikací pro Android ale nabízí ještě další vymoženost, kterou je emulátor Android zařízení. To je ostatně důvod, proč bude pro demonstraci užitečné. V každém případě lze použít téměř jakýkoliv jiný emulátor, například Genymotion nebo jiné. Já jsem zvolil tento díky snadnosti používání, a protože s ním bylo nejméně problémů.

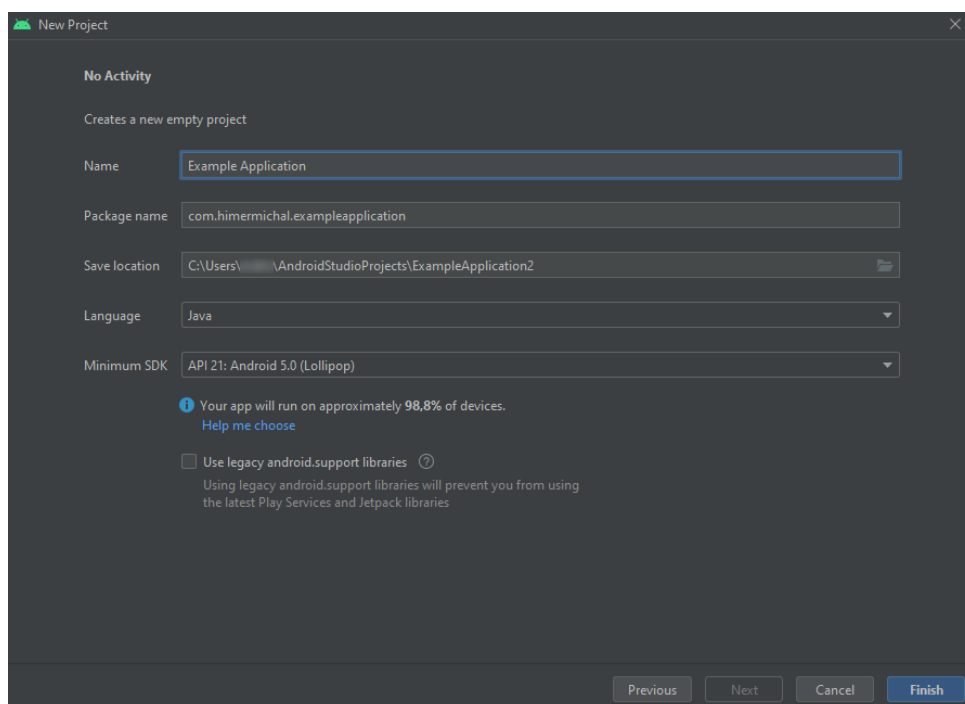
Pro stažení instalačního balíčku programu Android Studio je nutno navštívit oficiální stránku programu *developer.android.com/studio* (všechny odkazy jsou uvedené v souboru *links.txt*, který je součástí elektronické přílohy). Po kliknutí na tlačítko *Download Android Studio* nás stránka vyzve k souhlasu s podmínkami a ustanoveními ohledně používání programu. Po udělení souhlasu je zpřístupněno tlačítko a po kliknutí na něj je zahájeno stahování. Po otevření instalačního balíčku jsme vyzváni k udělení administrátorského práva potřebného k instalaci. Jakmile to uděláme, spustí se instalační procedura. V ní si můžeme zvolit instalační složku a zda si přejeme nainstalovat i emulátor pro virtuální zařízení. Protože právě emulátor je hlavní nástroj, který je k demonstraci potřeba, měli bychom se ujistit, že je volba zaškrknuta. Jakmile si zvolíme i instalační složku a složku startovacího menu, zahájí se instalační proces.

Po nainstalování Android Studia a jeho otevření se ještě otevře krátký průvodce, který doinstaluje všechny dodatečné potřebné soubory. Po dokončení tohoto procesu se otevře již úvodní obrazovka Android Studia (viz. obr. 5.1). Zde si můžeme vytvořit či otevřít projekty aplikací. K tomu, aby se zpřístupnily všechny funkce emulátoru, bude potřeba vytvořit prázdný projekt. Kliknutím na tlačítko *New project* se spustí průvodce. V první obrazovce si můžeme vybrat předlohu a v druhé detaily projektu (viz. obr. 5.2). Vzhledem k účelu vytváření projektu je ale zcela jedno, jakou předlohu si zvolíme a jaké nastavíme detaily, poněvadž projekt jako takový nebude pro demonstraci využíván. Po vytvoření

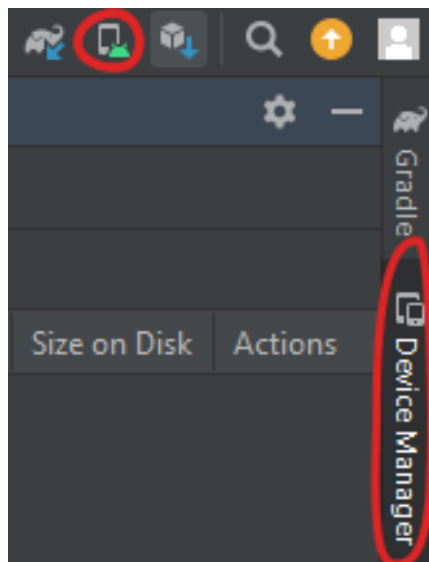
projektu je třeba vytvořit emulátor. To se udělá v okně *Device manager*, které lze spustit kliknutím na příslušný nápis v liště nalevo, anebo kliknutím na tlačítko s ikonou mobilního telefonu a hlavou robota Android (viz. obr. 5.3).



Obrázek 5.1: Úvodní obrazovka programu Android Studio [29].

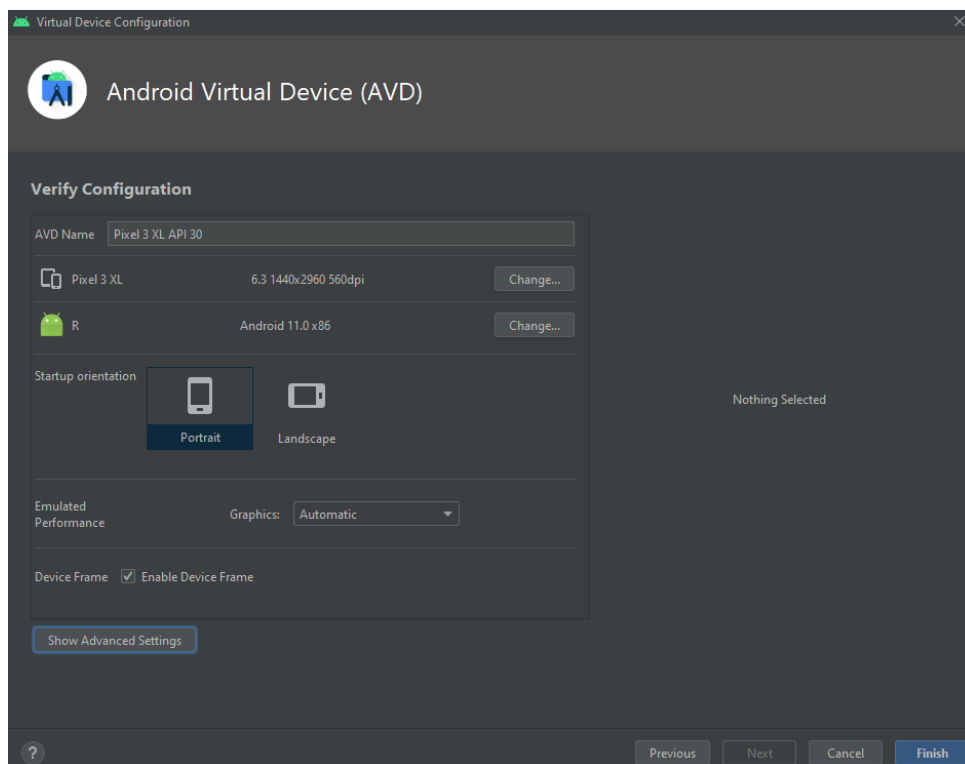


Obrázek 5.2: Detaily prázdného projektu v Android Studiu [29].



Obrázek 5.3: Možnosti spuštění správce virtuálních zařízení v pravém horním rohu programu [29].

Po spuštění správce virtuálních zařízení zvolíme pravděpodobně jedinou možnost, kterou nám program nabídne, a to *Create virtual device*. Otevře se nabídka s různými typy telefonů, které můžeme vytvořit. Typ telefonu není pro demonstraci příliš zásadní, jediná důležitá vlastnost je, zda má zařízení obchod Google Play. Zařízení, které jej nemají, běží v základním nastavení s administrátorskými právy. To je ale vlastnost, kterou potřebujeme už jen pro demonstraci absence binární protekce. Pro demonstraci jsem tedy zvolil zařízení Google Pixel 3 XL. V další nabídce je třeba vybrat verzi systému Android. Pro relevantnost zranitelností jsem zvolil spíše novější verzi, Android R, nebo také 11.0 (API úroveň 30). V případě, že tato verze SDK není v zařízení stažená, je potřeba ji nejprve stáhnout kliknutím na ikonu vedle názvu Android verze. Jakmile je operace dokončena, můžeme si ještě v další nabídce upravit jméno a detaily zařízení (viz. obr. 5.4). Jakmile jsme se vším spokojeni, kliknutím na tlačítko *Finish* dokončíme proces. Ke spuštění emulátoru pak stačí pouze kliknout na ikonu šipky ve sloupečku *Actions*. Spuštěný emulátor je na obrázku 5.5.



Obrázek 5.4: Konfigurace vytvářeného virtuálního zařízení [29].



Obrázek 5.5: Spuštěný emulátor pomocí Android Studio [29].

Frida

Frida je nástroj pro dynamickou analýzu vytvořený Ole André V. Ravnåsem. Podobně jako například Drozer obsahuje široké množství nástrojů, díky kterým je možné otestovat běžící aplikace proti široké škále zranitelností. Díky její dostupnosti a snadné instalaci patří mezi ty nejpopulárnější nástroje pro testery. V kapitole demonstrací využijeme její schopnost manipulovat s kódem aktivit k vypnutí detekce emulátoru.

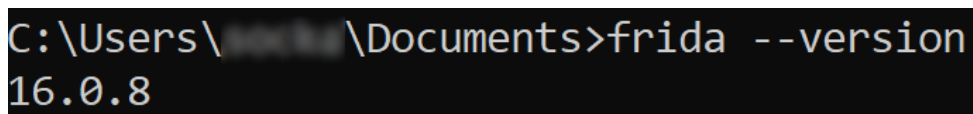
Samotná instalace Fridy je jednoduchá, avšak vyžaduje nainstalovaný Python3, pip3 (Pip Installs Packages), adb (Android Debug Bridge) a mobilní zařízení či emulátor běžící s administrátorskými právy. Na něm ještě musí běžet server Fridy, který umožní komunikaci se zařízením. Ač se to může zdát jako mnoho požadavků, nástroj adb jsme nainstalovali zároveň s Android Studiem, a pip je nainstalován zároveň s Pythonem. Emulátor jsme si již také připravili, takže ani o root zařízení se nemusíme starat. Pro instalaci Pythonu na zařízeních s operačním systémem Windows je nejjednodušší cestou jeho stáhnutí z aplikace Microsoft store. Zde stačí nalézt nejnovější verzi Pythonu a kliknutím na tlačítko *Získat* se automaticky stáhne, nainstaluje a nakonfiguruje pro snadnější používání. Alternativou je stáhnutí instalačního balíčku z oficiálních stránek *python.org* a jeho manuální instalace. Poté co máme vše připraveno, můžeme nainstalovat Fridu.

Pro stáhnutí Fridy do počítače je nutné použít konzoli, neboli příkazový řádek. Ten lze otevřít například pomocí klávesové zkratky Win + R a následně zadání příkazu `cmd` do dialogového okna. V konzoli pak stačí zadat příkaz:

```
pip install frida-tools
```

a Frida se sama nainstaluje. Pro ověření, že instalace proběhla správně můžeme použít příkaz:

```
frida --version
```



```
C:\Users\... \Documents>frida --version
16.0.8
```

Obrázek 5.6: Výstup příkazu `frida --version`, pakliže instalace proběhla úspěšně (číslo verze se může lišit v závislosti na aktualizacích nástroje) [29].

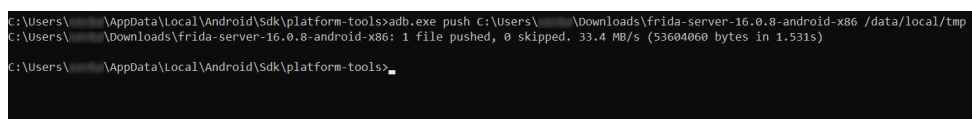
Výstup příkazu lze vidět na obr. 5.6. Posledním krokem je instalace a spuštění serveru na mobilním zařízení. K tomu musíme stáhnout správný soubor z repozitáře na platformě *github.com* (odkaz je uvedený v souboru `links.txt`, který je součástí elektronické přílohy). V tomto repozitáři je mnoho souborů, proto musíme vědět, který stáhnout. Správný soubor by měl mít název v podobě `frida-server-<číslo verze>-android-<typ systému>.xz`. Číslo verze musí být stejné jako verze Fridy nainstalované na počítači (viz. obr. 5.6). Typ systému pro náš emulátor je `x86`. Správný soubor v této práci

nese tedy název `frida-server-16.0.8-android-x86`. Kliknutím na příslušný soubor se zahájí stahování komprimovaného balíčku, který je třeba ještě rozbalit do libovolného úložiště. K přenesení souboru na mobilní zařízení využijeme nástroj `adb`. Ten je nainstalován ve složce `platform-tools`, která se nachází v adresáři s Android SDK. Pro navigování do tohoto adresáře využijeme příkaz:

```
cd C:\Users\<Uživatelské jméno>\AppData\Local\Android\Sdk  
\platform-tools
```

Jelikož komunikujeme s naším mobilním zařízením, je pro úspěšné provedení příkazu nutné, aby bylo zařízení zapnuté a připojené. Následujícím příkazem přeneseme stažený soubor serveru z jeho adresáře do adresáře `/data/local/tmp` na mobilním zařízení (výstup lze vidět na obr. 5.7):

```
adb.exe push <Adresář souboru frida-server> /data/local/tmp
```

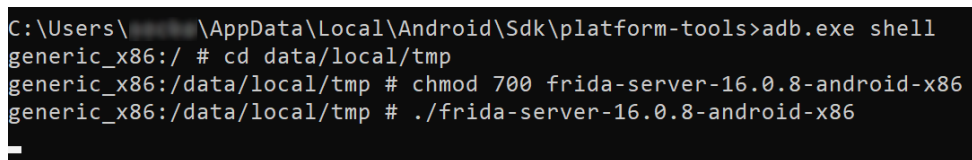


```
C:\Users\... \AppData\Local\Android\Sdk\platform-tools>adb.exe push C:\Users\... \Downloads\frida-server-16.0.8-android-x86 /data/local/tmp  
C:\Users\... \Downloads\frida-server-16.0.8-android-x86: 1 file pushed, 0 skipped, 33.4 MB/s (53604060 bytes in 1.531s)  
C:\Users\... \AppData\Local\Android\Sdk\platform-tools>
```

Obrázek 5.7: Výstup příkazu `adb.exe push` [29].

Jako poslední musíme přesunout samotnou konzoli do zařízení, udělit serveru oprávnění ke spuštění a nakonec jej spustit. K tomu slouží následující série příkazů:

1. `adb.exe shell`
2. `cd data/local/tmp`
3. `chmod 700 <název souboru frida-server>`
4. `./<název souboru frida-server>`



```
C:\Users\... \AppData\Local\Android\Sdk\platform-tools>adb.exe shell  
generic_x86:/ # cd data/local/tmp  
generic_x86:/data/local/tmp # chmod 700 frida-server-16.0.8-android-x86  
generic_x86:/data/local/tmp # ./frida-server-16.0.8-android-x86
```

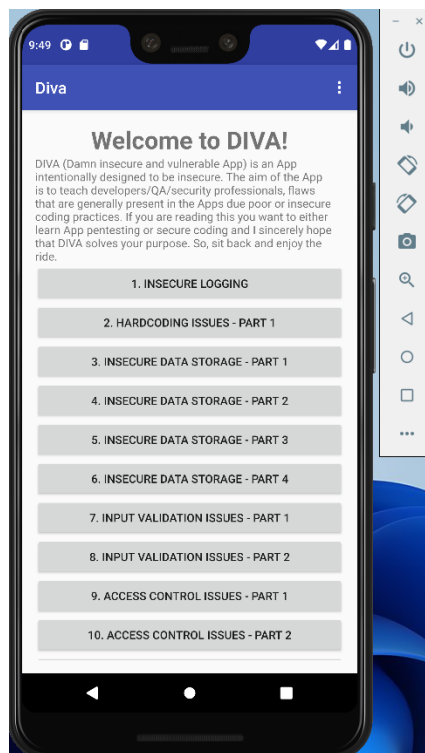
Obrázek 5.8: Provedení série příkazů sloužící ke spuštění serveru na emulátoru [29].

Výstupy této sekvence příkazů lze vidět na obr. 5.8. Poslední příkaz nemá výstup, protože server běží a konzole je tak zaseklá. Při demonstraci zranitelnosti absence binární protekce bude třeba mít server zapnutý a konzoli nevypínat. Místo toho pro potřebné příkazy otevřeme nové okno konzole. Tento postup se může na poprvé zdát někomu složitý, nejde však o nic jiného než jen o spuštění série správných příkazů za sebou. Nyní je už vše připraveno a jsme připraveni k demonstraci všech probíraných zranitelností.

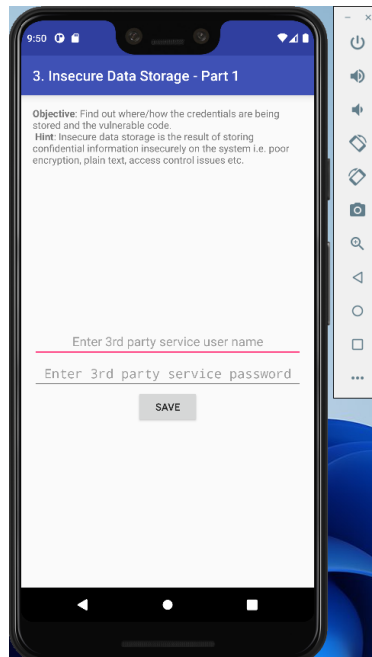
5.2 DEMONSTRACE: ZRANITELNOST NEZABEZPEČENÝCH DATOVÝCH ÚLOŽIŠŤ

První ze zranitelností je asi nejjednodušší ke zneužití, alespoň v tomto případě. Cílem je objevit zranitelný kód, který ukládá citlivá data špatným způsobem do veřejného úložiště (v tomto případě jde o sdílené preference). Po nalezení zranitelnosti ji stačí jen otestovat; spustit špatný kód a ověřit, že se do sdílených preferencí citlivá data opravdu zapíše. K demonstraci zranitelnosti bude využita zranitelná aplikace *DIVA* (*Damn Insecure and Vulnerable app*). *DIVA* je aplikace publikována na platformě GitHub uživatelem *0xArab* a jedná se o zranitelnou aplikaci obsahující širokou škálu zranitelností pro jejich demonstraci. Aplikační soubor je k dispozici na stránce projektu: <https://github.com/0xArab/diva-apk-file> (odkaz je opět uveden v souboru links.txt, který je součástí elektronické přílohy). Pro instalaci aplikace na zařízení stačí přetáhnout soubor z počítače do emulátoru, a aplikace se automaticky nainstaluje.

Pokud aplikaci otevřeme, zobrazí se nám úvodní obrazovka aplikace s mnoha úlohami, které v ní lze splnit (viz. obr. 5.9). Pro tuto demonstraci se budeme věnovat úloze 3. *Insecure Data Storage – Part 1*. Když tuto možnost rozklikneme, otevře se nám aktivita obsahující dvě pole pro vstup, tlačítko *SAVE* a popis úlohy (viz. obr. 5.10). Pole pro vstup zpracovávají dvě informace – uživatelské jméno a heslo. Toto budou zjevně informace, které budou ukládány bez zabezpečení.



Obrázek 5.9: Úvodní obrazovka aplikace DIVA s mnoha úlohami k řešení [29].



Obrázek 5.10: Aktivita pro úlohu nezabezpečených datových úložišť [29].

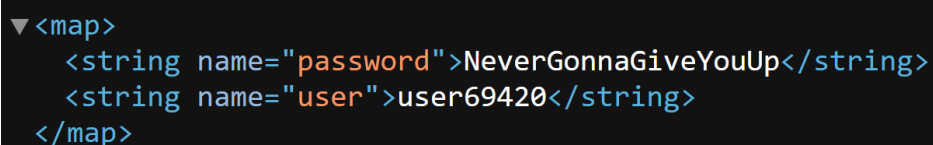
Když jsme získali základní povědomí o aktivitě, která zpracovává data špatným způsobem, můžeme se pokusit zjistit jak funguje do detailů prozkoumáním kódu. K tomu je ale třeba kód nejprve dekompileovat. Můžeme si pro to stáhnout nějaký nástroj, anebo využít nástroje online. Pro demonstraci jsem zvolil internetový nástroj *decompiler.com*. Ten .apk soubor po nahrání rychle dekompile a my jej můžeme detailně prozkoumat. Kódy jednotlivých aktivit se nachází v adresáři `/sources/jakhar/aseem/diva`. Jelikož jsou názvy souborů pojmenované podle příslušných aktivit, nalézt kód pro zkoumanou aktivitu netvoří problém. Po otevření souboru `InsecureDataStorage1Activity.java` se nám zobrazí celý její čitelný kód.

Jako první by nás měla zaujmout metoda `saveCredentials` (viz. obr. 5.11), jelikož již podle názvu je zodpovědná za ukládání přihlašovacích údajů. Z kódu lze vyčíst, že metoda volá sdílené preference a ukládá do nich vstupy z obou polí pod klíčem „user“ a „password“. Podstatná informace je také, že data ukládá v prostém textu, bez jakéhokoliv šifrování. Můžeme tedy vyvodit závěr, že zadané uživatelské údaje jsou ukládány do sdílených preferencí, a tedy bychom měli být schopni je tam najít. To je vše, co potřebujeme vědět. Podařilo se nám nalézt zranitelný kód a teď naši teorii stačí jen otestovat.

```
public void saveCredentials(View view) {
    SharedPreferences.Editor spedit = PreferenceManager.getDefaultSharedPreferences(this).edit();
    spedit.putString("user", ((EditText) findViewById(R.id.ids1Usr)).getText().toString());
    spedit.putString("password", ((EditText) findViewById(R.id.ids1Pwd)).getText().toString());
    spedit.commit();
    Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
}
```

Obrázek 5.11: Metoda `saveCredentials()` zodpovědná za ukládání přihlašovacích údajů [29].

Vrátíme se zpět do aplikace a vyplníme textová pole libovolnými hodnotami. Po kliknutí na tlačítko *SAVE* aplikace vrátí hlášku, že údaje byly úspěšně uloženy, a my bychom tak měli být schopni je nalézt v adresáři sdílených preferencí. Abychom jej mohli číst, musíme v Android Studiu otevřít záložku *Device File Explorer*. Otevře se průzkumník souborů, pomocí něhož lze procházet veškerá data v zařízení. Soubory aplikace DIVA se nachází v adresáři `data/data/jakhar.aseem.diva/`, přičemž ve složce `shared_prefs` je uložen hledaný soubor (soubor `jakhar.aseem.diva_preferences.xml`). Po otevření souboru přímo v Android Studiu můžeme vidět, že v souboru jsou skutečně vypsané uživatelské údaje v prostém textu bez jakéhokoliv zabezpečení (viz obr. 5.12). V aplikaci DIVA se jedná samozřejmě za záměrně vytvořenou chybu, avšak pokud by podobnou chybu obsahovala například bankovní aplikace, potenciální důsledky by mohly být enormního rozsahu.

A screenshot of a code editor with a dark background. It displays XML code for a preferences file. The code is as follows:

```
▼ <map>
  <string name="password">NeverGonnaGiveYouUp</string>
  <string name="user">user69420</string>
</map>
```

Obrázek 5.12: Soubor sdílených preferencí. V prostém textu je zde vypsáno uživatelské jméno (*user69420*) a heslo (*NeverGonnaGiveYouUp*) [29].

5.3 DEMONSTRACE: ZRANITELNOST EXPORTOVANÉ AKTIVITY

Zranitelnost exportované aktivity je na rozdíl od zranitelnosti nezabezpečených datových úložišť ta nejlehčí na objevení z probíraných tříd. Jak bylo již popsáno v sekci 4.2, v podstatě se stačí podívat do souboru `AndroidManifest.xml` a zkontrolovat, zda některá z aktivit není definována jako exportovaná. Cílem demonstrace je spustit exportovanou aktivitu (pokud nějakou nalezneme) a získat tak přístup k jinak nedostupné části aplikace. Pro demonstraci této zranitelnosti nám nyní poslouží aplikace *InsecureBankv2*. Opět se jedná o zranitelnou aplikaci, avšak na rozdíl od aplikace DIVA nemá definované úlohy pro demonstraci zranitelností, ale snaží se spíše předstírat skutečnou bankovní aplikaci. Obsahuje znovu veliké množství různých zranitelností a stejně jako DIVA byla publikována na platformě *github.com*, tentokrát uživatelem *dineshshetty*. Pro stažení je stejně jako u aplikace DIVA třeba nalézt `.apk` soubor a stáhnout jej, pro instalaci znovu stačí pouze přetáhnout myší soubor z počítače na zařízení.

Po otevření aplikace uvidíme úvodní obrazovku vyzývající nás k přihlášení (obr. 5.13). Správné přihlašovací údaje, které nás pustí do aplikace jsou uvedeny na stránce projektu, my je ale nebudeme potřebovat. Pokud aplikaci dekompilujeme, ve složce `resources` se nachází soubor `AndroidManifest.xml`. V něm nalezneme mimo jiné informace o aktivitách, které jsou exportované (viz. obr. 5.14). Tato aplikace takových aktivit obsahuje vcelku mnoho, máme tedy na výběr, kterou z nich se můžeme pokusit exploítovat. Já jsem zvolil aktivitu `.PostLogin`, poněvadž se evidentně jedná o aktivitu, která následuje po přihlášení. Pokud se nám ji podaří otevřít, získáme tím přístup do aplikace, aniž bychom znali přihlašovací údaje.



Obrázek 5.13: Úvodní obrazovka aplikace InsecureBankv2 [29].

```
securebankv2.PostLogin" android:exported="true"/>
securebankv2.WrongLogin"/>
securebankv2.DoTransfer" android:exported="true"/>
.insecurebankv2.ViewStatement" android:exported="true"/>
```

Obrázek 5.14: Exportované aktivity aplikace insecurebankv2 [29].

Pro exploitaci exportované aktivity `.PostLogin` využijeme nástroj `adb`, který byl nainstalován zároveň s Android Studiem. Jak bylo zmíněno v sekci 5.1, pro jeho spuštění se musíme přesunout do adresáře, kde se nachází následujícím příkazem:

```
cd C:\Users\<Uživatelské jméno>\AppData\Local\Android\Sdk\platform-tools
```

Nyní už je vše připraveno a můžeme se pustit do exploitace. Jako první je třeba se ujistit, že emulátor běží a je možné se k němu připojit. Příkazem

```
adb.exe devices
```

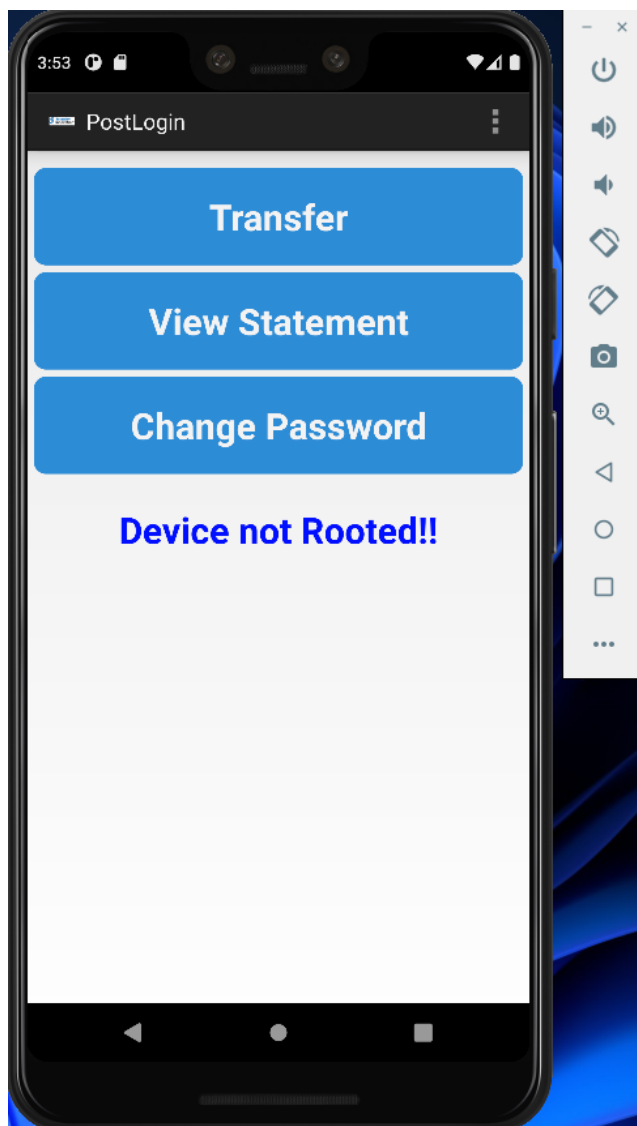
```
C:\Users\... \AppData\Local\Android\Sdk\platform-tools>adb.exe devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
emulator-5554    offline
```

Obrázek 5.15: Výstup příkazu `adb.exe devices` [29].

zjistíme, která zařízení jsou připojená. Pokud vše funguje jak má, měl by výstup vypadat podobně, jako na obrázku 5.15. Výstup může být ovlivněný mnoha faktory, kvůli kterým se může mírně lišit. Klíčové je, že je v seznamu zařízení zobrazené zařízení `emulator-5554`. Dalším krokem je se k zařízení připojit. K tomu opět využijeme nástroj `adb`, tentokrát s příkazem `shell`. To naši konzoli přenese do virtuálního zařízení. Veškeré příkazy které provedeme, budou provedeny přímo na něm. Tím, že je Android postavený na Linuxovém jádru, příkazy, které bychom chtěli zadat musí být v linuxové syntaxi. My ale stále pracujeme s `adb`, takže nás to nemusí příliš zatěžovat. Nyní už zbývá poslední krok, a to spustit exportovanou aktivitu. To zařídí následující příkaz:

```
am start com.android.insecurebankv2/.PostLogin
```

Příkaz `am` stojí pro *activity manager* a jak název vypovídá, slouží pro správu aktivit. Dále následuje pokyn, že chceme aktivitu spustit a název balíčku aplikace doprovazený znakem „/“ a názvem aktivity. Pokud byl příkaz zadán správně, když znovu otevřeme emulátor, měli bychom nalézt otevřenou aktivitu `PostLogin`, která by se nám jinak zpřístupnila až po zadání správných přihlašovacích údajů (viz. obr. 5.16). Takto jsme získali přístup k aktivitě následující až po přihlášení i bez nich. Z této aktivity se můžeme navíc pomocí tlačítek v ní pohybovat do jiných aktivit a provádět operace, které by měl správně provádět pouze autorizovaný uživatel. Získali jsme tak přístup k téměř celé aplikaci.

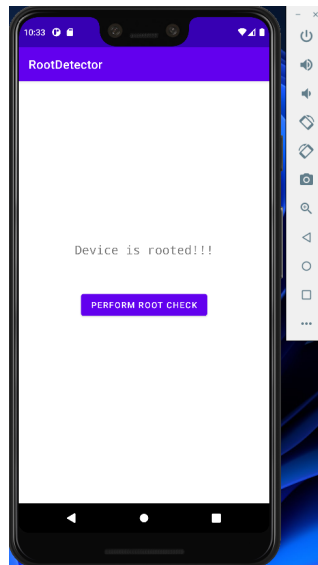


Obrázek 5.16: Aktivita .PostLogin, otevřená pomocí exploitování zranitelnosti exportované aktivity [29].

5.4 DEMONSTRACE: ABSENCE BINÁRNÍ PROTEKCE

Poslední ze tří demonstrovaných zranitelností je absence binární protekce. Původně měla být k demonstraci použita znovu aplikace `insecurebankv2`, protože i ta obsahuje root detekci (v aktivitě `.PostLogin` je možné vidět nápis „Device not rooted!!“). Ta se ukázala být nefunkční, poněvadž i bez použití jakýchkoliv maskovacích nástrojů aplikace náš emulátor neshledala jako rootovaný. Vytvořil jsem proto pro účely demonstrace jednoduchou aplikaci `RootDetector`, která otestuje, zda zařízení běží na emulátoru. Lze ji, stejně jako předešlé aplikace, stáhnout z příslušného repozitáře na platformě *github.com* nebo je také součástí elektronické přílohy. Cílem demonstrace bude aplikaci přesvědčit, že aplikace na emulátoru neběží. Simuluje případ, kdy aplikace obsahuje detekci emulátoru, avšak lze ji snadno nalézt a změnit její výstup k našemu prospěchu.

Otevřeme-li aplikaci na našem emulátoru, vidíme, že se skládá pouze z jediné aktivity. Ta obsahuje jenom text se stavem zařízení a tlačítko pro spuštění testu. Po kliknutí na tlačítko aplikace provede jednoduchou kontrolu, která ověří, zda některé hodnoty systémových proměnných odpovídají hodnotám typickým pro běžně používané emulátory. Na základě výsledku se poté změní hodnota textu. Pokud mobilní zařízení běží na emulátoru, vypíše se „Device is rooted!!!“, v opačném případě se vypíše „Device not rooted.“.



Obrázek 5.17: Výstup aplikace `RootDetector` při spuštění testu na emulátoru [29].

Z výstupu aplikace na obrázku 5.17 víme, že aplikace skutečně obsahuje detekci emulátoru a naše zařízení úspěšně rozpoznala. K vypnutí této funkce je klíčové ji nalézt v kódu a zjistit, jak funguje. Poněvadž aplikace obsahuje pouze jednu aktivitu s minimální funkcionalitou, nebude těžké jej nalézt. Po dekompilování aplikace se soubor s kódem aktivity nachází v adresáři

`RootDetector.apk/sources/com/cyphertech/rootdetector/MainActivity.java`.

```

public class MainActivity extends AppCompatActivity {
    TextView textView;

    /* access modifiers changed from: protected */
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView((int) R.layout.activity_main);
    }

    public void performCheck(View view) {
        this.textView = (TextView) findViewById(R.id.mainText);
        boolean checkBuildConfig = checkBuildConfig();
        if (checkBuildConfig) {
            this.textView.setText("Device is rooted!!!");
        } else {
            this.textView.setText("Device not rooted.");
        }
        Toast.makeText(getApplicationContext(), "Root check performed!", 0).show();
        Log.d("ROOTCHECK", "Root check performed. Result: " + checkBuildConfig);
    }

    private boolean checkBuildConfig() {
        return Build.MANUFACTURER.contains("Genymotion") || Build.MODEL.contains("google_sdk") || Build.MODEL.toLowerCase().contains("droid4x") || Build.MODEL.
    }
}

```

Obrázek 5.18: Dekompilovaný kód aplikace RootDetector [29].

Na obrázku 5.18 lze vidět kompletní kód aktivity. Metoda `onCreate()` je pouze načítací metoda aktivity, která zobrazí příslušný obsah obrazovky. Pod ní se však nachází dvě další metody. První z nich, metoda `performCheck()` má parametr `public`, což naznačuje její využití mimo script. Zde se tedy pravděpodobně jedná o funkci tlačítka. Metoda spouští druhou metodu `checkBuildConfig()`, která vrací proměnnou booleovského typu (pravda/nepravda) a na základě výsledku mění text na obrazovce. V závěru metody pak ještě zobrazí plovoucí text s potvrzením operace a do konzole vypíše výsledek testu. Vidíme tedy, že výsledek testu závisí zcela na výsledku volání metody `checkBuildConfig()`. Ta testuje mnoho konfiguračních parametrů zařízení a pakliže alespoň jeden z nich odpovídá parametru typickému pro emulátor, vrátí kladný výsledek. My však potřebujeme, aby vrácený výsledek byl vždy záporný. Zde přichází na scénu nástroj Frida a její schopnost vkládat vlastní kód při běhu aplikace. Ještě předtím, než spustíme proces exploity, musíme mít exploit. V našem případě se jedná o kód napsaný v jazyce JavaScript, který do metody `checkBuildConfig()` vloží naši vlastní funkcionalitu. Tento kód je součástí jak repozitáře aplikace na platformě *github.com*, tak elektronické přílohy.

```

1 console.log("Script loaded!");
2 Java.perform(function(){
3     var mainapp = Java.use("com.cyphertech.rootdetector.MainActivity");
4     mainapp.checkBuildConfig.implementation = function(){
5         console.log("Injection successful!");
6         var ret = false;
7         return ret;
8     };
9     send("Rootdetectioncheckresultsasfalse-ovator!");
10 });
11

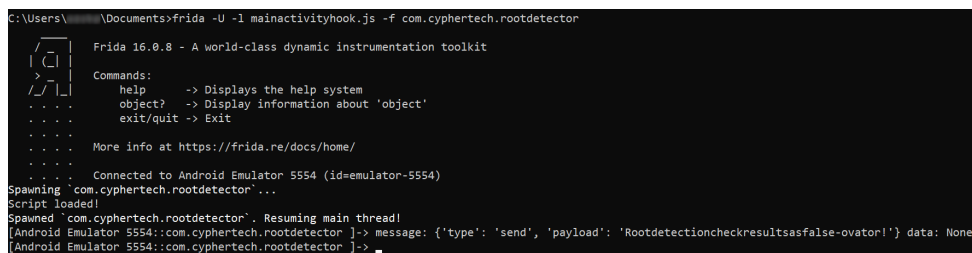
```

Obrázek 5.19: Kód `mainactivityhook.js` sloužící k implementaci vlastní funkcionality do metody `checkBuildConfig()` [29].

Jednoduchý kód na obrázku 5.19 nejprve načte konkrétní aktivitu, a poté do její konkrétní funkce naimplementuje funkci novou. V našem případě script nejprve odešle hlášení do konzole Fridy s obsahem „Injection successful!“ a poté vrátí hodnotu `false`. To je koneckonců vše, co po metodě `checkBuildConfig()` chceme, aby aplikaci vrátila zápornou hodnotu. Když už máme vše připravené, můžeme se pustit do exploity. K jejímu

provedení je důležité se ujistit, že je Frida správně nainstalována a její server na mobilním zařízení běží. Pokud ano, v novém okně konzole se přesuneme do adresáře s exploitem a spustíme následující příkaz, který vloží script `mainactivityhook.js` do kódu aplikace RootDetector:

```
frida -U -l mainactivityhook.js -f com.cyphertech.rootdetector
```



```
C:\Users\...\Documents>frida -U -l mainactivityhook.js -f com.cyphertech.rootdetector

Frida 16.0.8 - A world-class dynamic instrumentation toolkit

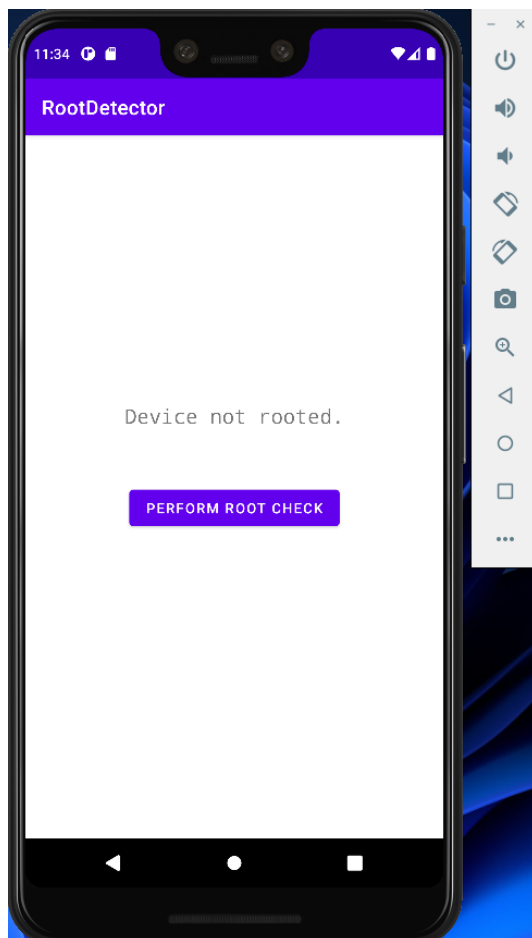
Commands:
  help      -> Displays the help system
  object?   -> Display information about 'object'
  exit/quit -> Exit

More info at https://frida.re/docs/home/

Connected to Android Emulator 5554 (id=emulator-5554)
Spawning 'com.cyphertech.rootdetector'...
Script loaded!
Spawned 'com.cyphertech.rootdetector'. Resuming main thread!
[Android Emulator 5554::com.cyphertech.rootdetector ]-> message: {'type': 'send', 'payload': 'Rootdetectioncheckresultsasfalse-ovator!'} data: None
[Android Emulator 5554::com.cyphertech.rootdetector ]->
```

Obrázek 5.20: Výstup příkazu pro vložení exploitu `mainactivityhook.js` do aplikace RootDetector [29].

Výstupem tohoto příkazu (viz. obr. 5.20) není třeba se příliš zabývat, důležité je pro nás ale hlášení „Script loaded!“, které značí, že náš kód byl úspěšně spuštěn. V emulátoru by se měla zatím spustit aplikace RootDetector s již upraveným kódem. Pokud klikneme na tlačítko *Perform root check*, text na obrazovce by se měl změnit na hodnotu „Device not rooted“ (viz. obr. 5.21). Vše tedy naznačuje tomu, že exploit fungoval. Pro jistotu se ještě můžeme ujistit zkontrolováním výstupu aplikace v záložce *Logcat* v Android Studiu (viz. obr. 5.22). I zde je vypsán výsledek jako negativní. Exploitace byla úspěšná, podařilo se nám změnit chování aplikace tak, abychom obešli detekci emulátoru. Pokud by se jednalo o aplikaci, která například na zařízeních s rootem blokuje veškerou funkcionalitu, podobným postupem bychom byli schopni aplikaci spustit i na emulátoru a detailně ji zkoumat.



(a)

```
[Android Emulator 5554::com.cyphertech.rootdetector ]-> Injection successful!  
[Android Emulator 5554::com.cyphertech.rootdetector ]-> █
```

(b)

Obrázek 5.21: Výstup aplikace RootDetector (a) a nástroje Frida (b) po úspěšné aktivaci exploitu [29].

```
D/ROOTCHECK: Root check performed. Result: false
```

Obrázek 5.22: Aplikace RootDetector vypsala do konzole výsledek testu jako negativní [29].

ZÁVĚR

Tato práce se zabývá třemi zranitelnostmi běžně se vyskytujícími v mobilních aplikacích na zařízeních s operačním systémem Android. V prvních kapitolách byl čtenář uveden do tématu. Získal základní informace o organizaci OWASP a jejích dokumentech. V kapitole 2 mu byl vysvětlen princip bezpečnostních mechanismů operačního systému Android a jejich cíle. Ve třetí kapitole byla rozebrána metoda ověřování bezpečnosti aplikací, tzv. penetrační testování. Detailně byl popsán jeho postup a jednotlivé fáze, a také jaké očekávané výsledky penetračního testování. V kapitole 4 byly pak do detailů rozebrány jednotlivé zranitelnosti. Čtenář se dozvěděl, co každou z nich způsobuje, jak ji odhalit, exploitovat a především jak se jí vyvarovat. Také se dozvěděl o jejich výskytu a dopadu v reálném světě. Pro ještě lepší pochopení jednotlivých zranitelností byly pak v praktické části všechny probírané zranitelnosti demonstrovány na jednoduchých ukázkových příkladech. Veškeré postupy včetně přípravy prostředí byly popsány tak, aby čtenář neměl problém si zranitelnosti vyzkoušet sám. V poslední kapitole byl vytvořen návrh výukové hry zaměřující se na další demonstraci vybraných zranitelností, a současně s ním byla také navržena a vyvinuta mobilní aplikace, na které by v rámci hry probíhaly veškeré testy. Její aktivity a jejich funkcionalita byly stručně popsány. Celý návrh hry byl plně zrealizován a naimplementován do prostředí Kybernetické arény na Fakultě elektrotechniky a komunikačních technologií Vysokého učení technického v Brně. Může zde sloužit nejen studentům VUT, ale také žákům okolních středních škol ke vzdělávání se v oblasti mobilní kyberbezpečnosti zábavnou cestou.

Cíle práce byly tedy úspěšně naplněny. Mimo návrhu výukové hry se čtenář také dozvěděl detailní podrobnosti o rozebíraných zranitelnostech a zároveň získal základní nadhled do oblasti mobilní kyberbezpečnosti. Získané informace může využít při svém dalším studiu v této oblasti nebo je může využít k zabezpečení svých vlastních aplikací. Věřím také, že práce namotivuje některé čtenáře k hlubšímu zájmu o oblast kyberbezpečnosti a přispěje tak ke zvýšení kvality penetračních testerů v České republice.

LITERATURA

- [1] About sublimelinter. [cit. 25.1.2023].

URL <https://www.sublimelinter.com/en/v3.10.10/about.html>

- [2] About the Owasp Foundation. [cit. 25.1.2023].

URL <https://owasp.org/about/>

- [3] <Activity>. [cit. 25.1.2023].

URL <https://developer.android.com/guide/topics/manifest/activity-element>

- [4] Android security: Adding tampering detection to your app - airpair. [cit. 25.1.2023].

URL <https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app>

- [5] Application sandbox: Android Open Source Project. [cit. 25.1.2023].

URL <https://source.android.com/docs/security/app-sandbox>

- [6] Application security: Android Open Source Project. [cit. 25.1.2023].

URL <https://source.android.com/docs/security/overview/app-security>

- [7] Back up User Data with auto backup: android developers. [cit. 25.1.2023].

URL <https://developer.android.com/guide/topics/data/autobackup>

- [8] Cross site scripting (XSS). [cit. 25.1.2023].

URL <https://owasp.org/www-community/attacks/xss/>

- [9] Data storage on Android. [cit. 25.1.2023].

URL <https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05d-testing-data-storage>

- [10] DRD23. do not use world readable or writeable to share files between apps. [cit. 25.1.2023].

URL <https://wiki.sei.cmu.edu/confluence/display/android/DRD23.+Do+not+use+world+readable+or+writeable+to+share+files+between+apps>

- [11] Intent: android developers. [cit. 25.1.2023].
URL <https://developer.android.com/reference/android/content/Intent>
- [12] Intents and intent filters: android developers. [cit. 25.1.2023].
URL <https://developer.android.com/guide/components/intents-filters>
- [13] Introduction to activities: android developers. [cit. 25.1.2023].
URL <https://developer.android.com/guide/components/activities/intro-activities>
- [14] M1: Improper platform usage. [cit. 25.1.2023].
URL <https://owasp.org/www-project-mobile-top-10/2016-risks/m1-improper-platform-usage>
- [15] M10: Lack of binary protections. [cit. 25.1.2023].
URL <https://owasp.org/www-project-mobile-top-10/2014-risks/m10-lack-of-binary-protections>
- [16] M2: Insecure data storage. [cit. 25.1.2023].
URL <https://owasp.org/www-project-mobile-top-10/2016-risks/m2-insecure-data-storage>
- [17] Meet android studio: Android developers. [cit. 25.1.2023].
URL <https://developer.android.com/studio/intro>
- [18] Offensive security's Exploit Database Archive. [cit. 25.1.2023].
URL <https://www.exploit-db.com/>
- [19] Optimizing and securing Android applications with R8. [cit. 25.1.2023].
URL <https://www.section.io/engineering-education/optimizing-and-securing-android-applications-with-r8/>
- [20] Owasp MASVS. [cit. 25.1.2023].
URL <https://mas.owasp.org/MASVS/>
- [21] Owasp Mobile Application Security. [cit. 25.1.2023].
URL <https://owasp.org/www-project-mobile-app-security/>
- [22] Owasp Top Ten. [cit. 25.1.2023].
URL <https://owasp.org/www-project-top-ten/>
- [23] <permission>. [cit. 25.1.2023].

- URL <https://developer.android.com/guide/topics/manifest/permission-element>
- [24] Permissions on Android: android developers. [cit. 25.1.2023].
- URL <https://developer.android.com/guide/topics/permissions/overview>
- [25] PMD. [cit. 25.1.2023].
- URL <https://pmd.github.io/>
- [26] Sign your app: android developers. [cit. 25.1.2023].
- URL https://developer.android.com/studio/publish/app-signing#sign_release
- [27] System and kernel security: Android Open Source Project. [cit. 25.1.2023].
- URL <https://source.android.com/docs/security/overview/kernel-security>
- [28] U166A: Průběh penetračního testování. [cit. 25.1.2023].
- URL <https://mbi.vse.cz/public/cs/obj/TASK-362>
- [29] Zdroj: autor.
- [30] Malware disguised as Uber's Android app steals your login details. Leden 2018, [cit. 25.1.2023].
- URL <https://beebom.com/malware-uber-android-app-steals-data/>
- [31] Android: Access to App Protected Components. Srpen 2020, [cit. 25.1.2023].
- URL <https://blog.oversecured.com/Android-Access-to-app-protected-components/>
- [32] Android obfuscation tools comparison. Únor 2021, [cit. 25.1.2023].
- URL <https://riis.com/blog/android-obfuscation-proguard-dexguard/>
- [33] How to implement root detection in Android applications. Květen 2021, [cit. 25.1.2023].
- URL <https://www.indusface.com/learning/how-to-implement-root-detection-in-android-applications/>
- [34] Android Root Detection Bypass using Frida. Červenec 2022, [cit. 25.1.2023].
- URL <https://redfoxsec.com/blog/android-root-detection-bypass-using-frida/>
- [35] What is Magisk: Everything you need to know. Leden 2022, [cit. 25.1.2023].
- URL <https://themagisk.com/what-is-magisk/>

- [36] What is malware? - definition and examples. Červen 2022, [cit. 25.1.2023].
URL <https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html>
- [37] ALDER, S.: Quest Diagnostics 2016 data breach settlement receives final approval. Březen 2020, [cit. 25.1.2023].
URL <https://www.hipaajournal.com/quest-diagnostics-2016-data-breach-settlement-receives-final-approval/>
- [38] COMSTOCK, J.: Quest Diagnostics' mobile app compromised in breach of 34,000 records. Prosinec 2016, [cit. 25.1.2023].
URL <https://www.mobihealthnews.com/content/quest-diagnostics-mobile-app-compromised-breach-34000-records>
- [39] Droidcon: Unpacking android security: Part 1 - improper platform usage. Únor 2022, [cit. 25.1.2023].
URL <https://www.droidcon.com/2022/02/15/unpacking-android-security-part-1-improper-platform-usage/>
- [40] FENTON, C.: Calebfenton/Simplify: Android Virtual Machine and deobfuscator. [cit. 25.1.2023].
URL <https://github.com/CalebFenton/simplify>
- [41] JOSEPH, J.: The history of the Owasp Mobile top 10 and what changes mean to developers. Říjen 2020, [cit. 25.1.2023].
URL <https://community.guardsquare.com/t/the-history-of-the-owasp-mobile-top-10-and-what-changes-mean-to-developers/276>
- [42] Kunsh: Android root detection bypass via code-tempering and prevention. Červenec 2022, [cit. 25.1.2023].
URL <https://kunshdeep.in/android-root-detection-bypass-via-/code-tempering-and-prevention/>
- [43] LUTKEVICH, B.: What is obfuscation and how does it work? Duben 2021, [cit. 25.1.2023].
URL <https://www.techtarget.com/searchsecurity/definition/obfuscation>
- [44] MISRA, A.; DUBEY, A.: *Android security*. London, England: CRC Press, 2019, [cit. 25.1.2023].
- [45] pxb1988: PXB1988/dex2jar: Tools to work with Android .dex and java .class files. [cit. 25.1.2023].
URL <https://github.com/pxb1988/dex2jar>

- [46] Skylot: Skylot/jadx: Dex to java decompiler. [cit. 25.1.2023].
URL <https://github.com/skylot/jadx>
- [47] SUMMERSON, C.: What is "systemless root" on Android, and why is it better? Červenec 2017, [cit. 25.1.2023].
URL <https://www.howtogeek.com/249162/what-is-systemless-root-on-android-and-why-is-it-better/>
- [48] Technologies, P.: Vulnerabilities and threats in mobile applications, 2019. Leden 2021, [cit. 25.1.2023].
URL <https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/>
- [49] TOSHIN, S.: Slack disclosed on Hackerone: Access of Android protected components via embedded intent. Leden 2017, [cit. 25.1.2023].
URL <https://hackerone.com/reports/200427>
- [50] TURNER, A.: How many people have smartphones worldwide. Říjen 2022, [cit. 25.1.2023].
URL <https://www.bankmycell.com/blog/how-many-phones-are-in-the-world>
- [51] UPSHAM, A.: Exploiting android activity «activity android:exported="true"». Duben 2021, [cit. 25.1.2023].
URL <https://aupsham98.medium.com/exploiting-android-activity-activity-android-exported-true-93ffeb263682>
- [52] WELLS, R. E.: Learn what an emulator is in the world of computing. Leden 2021, [cit. 25.1.2023].
URL <https://www.lifewire.com/what-is-an-emulator-4687005>

SEZNAM OBRÁZKŮ

2.1	Znázornění funkce jádra v rámci systému.	6
3.1	Ukázka využití linteru v prostředí Android Studio.	9
3.2	Zdrojový kód aplikace dekompilovaný nástrojem dex2jar.	10
4.1	Ukázka kódu se zranitelností nezabezpečených datových úložišť.	15
4.2	Ukázka nešifrovaného a šifrovaného souboru sdílených preferencí.	16
4.3	Exportovaná aktivita v souboru AndroidManifest.xml.	22
4.4	Schéma procesu exploitace exportované aktivity skrze vnořený intent. . . .	23
4.5	Ukázka obfuskovaného a neobfuskovaného zdrojového kódu.	27
5.1	Úvodní obrazovka programu Android Studio.	34
5.2	Detaily prázdného projektu.	34
5.3	Možnosti spuštění správce virtuálních zařízení.	35
5.4	Konfigurace vytvářeného virtuálního zařízení.	36
5.5	Spuštěný emulátor pomocí Android Studia.	36
5.6	Výstup příkazu <code>frida --version</code>	37
5.7	Výstup příkazu <code>adb.exe push</code>	38
5.8	Série příkazů pro spuštění serveru Fridy na emulátoru.	38
5.9	Úvodní obrazovka aplikace DIVA s množstvím úloh.	39
5.10	Aktivita pro úlohu nezabezpečených datových úložišť.	40
5.11	Metoda <code>saveCredentials()</code>	40
5.12	Soubor sdílených preferencí.	41
5.13	Úvodní obrazovka aplikace InsecureBankv2.	42
5.14	Exportované aktivity aplikace InsecureBankv2.	43
5.15	Výstup příkazu <code>adb.exe devices</code>	43
5.16	Aktivita <code>.PostLogin</code>	44
5.17	Výstup aplikace RootDetector při spuštění testu na emulátoru.	45
5.18	Dekompilovaný kód aplikace RootDetector.	46
5.19	Kód exploitu <code>mainactivityhook.js</code>	46
5.20	Konzolový výstup příkazu pro aplikaci exploitu.	47
5.21	Výstup aplikace po aktivaci exploitu.	48
5.22	Výsledek testu v logu vypsán jako negativní.	48