FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Využití zranitelnosti Janus na operačním systému Android
Student:	Bc. Vít Souček
Vedoucí:	Ing. Filip Štěpánek
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Janus (CVE-2017-13156) je zranitelnost vyskytující se v OS Android. Jejím využitím lze obejít kontrolu podpisu aplikace a spustit kód útočníka. Cílem práce je navrhnout a implementovat útok, který díky zranitelnsti Janus umožní do zařízení oběti nainstalovat legitimní aplikaci obsahující škodlivý kód útočníka. Ten bude cílit na narušení soukromí oběti (vzdálený odposlech).

Práci rozdělte na následovně:
1) Analyzujte současný stav útoku pomocí zranitelnosti Janus (principy, obrana, známé útoky, dopad na stávající zařízení, omezení pro útočníka).
2) Navrhněte průběh útoku:
- modul Android aplikace vykonávající odposlech,
- úpravu reálné Android aplikace tak, aby spouštěla modul z předchozího bodu,
- program pro příjem sebraných dat na zařízení útočníka,
- program upravující instalační balíček reálné aplikace tak, aby místo ní bylo díky zranitelnosti Janus možné nainstalovat škodlivou aplikaci.
3) Útok implementujte, otestujte, demonstrujte na reálném zařízení a vyhodnoťte výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 22. ledna 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Využití zranitelnosti Janus na operačním systému Android

Bc. Vít Souček

Katedra informační bezpečnosti
Vedoucí práce: Ing. Filip Štěpánek

20. května 2020

Poděkování

Předně děkuji vedoucímu této práce, Filipu Štěpánkovi, za to, že vymyslel toto zajímavé a zábavné téma a také za to, že věnoval tolik svého volného času konzultacím v průběhu tvorby této práce. Dále děkuji Jitce Veselé za poskytnutí ukázkově zranitelného zařízení, na kterém jsem výsledek své práce mohl demonstrovat. Děkuji samozřejmě také celé své rodině za podporu při studiu a poskytnutí ideálních podmínek pro psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 20. května 2020 .....

Odkaz na tuto práci

Souček, Vít. *Využití zranitelnosti Janus na operačním systému Android*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

CVE-2017-13156 („Janus“) je zranitelnost aplikačního virtuálního stroje Android Runtime. Práce se zabývá návrhem a uskutečněním útoku na fyzické zařízení. Text rozebírá principy CVE-2017-13156, návrh škodlivého kódu, jeho distribuci a možnosti útočníka cílit na více zařízení najednou. Výsledkem je škodlivá aplikace, která umožňuje vzdálený odposlech více Android zařízení.

Klíčová slova Android, CVE-2017-13156, Janus, Android Runtime, Android Package APK, APK Signature Scheme v1, Smali, elektronický podpis, nekontrolovaná instalace souboru nebezpečného typu, vzdálený odposlech, narušení soukromí, spyware

Abstract

CVE-2017-13156 (“Janus”) is a vulnerability of Android Runtime, an application virtual machine used in Android OS. This thesis focuses on the design and implementation of an attack exploiting this vulnerability on physical devices. The text discusses the vulnerability itself, design process of the malicious code exploiting “Janus”, distribution of the malicious code and the means of attacking multiple devices at once. Result is a malicious application allowing remote eavesdropping of multiple Android devices at once.

Keywords Android, CVE-2017-13156, Janus, Android Runtime, Android Package, APK, APK Signature Scheme v1, Smali, electronic signature, unrestricted upload of file with dangerous type, remote eavesdropping, privacy violation, spyware

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Zranitelnost aplikačního virtuálního stroje *Android Runtime* označovaná jako CVE-2017-13156 umožňuje do zařízení s operačním systémem Android nainstalovat soubor, který může být interpretován jako DEX soubor, tedy soubor obsahující binární bytekód *Dalvik*. Zároveň ale může být interpretován i jako instalační balíček aplikace pro operační systém Android (*Android Package*, zkráceně APK). Aby se APK balíček mohl nainstalovat na zařízení, musí být digitálně podepsán – vývojářem či distribuční službou, jako je *Google Play*. Škodlivý soubor využívající zranitelnost Janus, tedy soubor vzniklý spojením APK balíčku a DEX souboru, si zachová podpis původního APK. Při instalaci škodlivého balíčku se podpis úspěšně zkontroluje, ale do zařízení se místo legitimního kódu aplikace nainstaluje kód z DEX souboru, který je s balíčkem APK spojen.

Zranitelnost Janus ohrožuje zařízení s operačním systémem Android verze 5.0 a vyšší, jejichž bezpečnostní záplata byla vydána nejpozději v prosinci 2017. V době psaní této práce je zranitelnost tedy již skoro tři roky opravená, ale přesto je stále aktuální. Dostupnost bezpečnostních záplat pro zařízení s operačním systémem Android totiž záleží na jejich výrobcích. Společnost *Google*, která operační systém Android udržuje, vyžaduje po výrobcích zařízení pouze to, aby bezpečnostní záplaty vycházely alespoň dva roky od vydání daného zařízení. Počet zranitelných zařízení nelze přímo určit, protože nejsou známy statistiky záplatovaných zařízení. Operační systém Android je s 74,13 % trhu nejpoužívanějším operačním systémem pro mobilní zařízení. V době psaní této práce obsahuje 63,5 % zařízení s OS Android některou z verzí systému, která zranitelnost může obsahovat.

Zranitelnost CVE-2017-13156 byla objevena v roce 2017 bezpečnostními experty ze společnosti *GuardSquare*. Ti svůj nález nahlásili společnosti *Google*, která zranitelnost v systému opravila a tuto opravu vydala v rámci bezpečnostní záplaty pro prosinec 2017. V době objevení zranitelnosti nebyly známé žádné útoky ani škodlivé soubory, které by Janus využívaly.

Cílem této práce je analyzovat principy zranitelnosti Janus a známé útoky, které tuto zranitelnost využívají, a na základě této analýzy navrhnout a implementovat útok, který zranitelnost Janus využije k tomu, aby do zařízení jedné nebo více obětí nainstaloval škodlivou aplikaci, která oběti vzdáleně odposlouchává.

K tomuto útoku je potřeba vytvořit tři různé aplikace – aplikaci pro OS Android obsahující logiku odposlechu, dále aplikaci, která vytvoří škodlivý instalační balíček podle principů využití zranitelnosti Janus, a nakonec aplikaci pro útočníkův server, která přijímá data odeslaná z napadených zařízení.

Aby oběť útoku na první pohled nepoznala, že si do svého zařízení nainstalovala škodlivou aplikaci, je potřeba škodlivý instalační balíček vytvořit na základě instalačního balíčku legitimní aplikace. Legitimní kód je přepsán škodlivým kódem aplikace s logikou odposlechu vytvořené v rámci tohoto útoku. Upravený instalační balíček legitimní aplikace je poté zpět sestaven a jeho DEX soubor je podle principů zranitelnosti Janus spojen s APK balíčkem neupravené legitimní aplikace. Při instalaci tohoto balíčku je kontrolován platný podpis legitimní aplikace, ale je nainstalován kód upravené aplikace s přidaným odposlechem.

Tato práce je rozdělena do čtyř kapitol. První kapitola (*Analýza*) tvoří teoretickou část práce – jsou v ní představeny pojmy nutné pro pochopení fungování zranitelnosti Janus, která je následně podrobně popsána. Je zde představen špionážní software, dále je zde popsána architektura OS Android, struktura instalačních balíčků, průběh instalace, systém oprávnění a způsob digitálního podepisování aplikací pro OS Android. Poté jsou zde vysvětleny principy využití zranitelnosti CVE-2017-13156 a způsob implementace záplaty, která ji opravuje. Dále jsou představeny známé útoky využívající Janus a poté je zde popsán dopad, který zranitelnost má na současná zařízení.

Ve druhé kapitole (*Návrh*) je popsána struktura útoku implementovaného v rámci této práce. Nejprve je vysvětleno, jaké vlastnosti musí mít legitimní aplikace, na jejímž základě je vytvořen škodlivý instalační balíček. Poté jsou rozebrány kroky potřebné pro to, aby do legitimní aplikace mohla být přidána funkcionalita pro odposlech. Následně je popsán návrh podoby aplikace s logikou pro odposlech, serverové aplikace pro příjem dat a aplikace, která generuje škodlivý instalační balíček podle principů zranitelnosti Janus.

Ve třetí kapitole (*Realizace*) jsou popsány implementační detaily tří modulů, které dohromady tvoří útok, tedy aplikace pro OS Android s logikou odposlechu, serverové aplikace pro příjem dat a aplikace pro tvorbu škodlivého instalačního balíčku využívajícího zranitelnost Janus. Následně je ukázán konkrétní způsob, jakým je přidána funkcionalita pro odposlech do legitimní aplikace.

Ve čtvrté kapitole (*Testování)* je popsán průběh testů všech implementovaných aplikací. Testy jsou na závěr vyhodnoceny a následně je diskutován možný rozvoj aplikací, který eliminuje nevýhody aktuální implementace nebo rozšiřuje působnost útoku.

Kapitola 1

Analýza

Zranitelnost Janus (CVE-2017-13156) umožňuje útočníkovi pozměnit podepsaný instalační balíček reálné Android aplikace, nahradit celý zkompilovaný kód aplikace svým vlastním kódem a nainstalovat ho do zařízení bez narušení podpisu balíčku. Cílem této práce je implementovat útok, který Janus využije a naruší soukromí obětí tak, že je bude vzdáleně odposlouchávat.

Tato kapitola začíná obecným popisem špionážního softwaru, který tuto činnost vykonává. Dále jsou v této kapitole podrobně popsány principy zranitelnosti Janus. Před rozborem zranitelnosti Janus je potřeba definovat pojmy, které jsou důležité pro její pochopení. Nejprve je představena architektura operačního systému Android, průběh instalace aplikací včetně struktury instalačních balíčků a proces spouštění aplikací v OS Android. Následně jsou vysvětleny základní myšlenky digitálních podpisů a poté je text zaměřen na způsoby, kterými lze podepsat instalační balíčky aplikací pro operační systém Android.

1.1 Špionážní software

Aplikace pro operační systém Android, jejíž vytvoření je součástí této práce, má za úkol narušit soukromí uživatele, který si ji nainstaluje do svého zařízení. Konkrétně má aplikace provádět vzdálený odposlech, tedy nahrávat zvuk na mikrofon zařízení a odesílat ho na útočníkův server. Podobným aplikacím, které narušují soukromí svých uživatelů, se říká *spyware*.

Spyware je druh škodlivého softwaru, který sbírá informace z výpočetního zařízení bez vědomí jeho uživatele. Sbírané informace mohou mít různé formy, mohou to být například snímky obrazovky, fotografie, videa, zvuk z mikrofonu, stisknuté klávesy, přihlašovací údaje nebo jiné osobní informace [1]. Některé sbírané informace na první pohled ani nemusí působit jako citlivé údaje. Například výzkumníci z univerzity ve Stanfordu dokázali vytvořit mobilní aplikaci, která při telefonním hovoru sledovala vibrace gyroskopu a odesílala informace o nich na server, který z nich pomocí strojového učení dokázal

zpětně zrekonstruovat mluvený projev majitele napadeného zařízení [2]. Mezi *spyware* ale nepatří software nainstalovaný po tom, co uživatel odsouhlasil licenční smlouvu EULA[1], ve které jsou popsané aktivity tohoto softwaru, které souvisí se sběrem uživatelových dat [1].

Existuje mnoho důvodů, proč je *spyware* využíván – typicky jde o osobní či profesní zvýhodnění nebo posílení autority. Špionážní software však v praxi bývá využit i k jiným účelům, například rodiče chtějí kontrolovat internetovou aktivitu svých dětí, jeden z páru žárlí na svého partnera, nebo obchodní společnost chce získat výrobní tajemství konkurence. *Spyware* může také být využit při vyšetřování policií nebo bezpečnostními složkami. Pod *spyware* patří také tzv. *adware*, což je škodlivý software, který tajně sleduje aktivitu uživatele (především při prohlížení internetu) a na jejím základě mu zobrazuje cílenou reklamu [3].

Jako *spyware* jsou někdy označovány i tzv. *dual-use apps*, tedy aplikace se dvojím využitím. Jde o aplikace, které jsou vytvořené pro legitimní účely, například dětský zámek zařízení nebo aplikace umožňující nalézt zařízení v případě jeho ztráty nebo krádeže. Takové aplikace je možné využít i pro účely, pro které nebyly vytvořeny – aplikace lze nainstalovat do cizího zařízení a využívat je ke sledování jeho majitele [4].

Existují různé typy *spywaru*, které mají odlišné aspekty – například způsob instalace, účel nebo perzistenci. Způsob instalace *spywaru* se nejčastěji klasifikuje podle toho, do jaké míry je do něj zapojen uživatel zařízení. Pro útočníka nejjednodušším způsobem, jak *spyware* nainstalovat do zařízení oběti, je přimět uživatele, aby ho nainstaloval sám. Špionážní software tak může být součástí aplikace, kterou uživatel chce nebo potřebuje ve svém zařízení mít. Dalším způsobem, jak může útočník dopravit *spyware* do zařízení oběti, je tzv. *drive-by download*. Škodlivý software se do zařízení stáhne a nainstaluje po tom, co oběť navštíví útočníkovu webovou stránku. Existují i způsoby, kdy *spyware* ke své instalaci uživatelovu aktivitu nepotřebuje vůbec – *spyware* může být automaticky nainstalován jiným škodlivým softwarem, který se v zařízení již nachází [5].

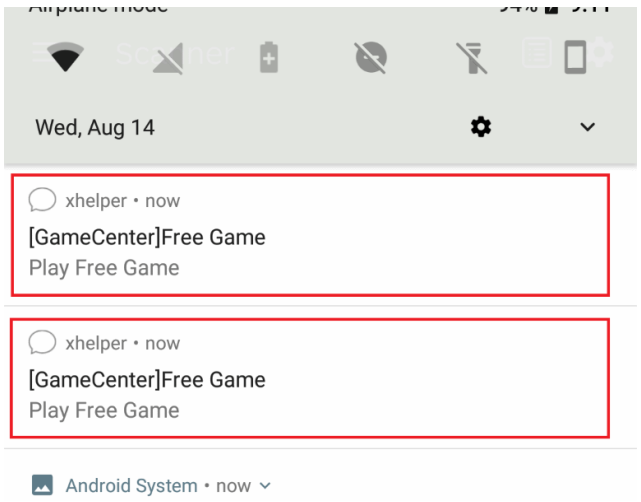
Pokud chce útočník vložit škodlivý kód do legitimní aplikace, zajímají ho tři hlavní vlastnosti této aplikace. První vlastností je rozšiřitelnost aplikace, tedy množství prostoru pro útočníkův kód, který legitimní aplikace poskytuje. Druhou vlastností je četnost spouštění legitimní aplikace uživatelem, případně její možnost nepřetržitého fungování. Některým druhům *spywaru* stačí být spuštěn jednou, jiné pro svou činnost zase potřebují více času, čehož docílí buďto nepřetržitým fungováním, nebo častým (ale kratším) spouštěním. Třetí pro útočníka zajímavou vlastností legitimní aplikace jsou její oprávnění. Útočník musí předpokládat, že možnosti jeho *spywaru* jsou v systému omezeny úplně stejně jako možnosti legitimní aplikace [6].

[1] *End-User License Agreement* je licenční smlouva, která určuje, jaké aktivity uživatel smí a nesmí se softwarem provádět [1].

Špionážní software, který pro svou činnost potřebuje delší dobu, musí řešit dva zásadní problémy – nesmí být v systému odhalen a v případě detekce nesmí být odstraněn. *Spyware* se může odhalení uživatelem vyhýbat například změnou svého jména tak, aby mezi ostatními procesy nebyl nápadný. Tato praktika však nestačí, pokud *spyware* nemá být odhalen antivirovým softwarem. Aby špionážní software obešel kontroly antivirů, může svůj škodlivý kód například zašifrovat [7].

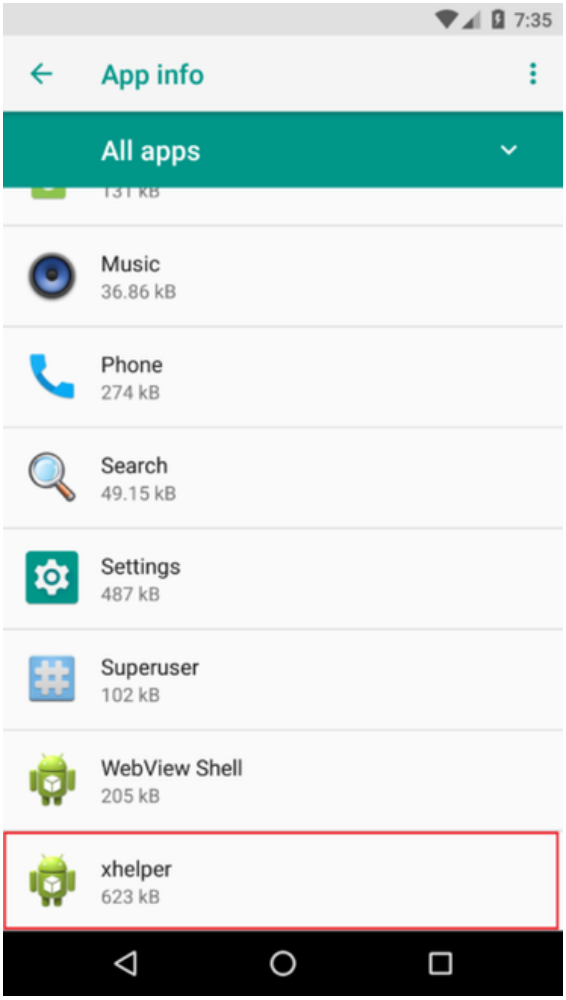
V případě, že je špionážní software v systému odhalen uživatelem, může svému odstranění zabránit například falešným dialogem pro odinstalování, který v uživateli vzbudí pocit, že se *spywaru* zbavil (podobným způsobům se říká *pasivní způsob obrany proti odstranění*). Útočník může do svého kódu zakomponovat i *aktivní obranu proti odstranění*. Pokud je *spyware* odhalen antivirovým softwarem, může antivirus odstranit jenom některé jeho části (ty, které považuje za škodlivé), například záznamy v registru nebo v seznamu aplikací, které se mají spustit po startu systému. Aktivní obranou proti takovému odstranění částí špionážního softwaru by potom bylo provádění periodické kontroly všech takových záznamů nebo komponent. Při této kontrole by *spyware* po zjištění změny (např. odstranění souboru nebo přepis hodnoty v registru) danou komponentu přepsal zpět do původního stavu [8].

Příkladem aktuálního špionážního softwaru pro operační systém Android, který se v systému velmi pečlivě skrývá, je *adware* zvaný *xHelper*. Tento škodlivý software byl objeven v květnu 2019 a vyskytuje se ve dvou verzích. První verze tohoto *adware* dokáže po své instalaci odstranit svou ikonu z nabídky aplikací i z plochy. Uživateli je pouze zobrazeno upozornění s textem „*xhelper*“ v horní liště. Do lišty se poté začnou přidávat další upozornění, která obsahují reklamy [9] – viz obrázek 1.1.



Obrázek 1.1: Reklamní sdělení v notifikační liště zobrazená škodlivým softwarem *xHelper* [10].

Druhá verze aplikace *xHelper* je v systému skrytá ještě důkladněji – také skrývá své ikony v nabídce aplikací a na ploše, ale navíc nevytvoří ani výše zmíněné upozornění s jejím jménem. Jediný důkaz, že je škodlivá aplikace v zařízení nainstalovaná, je položka „*xhelper*“ v seznamu nainstalovaných aplikací dostupném přes nastavení systému (viz obrázek 1.2).



Obrázek 1.2: *Adware xHelper* v seznamu nainstalovaných aplikací [10].

Nejzajímavější vlastností tohoto *adwaru* je, že se aplikace *xHelper* po několika hodinách od jejího odstranění do zařízení vrátí. Navíc, pokud uživatel uvede zařízení do továrního nastavení, *xHelper* v něm zůstane. Experti ze společnosti *Malwarebytes*, kteří *xHelper* analyzovali, již objevili způsob, jak tento *adware* ze zařízení trvale odstranit. Způsob, jakým se v zařízení skrývá a který mu umožňuje vydržet v zařízení po uvedení do továrního nastavení, však v době psaní této práce ještě objeven nebyl [9].

Škodlivá aplikace provádějící odposlech, jejíž implementace je cílem této práce, vznikne vložením škodlivého kódu s funkcionalitou odposlechu do legitimní aplikace, při jejíž volbě jsou brány v potaz výše uvedené vlastnosti. Škodlivá aplikace také potřebuje být spuštěna po co nejdelší dobu. Zároveň musí být implementována tak, aby mohla její činnost být skryta v procesu na pozadí. Návrh škodlivé aplikace vycházející z požadavků uvedených v této sekci je podrobně popsán v kapitole 2.

1.2 Architektura operačního systému Android

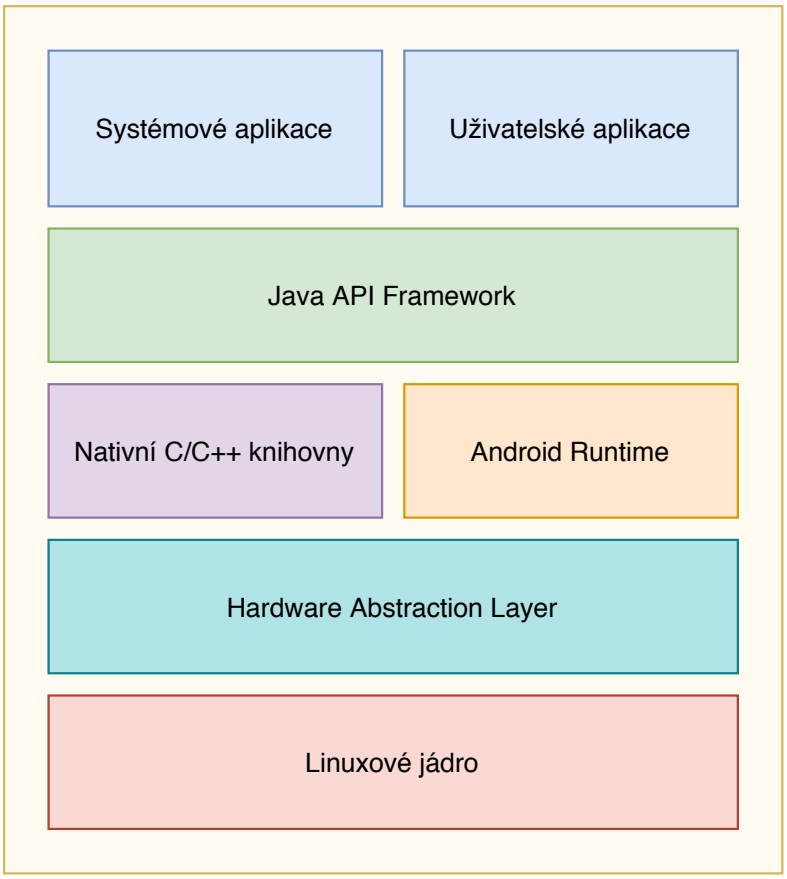
Android je populární operační systém pro zařízení s platformou ARM, kterou v dnešní době využívají převážně mobilní zařízení – např. chytré telefony, tablety, chytré hodinky, ale i zařízení pro chytrou domácnost nebo auta. Podle serveru *Statista.com* ho v prosinci 2019 využívalo téměř 2,5 miliardy zařízení, což činí 74,13 % světového trhu s operačními systémy pro mobilní zařízení [11].

Operační systém Android byl navržen Andym Rubinem, Chrisem Whitem, Nickem Searsem a Richem Minerem v roce 2003. Od roku 2005 vyvíjí operační systém Android společnost *Google*. Po přibližně pěti letech od vzniku operačního systému Android v roce 2003 byl v říjnu 2008 OS Android zpřístupněn veřejnosti [12].

Diagram znázorňující architekturu OS Android je na obrázku 1.3. Základem operačního systému Android je linuxové jádro. To má na starosti hlavní funkcionality operačního systému, jako např. plánování spouštění vláken, správu virtuální paměti nebo tvorbu síťových spojení. Mezi výhody použití právě linuxového jádra patří dostupnost zdrojového kódu, správa uživatelů a přístupových oprávnění nebo fakt, že jádro operačního systému Linux je obecně kvalitně zdokumentováno, a tedy usnadňuje práci výrobcům hardwaru při tvorbě ovladačů [13].

Nad jádrem se nachází vrstva zvaná *Hardware Abstraction Layer* (HAL). Ta poskytuje standardní rozhraní, která umožňují vyšším vrstvám (zejména Java API frameworku) pracovat s hardwarovým vybavením daného zařízení. HAL se skládá z knihovních modulů, z nichž každý implementuje rozhraní pro práci se specifickou hardwarovou komponentou, např. fotoaparátem nebo bluetooth modulem [13].

Každá aplikace je spuštěna ve svém vlastním aplikačním virtuálním stroji, který tvoří samostatný proces. Na zařízení s operačním systémem Android verze 5.0 a vyšší a jde o virtuální stroj *Android Runtime* (ART). Každá aplikace je na zařízení doručena v tzv. *Dalvik* bytekódu, což je označení pro přenosný kód, který je podobný *Java* bytekódu s tím rozdílem, že *Dalvik* bytekód je určen pro registrově orientovaný virtuální stroj a *Java* bytekód je určen pro zásobníkově orientovaný virtuální stroj. Na zařízení musí bytekód být zkompilován do spustitelného strojového kódu (již závislého na platformě). Bytekód *Dalvik* dostal jméno po předchůdci ART, aplikačním virtuálním stroji



Obrázek 1.3: Architektura operačního systému Android [13].

Dalvik, který fungoval jako *just-in-time* (JIT)[2] kompilátor. ART oproti tomu funguje jako *ahead-of-time* (AOT)[3] kompilátor [14].

Velké množství systémových komponent a služeb v OS Android (například HAL nebo ART) je postaveno na tzv. *nativním kódu*, což je kód již zkompilovaný pro běh na konkrétním procesoru, a proto pro svůj běh vyžadují nativní knihovny napsané v C a C++. Tyto knihovny spolu s ART zmíněným výše tvoří další vrstvu architektury operačního systému Android a vývojář aplikací pro OS Android je může díky Android NDK využít a implementací některých komponent svých aplikací v nativním C/C++ kódu zefektivnit jejich běh [13].

Sada všech funkcí operačního systému Android je vývojářům aplikací přístupná přes API[4] napsaná v jazyce Java. Tato rozhraní tvoří základní stavební bloky, které jsou potřeba pro tvorbu Android aplikací. Zjednodušují interakce

[2] Bytekód se překládá až v momentě, kdy je při běhu aplikace potřeba.

[3] Bytekód se zkompiluje jednou při instalaci a do zařízení se uloží pouze strojový kód.

[4] *Application Programming Interface*, sbírka funkcí, tříd či protokolů nějaké knihovny nebo programu, které může programátor využívat ve svém vlastním programu.

s jádrem systému a s ostatními vrstvami. V této sadě je např. *View System*, který lze využít pro tvorbu grafického rozhraní aplikace, *Notification Manager*, který umožňuje zobrazovat upozornění na stavové liště, nebo třeba *Content Provider*, který umožňuje aplikacím přistupovat k datům jiných aplikací [13].

V nejvyšší vrstvě architektury se nachází v systému nainstalované aplikace (v diagramu 1.3 uvedeny odděleně jako systémové a uživatelské). Operační systém Android mezi aplikacemi systémovými a aplikacemi, které si nainstaluje sám uživatel, z hlediska své systémové architektury nerozlišuje[5]. Díky tomu si uživatel může například změnit výchozí webový prohlížeč nebo aplikaci pro psaní SMS. Aplikace si navíc mohou vzájemně poskytovat funkcionalitu, kterou mohou vývojáři využít ve svých aplikacích. Pokud chce vývojář ze své aplikace například poslat SMS, nemusí její odeslání ve své aplikaci vytvářet sám, ale může využít libovolnou již nainstalovanou aplikaci a zprávu odeslat jejím prostřednictvím [13].

1.3 Principy instalace aplikací v OS Android

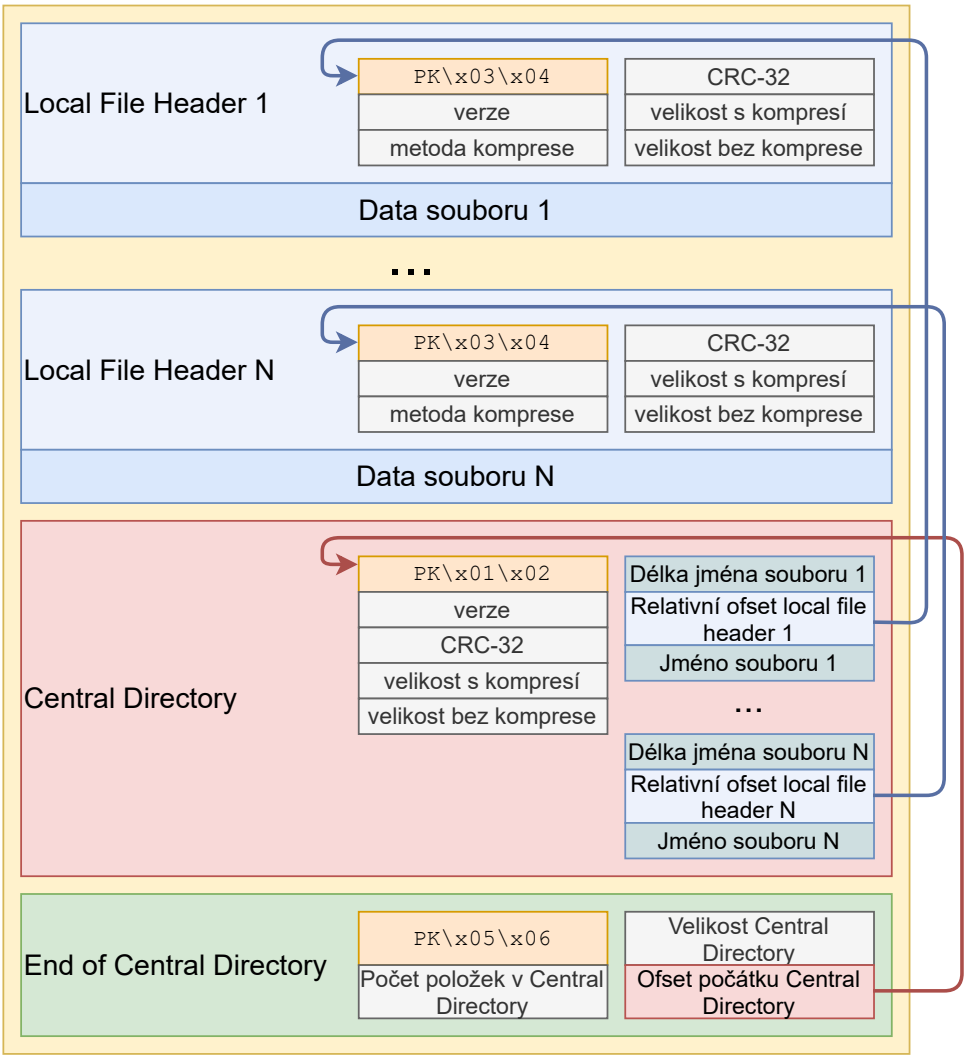
Aplikace pro Android jsou distribuovány ve formě balíčků zvaných APK (*Android Package*). Tyto balíčky obsahují vše, co aplikace potřebuje k úspěšnému spuštění – programový kód, metadata (jako např. digitální podpis aplikace) a ostatní soubory (např. obrazové soubory nebo zkompilované XML soubory, které definují grafické rozhraní) [15]. V této sekci je vysvětleno, jakou strukturu má instalační balíček aplikace, jak probíhá instalace a spuštění aplikace z hlediska operačního systému a jak funguje systém oprávnění, o která může aplikace v operačním systému Android žádat.

1.3.1 Soubor APK

Instalační balíček aplikace pro operační systém Android se nazývá APK. Soubor APK byl až do představení operačního systému Android verze 7.0 v roce 2016 pouze ZIP archiv s pravidly pro strukturu jeho obsahu [16]. ZIP archiv je soubor, který může obsahovat jeden nebo více souborů nebo adresářů, a umožňuje je bezeztrátově zkomprimovat. Struktura formátu ZIP je znázorněna na obrázku 1.4 [17].

Archiv ve formátu ZIP je zpracováván od konce. Nejprve se nalezne a přečte sekce *End of Central Directory*, která začíná sekvencí bytů `PK 05 06`. Pomocí ní se nalezne počátek sekce *Central Directory*, která začíná sekvencí `PK 01 02` a obsahuje seznam jmen a ofsetů všech souborů obsažených v archivu. Každý z obsažených souborů má vlastní sekci *Local File Header*, která začíná sekvencí `PK 03 04` a obsahuje metadata souboru (např. metodu kom-

[5] Jeden rozdíl mezi systémovými a uživatelskými aplikacemi však je – systémovým aplikacím (předinstalované v zařízení, podepsané stejným klíčem jako samotný systém) mohou být přidělena vyšší systémová oprávnění.



Obrázek 1.4: Schéma formátu ZIP souboru [17].

prese, velikost s kompresí a bez komprese). Za sekcí *Local File Header* se vždy nachází samotná data souboru [17].

Tvorba souboru APK při vývoji aplikace je proces skládající se z několika kroků. Nejprve jsou převedeny všechny příslušné XML soubory (např. definice grafického rozhraní aplikace) do binární podoby. Poté jsou všechny AIDL soubory (*Android Interface Definition Language* – programátorská definice rozhraní, pomocí kterých aplikace komunikuje se servery nebo ostatními aplikacemi) přeloženy do jazyka Java. Veškerý zdrojový kód aplikace je následně spolu se soubory získanými z předchozích kroků zkompilován do souborů `.class`, které jsou poté společně s knihovnami třetích stran převedeny do jednoho DEX (*Dalvik executable*) souboru – `classes.dex`. Všechny zkom-

pilované i nezkompilované zdroje (např. obrázky) a soubor `classes.dex` jsou následně zabaleny do APK archivu a podepsány alespoň jedním ze tří existujících podpisových schémat (viz sekce 1.4 na straně 14) [15].

Výsledný balíček APK obsahuje následující položky [15]:

- `/assets`: Adresář, kam vývojář může umístit soubory, které má aplikace mít k dispozici.
- `/res`: Adresář obsahující rozvržení grafického rozhraní, obrázky a další soubory, ke kterým aplikace přistupuje z kódu.
- `/lib`: Adresář s nativními C/C++ knihovnami, které aplikace využívá. Knihovny jsou zkompilované a roztříděné do podadresářů podle cílové architektury (např. x86, ARM, MIPS).
- `/META-INF`: Adresář s certifikátem aplikace a se soubory, které slouží k ověření původu a zachování integrity tohoto archivu.
- `AndroidManifest.xml`: Soubor, který obsahuje konfiguraci aplikace a bezpečnostní parametry. Například využívané *service* (viz sekce 2.1.1 na straně 37) nebo požadovaná oprávnění.
- `classes.dex`: Soubor obsahující spustitelný *Dalvik* bytekód aplikace.
- `resources.asrc`: Soubory, které jsou uloženy v adresáři `/res` mohou být místo toho zkompilovány do tohoto souboru.

1.3.2 Průběh instalace

V předchozí sekci byla představena tvorba instalačního balíčku aplikace pro operační systém Android. V této sekci je rozebráno, jaké kroky je potřeba provést po doručení instalačního balíčku do zařízení, aby bylo možné aplikaci korektně nainstalovat a následně ji spustit. Komponenty operačního systému Android *Package Manager Service* a *installd*, které instalaci aplikace zprostředkovávají, musí učinit následující kroky [18]:

- Určit správné místo v systému souborů, kam aplikaci nainstalovat. Aplikace, které mají být systémové (mohou být spuštěny s vyššími oprávněními, zpravidla bývají předinstalované v zařízení) se ukládají do adresáře `/system/priv-app/`. Aplikace uživatelské se ukládají do adresáře `/data/app/`.
- Určit, zda se jedná o instalaci nebo aktualizaci.
- Ověřit podpis instalačního balíčku (viz sekce 1.4).
- Uložit soubor APK do určeného adresáře.

- Určit *user identifier* (UID) aplikace. Každá aplikace (proces) má vlastní UID, na které se mohou vázat oprávnění, se kterými je aplikace spuštěna.

- Vytvořit adresář pro data aplikace v souborovém systému operačního systému Android a nastavit mu příslušná oprávnění.

- Extrahovat nativní knihovny a uložit je do adresáře jménem `/libs`, což je podadresář adresáře pro data aplikace, který byl vytvořen v předchozím kroku.

- Extrahovat DEX soubor s kódem aplikace (`classes.dex`), optimalizovat jeho bytekód (výstupem je tzv. ODEX soubor) a poté ho uložit do globálního *Dalvik cache* adresáře (`/data/dalvik-cache`).

- Přečíst `AndroidManifest.xml` a na jeho základě přidat záznam o balíčku do dvou souborů. Prvním je soubor `packages.list`. Ten obsahuje pro každý nainstalovaný balíček informace o jeho názvu, UID a adresáři, kde je aplikace nainstalovaná. Druhý soubor, `packages.xml`, obsahuje informace o každém nainstalovaném balíčku a oprávněních, se kterými se má aplikace spustit.

- Vyslat informaci (tzv. *broadcast*) oznamující, že byl nainstalován nový balíček. *Broadcast* je zpráva, která je poslána všem aplikacím a komponentám systému, které si zaregistrují tzv. *broadcast receiver*, neboť chtějí být informovány o instalaci nového balíčku. Na tuto zprávu můžou tyto aplikace pomocí *receiveru* nějakým způsobem reagovat, např. antivirová aplikace může novou aplikaci zkontrolovat.

1.3.3 Spuštění aplikace

Při startu operačního systému je spuštěn aplikační virtuální stroj (zvaný *Zygote*), který poslouchá na UNIXovém soketu a přijímá požadavky na spuštění nových aplikací. Když uživatel spustí aplikaci, komponenta *Activity Manager Service* pošle *Zygote* zprávu s parametry potřebnými pro start aplikace. V momentě, kdy *Zygote* obdrží takovýto požadavek, jednoduše zduplikuje svůj vlastní proces příkazem `fork`, ovšem s parametry a kódem spouštěné aplikace. Před spuštěním aplikace je zkontrolován obsah *Dalvik cache*, zda se v ní již nenachází zoptimalizovaná verze DEX souboru s kódem aplikace. Pokud nenachází, systém musí kód nejprve zoptimalizovat (což může mít vliv na dobu spouštění aplikace).

Tento způsob spouštění je velice efektivní, protože systémové knihovny jsou sdílené mezi jednotlivými aplikačními virtuálními stroji a nemusí se s každou novou aplikací znovu nahrávat do paměti [19].

1.3.4 Oprávnění aplikací v operačním systému Android

Chytrá zařízení s operačním systémem Android poskytují svému uživateli různorodé funkce – lze pomocí nich například uskutečňovat telefonní hovory, sledovat GPS polohu, prohlížet internetové stránky nebo pořizovat fotografie. Aplikace typicky nepotřebuje využívat všechny zmíněné funkce najednou. Z bezpečnostního hlediska je lepší, aby každá aplikace měla přístup pouze k těm funkcím, které skutečně potřebuje. Proto má Android svůj vlastní systém oprávnění. Při využívání zranitelnosti Janus představuje oprávnění pro útočníka nezanedbatelnou překážku, což je dále podrobněji rozvedeno v sekci 1.5.1 na straně 28.

Aplikace, která vyžaduje přístup k určité funkci systému, musí žádost o oprávnění mít uvedenou ve svém `AndroidManifest.xml` v elementu `<uses-permission>`. Každé oprávnění má svůj vlastní název, který je do manifestu nutné napsat. Například, pokud má aplikace využívat síťové prvky zařízení, musí vývojář do manifestu pod `<uses-permission>` uvést oprávnění `android.permission.INTERNET` [20].

Oprávnění jsou v systému vynucována různými způsoby. Jak již bylo zmíněno v sekci 1.3.2, každá aplikace na Androidu je spuštěna pod svým vlastním linuxovým UID. Některá oprávnění jsou realizována tak, že se UID aplikace přidá do linuxové skupiny spojené s daným oprávněním. Například žádost o oprávnění pro přístup na síť (`INTERNET`) přidá UID aplikace do skupiny `inet`, což jí umožní otvírat `AF_INET` a `AF_INET6` sokety na úrovni jádra systému. Přiřazení jednotlivých oprávnění k linuxovým skupinám je definované v souboru `/system/etc/permissions/platform.xml` [21].

Existují ale i oprávnění, která jsou na vyšší úrovni abstrakce a žádné skupiny přidělené nemají. Například oprávnění pro přístup k SMS zprávám (`android.permission.READ_SMS`) neumožní aplikaci přímo přistupovat k SMS databázi, ale povolí jí posílat dotazy na tzv. *content provider*[6], zde konkrétně `content://sms` [20].

Ne všechna oprávnění mají stejnou váhu – například přidělení oprávnění nastavit budík je mnohem méně rizikové než přidělení oprávnění k instalaci aplikací. Oprávnění v operačním systému Android se proto dělí do několika skupin (*protection levels*) podle jejich rizikovosti [22] (viz tabulka 1.1).

Oprávnění, která jsou klasifikována jako *normal*, jsou aplikaci přidělena automaticky při instalaci[7]. Oprávnění, která jsou klasifikována jako *dangerous*, vyžadují explicitní povolení od uživatele. Až do verze OS Android 6.0 (rok 2015) se uživateli před instalací aplikace vypsal seznam všech nebezpečných oprávnění vyžadovaných aplikací a uživatel je musel povolit všechna najednou, jinak se instalace neprovedla. Od verze OS Android 6.0 již uživateli při instalaci aplikace žádná oprávnění vypsána nejsou. Aplikace si za běhu sama musí ohlídat (pomocí API volání), zda vyžadované oprávnění již získala. Pokud

[6] *Content provider* je strukturované rozhraní pro přístup ke sdíleným datům [21].

[7] Ale pouze ta, která má aplikace uvedena ve svém `AndroidManifest.xml` [22].

nezískala, musí o něj uživatele (opět pomocí API volání) požádat. Uživateli se potom zobrazí okno s otázkou, zda aplikaci přiděluje jedno konkrétní oprávnění. Pokud uživatel odmítne, aplikace může fungovat dále, ovšem nesmí používat komponentu, k níž se žádané oprávnění vztahovalo [22].

Protection level	Popis
normal	Výchozí hodnota pro neriziková oprávnění.
dangerous	Oprávnění může vést k přístupu k potenciálně citlivým informacím nebo provádění změn na zařízení.
signature	Může být přiděleno pouze aplikaci podepsané stejným certifikátem, jako aplikace, která dané povolení definovala (tzv. *custom permission*).
signatureOrSystem	Stejné jako *signature*, ale může být přiděleno i systémové aplikaci.
system	Oprávnění lze přidělit pouze systémové aplikaci.

Tabulka 1.1: Úrovně (*protection levels*) oprávnění v operačním systému Android a jejich význam [20].

1.4 Digitální podpis

Digitální podpis je pro zranitelnost Janus důležitý. Podpis instalačních balíčků APK je poměrně složitý proces, který má za úkol zachovat integritu aplikace a ověřit její původ. Pomocí zranitelnosti Janus však útočník dokáže integritu aplikace narušit a její původ zfalšovat.

Digitální podpis je kryptografická operace, která se používá pro ověření původu zprávy (případně souboru) a zachování její integrity. Přestože existuje mnoho různých algoritmů pro samotné podepisování, základní myšlenka je u všech stejná – zpravidla se nepodepisuje celá zpráva nebo soubor, ale pouze její tzv. *hash* [23].

Hash je termín z anglického jazyka, který se používá pro výstup hashovací funkce, což je jednosměrná funkce[8] s pevnou délkou výstupu, která pro jeden konkrétní vstup vrátí vždy stejný výstup a pro dva různé vstupy ideálně vrací dva různé výstupy[9]. Malá změna ve vstupu by měla vést na tak velkou změnu ve výstupu, aby nový výstup vypadal jako nekorelující se starým (tzv. *avalanche effect*). Kryptograficky bezpečné hashovací funkce, které se při podepisování používají, by dále měly splňovat bezkoliznost 1. a 2. řádu – nelze

[8] Výstup lze snadno vypočíst, ale zpětně získat z výstupu vstup je výpočetně velmi složité.

[9] Pouze v ideálním případě. Ve skutečnosti, kvůli již zmíněné pevné délce výstupu, je obor hodnot hashovací funkce omezen a tzv. kolize mohou nastat. Čím je výstup delší, tím je pravděpodobnost kolize nižší.

efektivně nalézt dva různé vstupy se stejným výstupem hashovací funkce a pro daný vstup nelze efektivně nalézt jiný vstup, který by vedl na stejný výstup. Výhodou podepisování výstupů hashovacích funkcí místo celých souborů je, že je podepisováno pouze malé množství dat charakteristických pro danou zprávu nebo soubor. Při sebemenším pozměnění zprávy jsou podepisována úplně jiná data a útočník nemá možnost zprávu změnit tak, aby hash (a tím i podpis) zachoval [24].

Digitální podpis je asymetrická kryptografická operace. To znamená, že se během ní používají dva klíče – veřejný (ten může znát kdokoliv) a privátní (ten smí znát pouze podepisující a nelze ho nijak odvodit z klíče veřejného). Podpis zprávy potom vznikne tak, že podepisující zašifruje[10] hash zprávy svým privátním klíčem. Kdokoliv, kdo má veřejný klíč podepisujícího, s ním může podpis dešifrovat, čímž by měl (pokud je podpis pravý) získat hash, kterou podepisující původně zašifroval. Příjemce zprávy si dále spočítá hash zprávy a pokud se s ní dešifrovaný podpis shoduje, je podpis legitimní a původ zprávy a její integrita jsou tak ověřeny [23].

Aby si příjemce zprávy mohl být jistý, že veřejný klíč, který získal od odesílatele, je skutečně odesílatelův a ne podvržený útočníkem, může odesílatel místo pouhého veřejného klíče zveřejnit svůj certifikát. Certifikát je struktura, která mimo jiné obsahuje jméno majitele, jeho veřejný klíč a dobu platnosti certifikátu. Samotný certifikát je podepsaný tzv. *certifikační autoritou*, což je důvěryhodná organizace, která ověřuje identitu žadatelů o certifikát. Tím, že je podepsaný, je samotný certifikát chráněn před pozměněním. Příjemce zprávy si do svého zařízení může nainstalovat (nebo má již předinstalované) certifikáty důvěryhodných certifikačních autorit. Pomocí veřejných klíčů v nich může ověřit pravost certifikátu odesílatele [24].

1.4.1 Podpis instalačního balíčku APK

K podpisu instalačního balíčku aplikace pro operační systém Android jsou využívány digitální certifikáty ve formátu X.509, k jejichž veřejným klíčům má privátní klíč pouze vývojář dané aplikace. Podpis balíčku APK je povinný – během vývoje aplikace jsou i její neoficiální verze vývojovými prostředími automaticky podepisovány tzv. *debug* klíči. Certifikát sloužící k ověření podpisu (`CERT.RSA`) je v APK uložen v adresáři `/META-INF` [25].

Účel podpisu APK souboru je jednak jeho ochrana před nežádoucím pozměněním, ale také prokázání původu aplikace, aby systém mohl určit, do jaké míry je instalační balíček důvěryhodný. Aktualizace aplikace se nainstaluje pouze v případě, že je podepsána stejným klíčem jako již nainstalovaná aplikace. Některé aplikace spolu mohou komunikovat nebo spolupracovat a podpis využívat k tomu, aby se ujistily, že jsou pro sebe vzájemně důvěryhodné [26].

[10] Jak konkrétně zde již záleží na zvoleném algoritmu podpisu.

Operační systém Android neimplementuje konvenční *Public Key Infrastructure* proces – nijak nekontroluje, zda jsou všechna pole v certifikátu platná (např. jméno), že autor je skutečně ten, za koho se vydává, nebo zda je certifikát podepsán nějakou certifikační autoritou. Ve skutečnosti má většina aplikací certifikát podepsaný sám sebou (tzv. *self-signed*). Jediný okamžik, kdy je podpis kontrolován, je při instalaci aplikace. Po instalaci další kontroly již neprobíhají a pokud certifikátu vyprší platnost, nic se nestane a aplikace bude stále normálně fungovat. Společnost *Google* doporučuje nastavovat platnost certifikátů alespoň na 25 let, aby nevznikaly problémy s platností při aktualizaci aplikace [27].

V dnešní době existují již tři tzv. podpisová schémata určující, co a jak se má podepsat a kam se má podpis uložit. Soubor APK je možné podepsat pomocí všech tří schémat zároveň (což je i doporučeno z důvodu zpětné kompatibility) [26]. Pro využití zranitelnosti Janus je klíčové znát princip podpisového schématu verze 1. V této sekci je též vysvětleno, jak fungují schémata verze 2 a 3.

1.4.2 *APK Signature Scheme v1*

Podpisové schéma verze 1 bylo představeno již v operačním systému Android 1.0 v roce 2008 a je totožné s tzv. *JAR signing* – soubory ve formátu JAR (*Java ARchive*), což jsou balíčky pro distribuci aplikací napsaných v jazyce Java, jsou stejně jako APK založeny na archivu ZIP. Co se obsahu JAR archivů týče, mají proti APK volnější formát. Schématické znázornění *APK Signature Scheme v1* je na obrázku 1.5 na straně 18.

APK podepsané schématem verze 1 musí obsahovat soubor `MANIFEST.MF` v adresáři `/META-INF`, ve kterém musí být zaznamenány všechny soubory, které APK obsahuje a také jejich kontrolní hash. Tím se jednotlivé soubory ochrání proti nechtěnému pozměňování. Příklad, jak může vypadat položka v manifestu APK souboru, je ve výpisu 1.1 [26].

```
Name: test/classes/ClassOne.class
SHA1-Digest: TD1GZt8G11dXY2p4olSZPc5Rj64=
```

Výpis 1.1: Příklad jedné položky v souboru `MANIFEST.MF`. Tato položka obsahuje jméno souboru `test/classes/ClassOne.class` a jeho SHA-1 hash [28].

Samotný soubor `MANIFEST.MF` je také chráněn proti pozměnění. V APK archivu se totiž musí nacházet soubor `CERT.SF` (přípona `.SF` je označení pro *signature file*), který obsahuje jednak hash celého souboru `MANIFEST.MF`, ale i hash každé jeho položky zvlášť (zde již nejde o hash souborů v archivu, ale skutečně o hash dvouřádkové položky, která je znázorněna ve výpisu 1.1). Při kontrole podpisu je nejdřív ověřena hash celého souboru `MANIFEST.MF` a

pokud tato kontrola selže, ověřují se hashe jednotlivých položek. Ukázka, jak může obsah souboru *signature file* vypadat, je ve výpisu 1.2 [28].

```
Signature-Version: 1.0
SHA1-Digest-Manifest: h1yS+K9T7DyHtZrtI+LxvgqaMYM=
Created-By: 1.7.0_06 (Oracle Corporation)

Name: test/classes/ClassOne.class
SHA1-Digest: fcav7ShIG6i86xPepmitOVo4vWY=

Name: test/classes/ClassTwo.class
SHA1-Digest: xrQem9snnPhLySDiZyclMlsFdtM=
```

Výpis 1.2: Příklad obsahu souboru `CERT.SF`. Na začátku je uvedena verze podpisového schématu, poté následuje SHA-1 hash celého `MANIFEST.MF` a pod ním jsou uvedena jména jednotlivých souborů v APK archivu a SHA-1 hashe jejich položek ze souboru `MANIFEST.MF` [28].

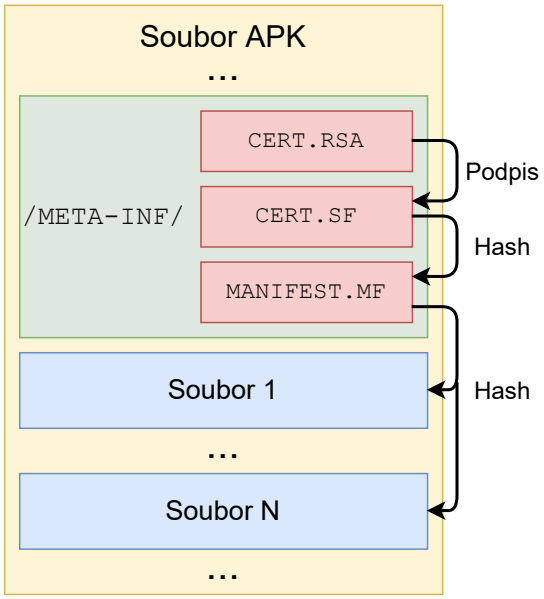
Dalším povinným souborem v *APK Signature Scheme v1* je `CERT.RSA` (případně, protože Android pro podpisy podporuje kromě algoritmu RSA i DSA a ECDSA, může přípona být i `.DSA` nebo `.EC`). Tento soubor obsahuje jednak kryptografický podpis souboru `CERT.SF`, ale také certifikát sloužící k ověření tohoto podpisu [26].

Při ověření podpisu APK je nejprve ověřen podpis souboru `CERT.SF` uložený v `CERT.RSA`. Dále je zkontrolována hash celého souboru `MANIFEST.MF` uložená v `CERT.SF`. Pokud tato kontrola selže, je zkontrolována hash každé položky `MANIFEST.MF` zvlášť (též uloženy v `CERT.SF`). Následně jsou ověřeny všechny hashe souborů z archivu, které jsou zapsány v `MANIFEST.MF`. Ověření podpisu APK může ještě selhat, pokud archiv obsahuje soubory, které nejsou v `MANIFEST.MF` vůbec uvedeny [26].

V souvislosti s tímto podpisovým schématem byla v Androidu objevena celá řada zranitelností (včetně zranitelnosti Janus, té je v této práci věnována samostatná sekce). Nejznámější z nich je pravděpodobně *Master Key Vulnerability* [26], která je zranitelnosti Janus velmi podobná. Jak zde již bylo zmíněno, kontrola podpisu APK souboru probíhá při jeho instalaci, kdy jsou jednotlivé soubory extrahovány, jsou kontrolovány hashe a kryptografický podpis a pokud dojde k neshodě, je instalace APK odmítnuta. Díky *Master Key Vulnerability* může útočník do APK archivu přidat vlastní soubor, který má stejný název jako jiný soubor, který se v archivu již vyskytoval. V ověřovacím procesu je kontrolována integrita správného souboru, ale instalace dále pracuje se souborem útočníka. Ten tak může v APK archivu například nahradit DEX soubor s kódem aplikace [26].

Další zranitelností pak je tzv. *second Masterkey*. Ta spočívá v tom, že ofsety zapsané v položkách souboru ZIP (obrázek 1.4 na straně 10) jsou podle

dokumentace bezznaménkové, ale při instalaci s nimi bylo zacházeno jako se znaménkovými. Ve výsledku je tato zranitelnost pro útočníka ještě výhodnější, protože se nemusí omezovat na soubory, které v APK již jsou, ale může do něj přidat své vlastní, které díky uměle vytvořeným ofsetům budou použity při instalaci, ale ne při ověření [26].



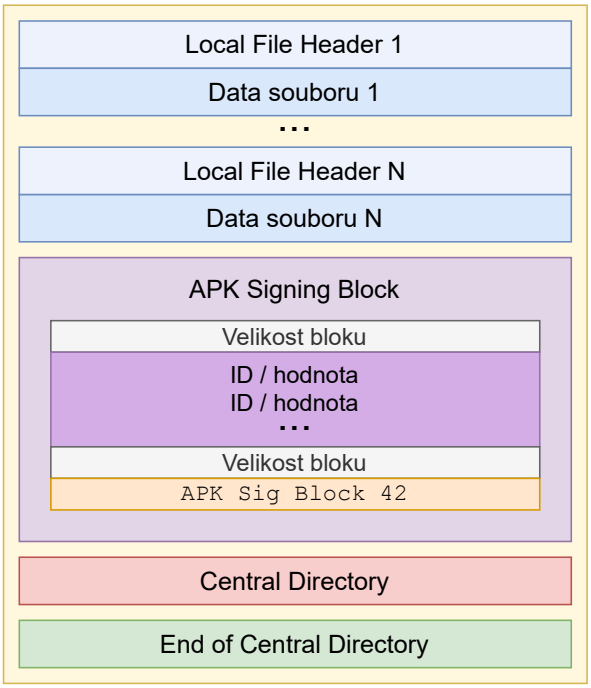
Obrázek 1.5: Znázornění *APK Signature Scheme v1*. Šipky vedou od souborů obsahujících hash nebo podpis k souborům, o jejichž hash nebo podpis se jedná.

1.4.3 *APK Signature Scheme v2*

S operačním systémem Android verze 7.0 v roce 2016 přišla nová verze podpisového schématu, která přinesla silnější ověření APK souboru. Namísto toho, aby se kontrolovala integrita každého obsaženého souboru zvlášť, je kontrolována integrita archivu jako celku [27].

S příchodem *APK Signature Scheme v2* již APK soubory nejsou pouze ZIP archivy. To je z toho důvodu, že druhá verze podpisového schématu do struktury ZIP souboru těsně před sekci *Central Directory* přidává sekci zvanou *APK Signing Block* (viz obrázek 1.6) [16].

APK Signing Block na začátku obsahuje velikost bloku, poté následuje sekvence dvojic „ID – hodnota“, poté opět velikost bloku (totožný záznam jako na začátku bloku) a celý *APK Signing Block* je zakončen šestnáctibytovou konstantní značkou `APK Sig Block 42`. Při zpracovávání APK souboru je nejprve běžným způsobem, stejně jako při zpracovávání ZIP archivu, nalezena sekce *Central Directory*, od které se postupuje směrem k začátku souboru,



Obrázek 1.6: Struktura APK souboru po podpisu pomocí *APK Signature Scheme v2* [16].

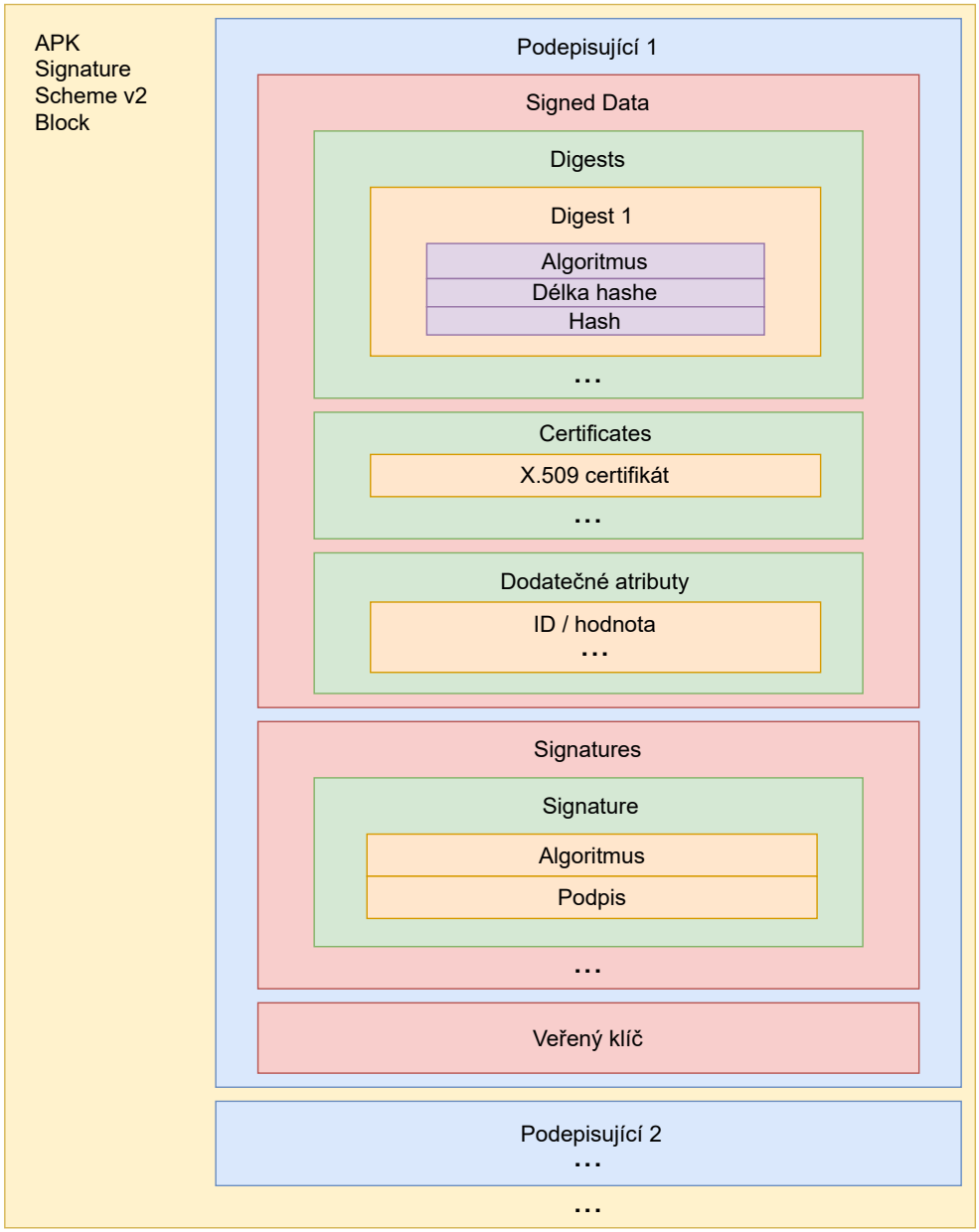
a přečte se konstantní značka *APK Signing* bloku, za kterou následuje velikost bloku. Když je velikost známá, je možné přečíst celý blok [29].

Mezi dvojicemi „ID – hodnota“ je pod ID `0x7109871a` uložen *APK Signature Scheme v2 Block*. V něm může být pro každého podepisujícího uloženo více[11] záznamů pro kontrolu integrity (včetně algoritmů, kterými byly získány), certifikátů ve formátu X.509, které představují identitu podepisujícího (první z nich se bere jako hlavní), veřejný klíč (patřící k hlavnímu certifikátu) a další přídavné atributy. Struktura *APK Signature Scheme v2 Blocku* je znázorněna na obrázku 1.7 [16] [29] [30].

Podpisové schéma verze 2 chrání integritu samotných souborů v archivu, sekce *Central Directory*, sekce *End of Central Directory* a bloků *Signed data* v rámci *APK Signing Blocku*. Integrita sekcí *Central Directory*, *End of Central Directory* a souborů v ZIPu je chráněna jedním nebo více výstupy hashovací funkce spočítanými z jejich obsahu a uloženými v blocích *Signed data*, jejichž integrita je chráněna jedním nebo více podpisy [29].

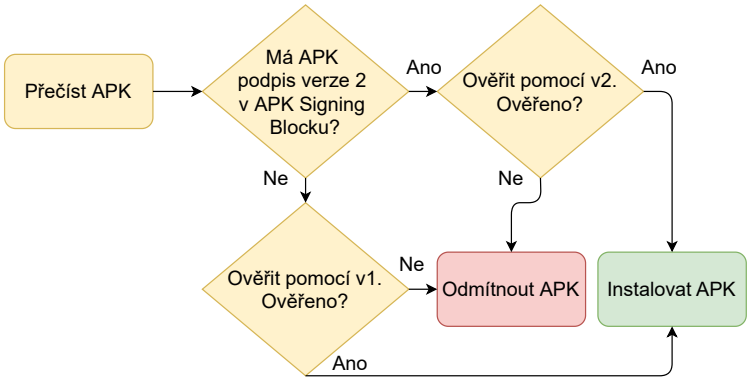
Pokud by útočník měl APK soubor podepsaný schématem verze 2 a pokusil se ho ověřit tak, jako by byl podepsaný verzí 1, ověření by selhalo. APK archivy podepsané schématem druhé verze mají v souboru `CERT.SF` atribut

[11] U každé kategorie ale musí být alespoň jeden zástupce.



Obrázek 1.7: Struktura *APK Signature Scheme v2 Blocku* [16] [29] [30].

`X-Android-APK-Signed`. Jako hodnotu tohoto atributu mají APK soubory seznam verzí schémat, kterými jsou podepsány [29]. Diagram průběhu ověření APK souboru s více podpisovými schématy je na obrázku 1.8.



Obrázek 1.8: Postup ověření podpisu *APK Signature Scheme v2* [29].

Při ověření podpisu verze 2 je nejprve nutné v APK najít *APK Signing Block* a zkontrolovat, že obě položky s jeho velikostí mají stejnou hodnotu. Dále také, že sekce *End of Central Directory* následuje hned za sekcí *Central Directory* a také, že za sekcí *End of Central Directory* nenásledují žádná další data. Poté je nutné nalézt *APK Signature Scheme v2 Block*. Pokud neexistuje, ověřování proběhne podle schématu verze 1 [29]. Následující kroky se provedou pro každého podepisujícího v *APK Signature Scheme v2 Blocku* [29]:

- Vybrat nejsilnější podporovaný algoritmus ze sekce *Signatures* (pořadí podle síly je záležitostí jednotlivých implementací).
- Ověřit odpovídající podpis ze sekce *Signatures* proti položkám sekce *Signed data* za použití *Veřejného klíče*.
- Ověřit, že seřazené seznamy podpisových algoritmů v sekcích *Digests* a *Signatures* jsou totožné. Tento krok předchází útokům typu *signature stripping*[12].
- Spočíst hash obsahu APK souboru stejným algoritmem, který používá zvolený podpisový algoritmus.
- Ověřit, že spočtená hash je identická, jako odpovídající hash ze sekce *Digests*.
- Ověřit, že atribut `SubjectPublicKeyInfo` prvního z certifikátů je stejný jako *Veřejný klíč*.

[12]Útok, kdy útočník systém donutí použít tak slabý podpisový algoritmus, že již nemusí být bezpečný a útočník ho dokáže zfalšovat, případně nějak jinak prolomit.

Ověření je úspěšné, pokud byl v *APK Signature Scheme v2 Blocku* nalezen alespoň jeden podepisující a pokud pro každého nalezeného podepisujícího úspěšně skončila posloupnost kroků popsaná výše [29].

1.4.4 *APK Signature Scheme v3*

V operačním systému Android 9 (2018) byla představena nová verze podpisového schématu. Spíše než o nové schéma jde o rozšíření *APK Signature Scheme v2*, proto se mu také někdy říká *APK Signature Scheme v2+*.

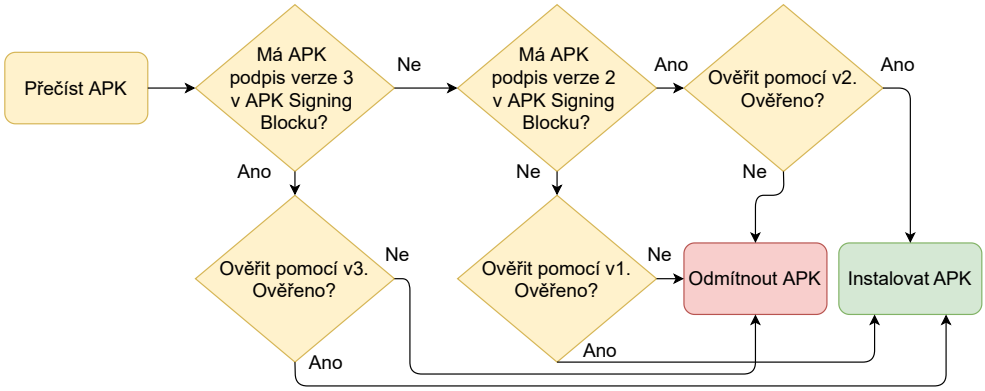
Hlavní novinka, kterou podpisové schéma verze 3 přináší, je tzv. *certificate rotation*. Jde o možnost vyměnit certifikát za jiný, aniž by aplikace kvůli novému certifikátu ztratila důvěryhodnost. Typicky se tato operace provádí, pokud certifikátu vyprší platnost, mění se certifikační autorita nebo je některý z certifikátů kompromitován (je vyzrazen jeho privátní klíč) [31].

Stejně jako *APK Signature Scheme v2* používá verze 3 *APK Signing Block* jako součást struktury samotného ZIP souboru. V něm je pod ID `0xf05368c0` uložen *APK Signature Scheme v3 Block*. Ten je velmi podobný *APK Signature Scheme v2 Blocku*. Je v něm však přidaná informace o podporovaných verzích Android SDK (de facto verze operačního systému) a dále mezi dodatečnými atributy nově obsahuje strukturu, která se nazývá *proof-of-rotation* (ID `0x3ba06f8c`) [32].

Struktura *proof-of-rotation* umožňuje aplikacím výše zmíněnou rotaci certifikátů, tedy jejich výměnu. Ta proběhne, aniž by aplikace byla ostatními aplikacemi, se kterými komunikuje, zablokována z důvodu nedůvěryhodnosti certifikátu. Toho je docíleno novými položkami u podpisu aplikace. Jednou z nich je ujištění pro třetí strany, že certifikát aplikace je důvěryhodný, pokud jsou důvěryhodní i jeho předchůdci. Další z nich jsou starší certifikáty aplikace, kterým ostatní aplikace stále důvěřují [32].

Struktura *proof-of-rotation* obsahuje jednosměrný spojový seznam, jehož každý uzel obsahuje certifikát, kterým byla podepsána jedna ze starších verzí aplikace. Seznam je řazený podle verze a v kořeni obsahuje nejstarší certifikát. Každým z certifikátů v seznamu je podepsán ten, který po něm ihned následuje, což slouží jako důkaz důvěryhodnosti nového certifikátu [32].

Ověření podpisu se skládá ze stejných kroků, jako ověření v případě schématu verze 2. Mezi kroky, které se provádí pro každého podepisujícího, je přidána kontrola minimální a maximální verze SDK (jedna z přidaných položek do *v3*). Na konci těchto kroků, pokud u daného podepisujícího existuje struktura *proof-of-rotation*, je ještě nutné ověřit, že *proof-of-rotation* je platná (podpisy v řetězu sedí) a že hlavní certifikát (používaný po celou dobu ověřování) je poslední certifikát v seznamu. Pokud v *APK Signing Blocku* neexistuje položka *APK Signature Scheme v3 Block*, kontroluje se podpis pomocí metody schématu verze 2 (viz obrázek 1.9) [32].



Obrázek 1.9: Postup ověření podpisu *APK Signature Scheme v3* [32].

1.5 Zranitelnost Janus

Janus je zranitelnost v operačním systému Android, která je oficiálně označovaná jako CVE-2017-13156 či také Android ID A-64211847. Zranitelnost byla objevena v červenci roku 2017 bezpečnostními výzkumníky společnosti *GuardSquare* a byla pojmenována podle starořímského boha duality[13] (obrázek 1.10) [34]. Janus je zranitelnost v aplikačním virtuálním stroji *Android Runtime* (viz sekce 1.2), která může vést k nežádoucí instalaci nebezpečného souboru [35].

Zranitelnost Janus se týká formátu instalačních balíčků pro operační systém Android a způsobu jejich podepisování – konkrétně *APK Signature Scheme v1*. Jak již bylo zmíněno v sekci 1.3.1, soubor APK byl až do představení *APK Signature Scheme v2*, kdy byl do jeho struktury přidán *APK Signing Block*, obyčejným ZIP souborem. Struktura ZIP souboru byla představena v sekci 1.3.1 a znázorněna na obrázku 1.4 na straně 10.

Ve skutečnosti ale vnitřní struktura ZIP archivu není pevně daná – při rozhodování, zda se jedná o platný ZIP soubor, se kontroluje pouze přítomnost povinných sekcí (*Central Directory*, *End of Central Directory*, *Local File Headery*) podle jejich hlaviček a ofsetů. Nijak se ale nekontrolují data, která se nachází mezi těmito sekcemi [34].

Ve specifikaci ZIP formátu se nenachází požadavek, aby výše zmíněné sekce byly bezprostředně za sebou. ZIP soubor tak může vypadat jako např. na obrázku 1.11 na straně 25. Mezi jednotlivými sekcemi může být zapsáno libovolné množství libovolných dat, která fungují pouze jako výplň. Pokud se

[13] Janus je jeden z nejstarších starořímských bohů. V minulosti byl patronem vrat, dveří, vchodů, začátků a konců, dodnes je po něm v některých jazycích pojmenován měsíc leden (lat. *Ianuarius*). Janus je obvykle zobrazován s dvěma opačnými obličeji (obrázek 1.10 na straně 24) a byl bohem přeměny minulosti na budoucnost, přechodu z jednoho stavu do druhého, nebo obecně duality [33]. Díky této symbolické podvojnosti byla po bohu Janovi pojmenována i zranitelnost, která je hlavním předmětem této práce.

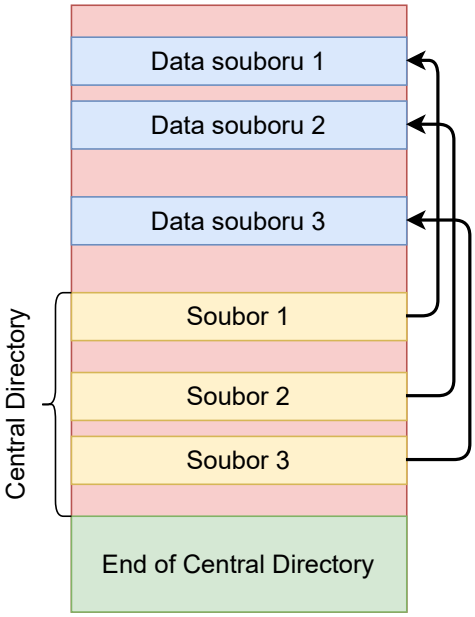
Obrázek 1.10: Starořímský bůh Janus – typické vyobrazení s dvěma různými opačně hledícími tvářemi [36].

v sekci *End of Central Directory* nachází platný ofset sekce *Central Directory* a v sekci *Central Directory* jsou platné všechny ofsety sekcí *Local File Header* jednotlivých souborů, je platný celý archiv [34].

V sekci 1.4.2 o *APK Signature Scheme v1* již bylo zmíněno, že hlavní nevýhodou podpisového schématu verze 1 je absence kontroly integrity archivu jako celku (tato kontrola byla přidána až ve schématu druhé verze). *APK Signature Scheme v1* provádí kontrolu integrity na úrovni souborů, které archiv obsahuje. To jsou jednak soubory, které do archivu vložil programátor nebo vývojové prostředí, ale i soubory, které obsahují kontrolní informace k podpisu. Podpisové schéma verze 1 ale nekontroluje integritu na úrovni sekcí – *Central Directory*, *End of Central Directory* a *Local File Header*. Stejně tak nekontroluje integritu výplňových dat mezi jednotlivými sekcemi.

Bezpečnostní výzkumníci ze společnosti *GuardSquare* zjistili [34], že mezi jednotlivé sekce APK souboru mohou vložit libovolný *Dalvik Executable* (DEX) soubor. DEX soubor, jak již bylo zmíněno v sekci 1.3.1 na straně 9, obsahuje veškerý programový kód aplikace pro operační systém Android. Jeho formát je vidět na obrázku 1.12 na straně 26. Na začátku DEX souboru se nachází hlavička. Hlavička začíná, stejně jako sekce v ZIP archivu, poznávací značkou, která se skládá ze slova `dex` a verze, např. `038`. V hlavičce je dále uložen kontrolní součet a za ním následují ofsety a délky jednotlivých sekcí DEX souboru. Bytekód aplikace je v *Dalvik executable* rozdělen do sekcí podle typu obsahu. Mezi tyto sekce patří např. sekce s řetězci, použitými typy, definovanými metodami nebo třídami [37].

Podobně jako v ZIP archivu může v DEX souboru být výplň. V DEX je pro ni prostor úplně na konci souboru, za poslední sekcí s daty. I do výplně DEX souboru lze vložit libovolné množství dat, aniž by byla narušena jeho integrita. Výzkumníci přišli na to, že mohou spojit *Dalvik executable* s APK



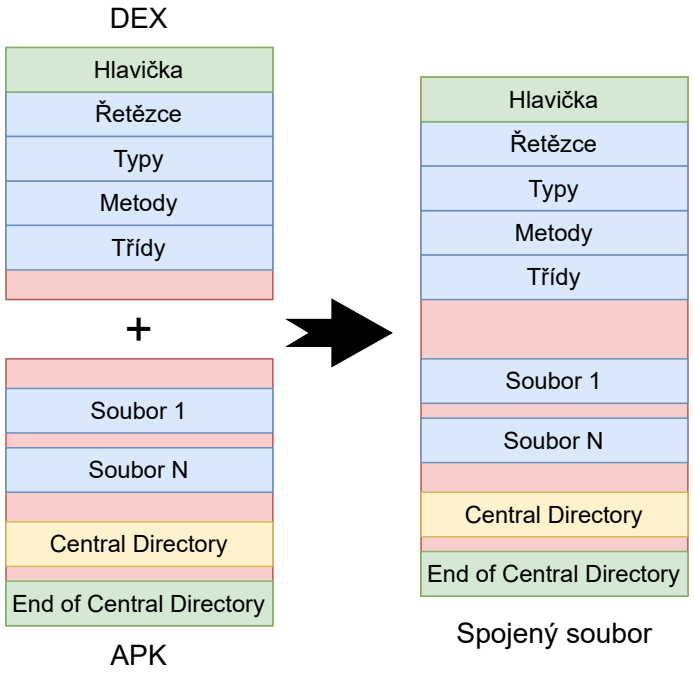
Obrázek 1.11: Možná struktura ZIP archivu s třemi soubory. Červenou barvou je znázorněna výplň, která se v souboru může nacházet. ZIP se zpracovává odzadu, takže výplň může být i na samém začátku souboru.

archivem tak, že výsledný spojený soubor začíná hlavičkou DEX souboru, za níž pokračují jeho data, a končí celým APK archivem (úplně poslední položkou je ZIP sekce *End of Central Directory* z APK). Na takto spojený soubor bude z hlediska systému Android nahlíženo jako na DEX i APK zároveň (odtud pochází pojmenování zranitelnosti Janus) [34]. Toto spojení je znázorněno na obrázku 1.12.

Zásah do APK souboru navíc nenaruší podpis aplikace podpisovým schématem verze 1, protože soubory patřící APK archivu, u kterých je podpisem jednotlivě hlídána integrita, zůstanou zachovány. V samotném APK archivu se upraví pouze výplň a ofsety zapsané v jeho ZIP sekcích [27].

Při instalaci aplikace v OS Android, jak je řečeno v sekci 1.3.2, je archiv APK rozbalen a extrahován na různá místa souborového systému. Při instalaci APK, které má před sebe napojený DEX soubor, se na DEX hledí pouze jako na bezvýznamnou výplň. Instalační balíček APK se kontroluje a zpracovává odspodu, takže se systémové komponenty rozbalující archiv k datům DEX souboru ani nedostanou [34].

Po extrakci ostatních souborů z APK archivu komponentami *Package Manager Service* a *installd* by aplikační virtuální stroj *Android Runtime* (ART) měl z APK extrahovat soubor `classes.dex` (DEX legitimní aplikace) a pracovat s ním. ART ale při instalaci dokáže pracovat nejen s APK balíčky,



Obrázek 1.12: Spojení instalačního balíčku *Android Package* (APK) a souboru *Dalvik Executable* (DEX) [34]. Červeně jsou znázorněna data, která slouží pouze jako výplň.

ze kterých DEX sám extrahuje, ale i se samotnými DEX soubory. Když ART obdrží soubor k dalšímu zpracování, zkontroluje nejprve, jestli tento soubor na začátku obsahuje dříve zmíněnou poznávací sekvenci DEX formátu. Na základě toho rozhodne, zda obdržel DEX, nebo APK [34]. Rozhodování *Android Runtime*, zda soubor zpracuje jako ZIP archiv, nebo jako DEX soubor, je ve výpisu 1.3.

Ve výpisu 1.3 jsou vypsány klíčové části členské funkce `Open` třídy `DexFile`, součásti aplikačního virtuálního stroje *Android Runtime*, zde ve verzi z roku 2017 [38], kdy byla objevena zranitelnost Janus. Třída `DexFile` je definována v souboru `art/runtime/dex_file.cc` [38]. V aktuální verzi (v době psaní této práce) je kód přesunut do souboru `art/libdexfile/dex/dex_file_loader.cc` [39]. Soubor, který je funkci předán jako první argument, je otevřen a je z něj přečtena poznávací značka `magic`. Na řádku 224 je rozhodnuto, zda jde o poznávací značku ZIP souboru (přesněji, protože je značka přečtena ze začátku souboru, o poznávací značku první sekce *Local File Header*). Pokud načtená značka `magic` patří ZIP archivu, je soubor jako ZIP dále zpracován. Pokud ne, je na řádku 227 zkontrolováno, zda jde o platnou poznávací značku DEX formátu. Pokud ano, je soubor zpracován přímo jako DEX. V opačném případě je vrácena chyba.

```
211 bool DexFile::Open(const char* filename, /*...*/){
      // ...
218   uint32_t magic;
219   File fd = OpenAndReadMagic(filename, &magic,
          error_msg);
      // ...
224   if (IsZipMagic(magic)) {
225     return DexFile::OpenZip(fd.Release(), location,
            /*...*/);
226   }
227   if (IsValidMagic(magic)) {
228     std::unique_ptr<const DexFile> dex_file(
            DexFile::OpenFile(/*...*/));
      // ...
240   }
241   *error_msg = StringPrintf("Expected valid zip or
        dex file: '%s'", filename);
242   return false;
243 }
```

Výpis 1.3: Výtah z členské funkce `Open` patřící třídě `DexFile`, která je součástí *Android Runtime*. V této funkci je rozhodnuto (řádky 224-240), zda bude se souborem zacházeno jako s DEX, nebo jako s APK [38]. Zelené komentáře značí místa, kde byl vynechán kód, který pro ukázku nebyl podstatný.

Při instalaci souboru spojeného způsobem jako na obrázku 1.12 bude na soubor nahlíženo nejprve jako na APK s výplní na začátku, jehož obsah se rozbalí korektním způsobem, ale poté bude na spojený soubor z hlediska ART nahlíženo jako na DEX s výplní na konci, který bude místo legitimního `classes.dex` zoptimalizován, uložen do `dalvik-cache` a spouštěn [34].

Existují dva hlavní způsoby, jak lze zranitelnost Janus využít. První způsob je skrytí *malwaru*. *Malware* se ukryje v DEX souboru, který je spojen s APK obsahujícím minimum funkcionality. Tento spojený soubor na nezáplatovaném systému projde bezpečnostními kontrolami a *malware* se nainstaluje [40].

Druhý způsob využití zranitelnosti Janus je úprava již existující aplikace bez vědomí jejího vývojáře či distributora. Útočník tak může do zařízení doručit vlastní kód. Spojený soubor se může vydávat za aktualizaci již nainstalované legitimní aplikace, protože při spojení DEX souboru a APK archivu není narušen podpis APK a spojený soubor je tak pro systém důvěryhodný. Tento způsob využití zranitelnosti je teoreticky možné použít i k napadení privilegovaných systémových aplikací, které k instalaci vyžadují podpis stejným certifikátem, jako samotný systém [40].

1.5.1 Omezení pro útočníka

Pokud chce útočník využít zranitelnost Janus tím způsobem, že před legitimní APK archiv připojí DEX soubor s vlastním kódem, musí legitimní aplikaci volit pečlivě. Protože pomocí zranitelnosti Janus dokáže útočník pozměnit pouze APK archiv podepsaný *APK Signature Scheme v1*, primárním kritériem výběru reálné aplikace je pro něj to, aby zvolený APK soubor nebyl podepsán navíc i jiným schématem [34].

Při instalaci souboru vzniklého spojením DEX souboru a APK archivu jsou nejprve rozbaleny a použity soubory z APK, včetně metadat a informací například ze souboru `AndroidManifest.xml`. Jak bylo řečeno v sekcích 1.3.1 a 1.3.4, soubor `AndroidManifest.xml` obsahuje jednak bezpečnostní nastavení aplikace (například vyžadovaná oprávnění), ale i jiné konfigurační informace, například využívané služby (*services*, podrobněji vysvětleno v sekci 2.1.1 na straně 37) nebo třídy registrované jako *broadcast receiver* (podrobněji popsáno v sekci 2.1.2 na straně 39).

Pokud by útočník soubor `AndroidManifest.xml` pozměnil, narušil by tím podpis balíčku, jelikož podpisové schéma verze 1 hlídá integritu každého obsaženého souboru, a systém by odmítl aplikaci nainstalovat [27].

Útočník se tedy musí řídit tím, co je v souboru `AndroidManifest.xml` již uvedeno. Kód z útočníkova DEX souboru bude spuštěn pouze s takovými oprávněními, o která žádá legitimní aplikace, před jejíž APK útočník svůj DEX soubor připojí. Pokud legitimní aplikace nemá v manifestu uvedeny třídy zaregistrované jako *broadcast receiver*, nelze navíc z útočníkova kódu od systému přijímat *broadcast* informace např. o změně stavu síťových připojení nebo provedených změnách jinde v systému. Útočníkův kód nemůže ani spouštět nebo využívat služby (procesy na pozadí), které ve svém manifestu nemá legitimní aplikace vypsané [40].

Pokud ale legitimní aplikace služby, *broadcast receivery* nebo *content providery* v `AndroidManifest.xml` uvedeny má, musí je při instalaci systém nalézt i v útočníkově kódu. Pokud je útočník nepotřebuje, musí je i tak ve svém DEX souboru implementovat. Stačí je ale mít implementované jako prázdné třídy [40].

1.5.2 Záplata a obrana před útoky

Zranitelnost Janus se týká pouze *APK Signature Scheme v1*. Jak bylo zmíněno v sekci 1.4.3, s verzí 7.0 operačního systému Android bylo v roce 2016 představeno *APK Signature Scheme v2*, které soubory vzniklé spojením DEX a APK neumožňuje vytvořit. Podpisové schéma verze 2 totiž na rozdíl od podpisového schématu verze 1 u APK archivu kontroluje integritu souboru jako celku od začátku až po *APK Signing Block* a také integritu sekcí *Central Directory* a *End of Central Directory*.

Ověření podpisu *APK Signature Scheme v2* tedy soubor vytvořený spojením DEX a APK odhalí hned ze dvou důvodů. Prvním důvodem je fakt, že kontrola integrity probíhá i u míst, která specifikace ZIP archivu a podpisové schéma verze 1 považují pouze za výplň (viz sekce 1.4.3). Pokud se změní jediný bit této výplně, ověření podpisu selže.

Druhým důvodem, proč podpisové schéma druhé verze nepovolí instalaci spojeného souboru, je fakt, že pro spojení DEX souboru a APK balíčku je třeba v APK v sekci *End of Central Directory* změnit ofset počátku sekce *Central Directory* a v sekci *Central Directory* změnit ofsety sekcí *Local File Header* všech obsažených souborů a adresářů (viz obrázek 1.4 na straně 10). Tyto ofsety jsou totiž počítány od začátku souboru, takže se po připojení APK archivu za DEX soubor zvýší o délku DEX. Jak ale bylo řečeno výše, *APK Signature Scheme v2* hlídá integritu sekcí *Central Directory* a *End of Central Directory*, takže jakýkoliv zásah do nich není možný.

Zařízení, která mají verzi operačního systému starší než 7.0, ale *APK Signature Scheme v2* nepodporují. Na druhou stranu zařízení s verzí systému Android 7.0 a novější z důvodu zpětné kompatibility podporují i *APK Signature Scheme v1* (o podpisu pomocí schématu verze 2 rozhoduje sám vývojář aplikace). Společnost Google tedy vydala 1. prosince 2017 záplatu, která využití zranitelnosti Janus v podpisovém schématu verze 1 znemožňuje [41].

Zranitelnost Janus byla opravena následujícím způsobem. Komentář u *commitu* v oficiálním repozitáři se zdrojovým kódem operačního systému Android, ve kterém byla oprava zveřejněna, říká:

> `zip_archive: reject files that don't start with an LFH signature` [42].

V souboru `zip_archive.cc` byl pozměněn způsob zpracovávání ZIP archivů. Soubory, které nezačínají poznávací značkou `PK 03 04`, tedy poznávací značkou sekce *Local File Header* (viz obrázek 1.4 na straně 10), jsou při instalaci balíčku odmítnuty [42]. Podmínka pro úspěšnou instalaci souboru vzniklého spojením útočníkova DEX a legitimního APK je, aby tento soubor začínal poznávací značkou DEX (viz sekce 1.5). Spojený soubor využívající zranitelnost Janus tedy bude při instalaci odmítnut. Kód, kterým byla zranitelnost Janus opravena, je v ukázce 1.4.

Ukázka 1.4 pochází ze souboru `zip_archive.cc` [43]. Funkci `ParseZipArchive` je jako argument předán soubor, který je otevřen a načten do paměti. Na řádku 446 je ze začátku (ofset 0 jako poslední argument funkce `mapped_zip.ReadAtOffset`) tohoto souboru přečteno 32 bitů (`sizeof(uint32_t)`), které jsou následně uloženy do proměnné `lfh_start_bytes`. Na řádku 450 je proměnná `lfh_start_bytes` porovnána s proměnnou `LocalFileHeader::kSignature`, což je konstanta, která má hodnotu `0x04034b50` [44], neboli `PK 03 04`, což je poznávací značka sekce *Local File Header* v ZIP souborech (viz obrázek 1.4 na straně 10).

```
363 static int32_t ParseZipArchive(ZipArchive* arch){
      // ...
445   uint32_t lfh_start_bytes;
446   if (!arch->mapped_zip.ReadAtOffset(
            reinterpret_cast<uint8_t*>(&lfh_start_bytes),
            sizeof(uint32_t),
            0)) {
447     ALOGW("Zip: Unable to read header for entry
            at offset == 0.");
448     return -1;
449   }
450   if (lfh_start_bytes !=
              LocalFileHeader::kSignature) {
451     ALOGW("Zip: Entry at offset zero has invalid LFH
            signature %" PRIx32, lfh_start_bytes);
452   // ...
453     return -1;
454   }
455   ALOGV("+++ zip good scan %" PRIu16 " entries",
          num_entries);
456   return 0;
457 }
```

Výpis 1.4: Kód, kterým byla opravena zranitelnost Janus. Část funkce `ParseZipArchive`, která má na starosti zpracování ZIP archivů [43]. Zelené komentáře značí místa, kde byl vynechán kód nepodstatný pro ukázku.

Pokud obsah proměnné `lfh_start_bytes` poznávací značce neodpovídá, funkce končí s chybou. V opačném případě funkce vrací úspěch [43].

Hlavní obranou proti škodlivému softwaru, který Janus využívá, je výše zmíněná záplata systému. Pokud tuto záplatu na zařízení nelze nainstalovat, je nutné spolehnout se na jinou obranu. Obecně je doporučováno v nastavení zařízení nepovolovat instalaci aplikací z neznámých zdrojů (aplikace se tak instalují pouze od důvěryhodných distributorů). Dále je možné (viz sekce 4.4) na zařízení spustit službu *Google Play Protect*, která je v OS Android předinstalovaná a škodlivé soubory spolehlivě detekuje, nebo nainstalovat některou z antivirových aplikací třetích stran.

1.5.3 Dopad na stávající zařízení

Protože Janus je zranitelnost aplikačního virtuálního stroje *Android Runtime*, který byl v operačním systému Android představen ve verzi 5.0 [14], je tato verze i nejnižší verzí systému, na které lze Janus využít [34].

V OS Android verze 7.0 bylo představeno podpisové schéma verze 2, což systém zčásti chrání. Na Android 7.0 a vyšší však lze z důvodu zpětné kompatibility instalovat i aplikace podepsané *APK Signature Scheme v1* [26], takže i verze systému 7.0 a 7.1 je nutné označit jako potenciálně zranitelné.

Verze operačního systému Android 8.0 byla vydána 21. srpna 2017 [45], což je méně než 4 měsíce před vydáním záplaty opravující zranitelnost Janus (prosinec 2017 [41]). Verze 8.0 operačního systému Android se též uvádí jako zranitelná, i když drtivá většina zařízení s touto verzí systému je v dnešní době záplatovaná. Společnost *Google* totiž po výrobcích Android zařízení vyžaduje, aby na každé zařízení vycházely bezpečnostní záplaty minimálně po dobu dvou let od vydání tohoto zařízení a aby pro každé zařízení byly v rámci prvního roku od jeho vydání zpřístupněny alespoň čtyři bezpečnostní záplaty systému[14] [46]. Přehled všech verzí, kterých se zranitelnost týká, a jejich procentuální rozšíření je v tabulce 1.2.

Číslo verze	Název	Rozšíření
5.0 – 5.1	Lollipop	14,5 %
6.0	Marshmallow	16,9 %
7.0 – 7.1	Nougat	19,2 %
8.0	Oreo	12,9 %
		Celkem: 63,5 %

Tabulka 1.2: Přehled verzí operačního systému Android, kterých se týká zranitelnost CVE-2017-13156 [47]. Možnost využití zranitelnosti závisí na aplikovaných záplatách. Procentuální rozšíření je počítáno ze všech zařízení s operačním systémem Android. Maximální procentuální zastoupení zranitelných Android zařízení je v době psaní této práce 63,5 %.

Nelze přesně určit, kolik v dnešní době existuje zařízení s operačním systémem Android, které CVE-2017-13156 obsahují. Tabulka 1.2 sice obsahuje procentuální rozšíření jednotlivých zranitelných verzí OS Android [47], ale nejsou v ní brány v potaz bezpečnostní záplaty. K počtu záplatovaných zařízení nejsou k dispozici žádná data – každý výrobce může svá zařízení podporovat po různě dlouhou dobu [46].

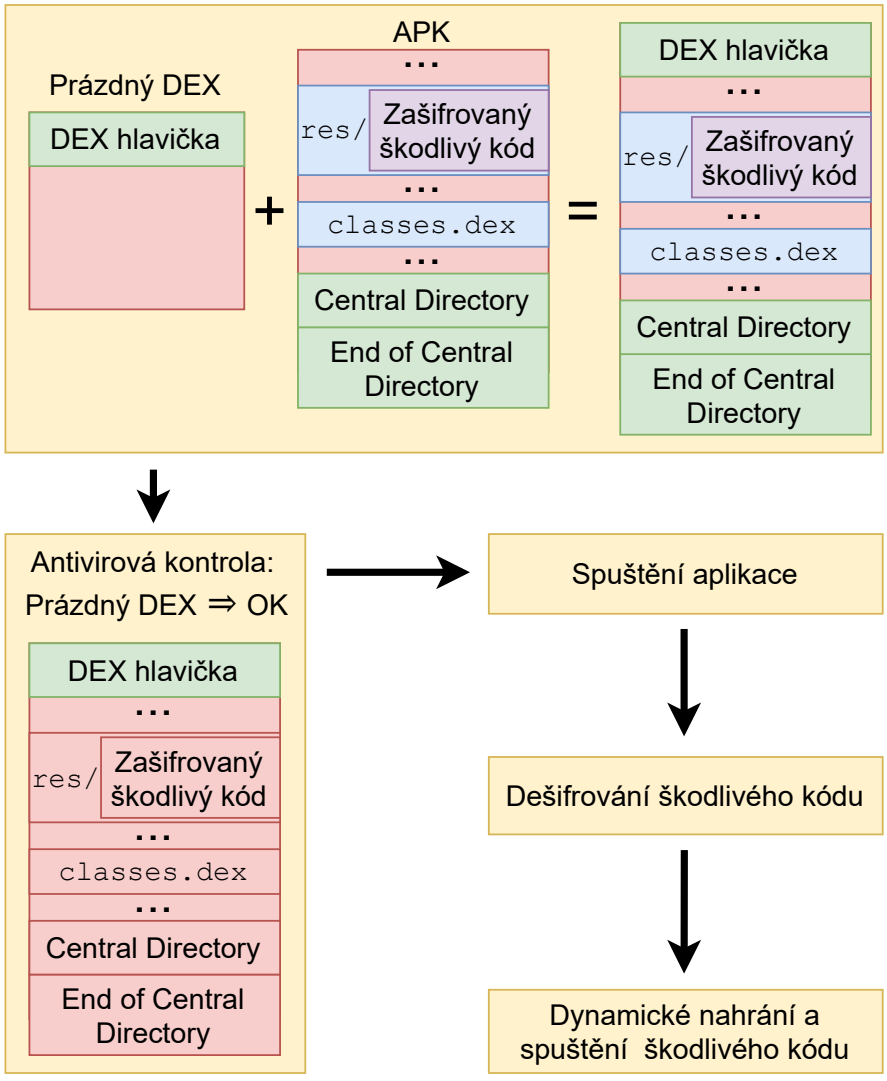
1.5.4 Známé útoky

V oficiální zprávě o nalezení zranitelnosti Janus společností *GuardSquare*, která vyšla v listopadu 2017, je uvedeno, že výzkumníci z *GuardSquare* žádný reálný útok, který by Janus využíval, v té době neobjevili [34].

O měsíc později, v prosinci 2017, společnost *TrendMicro* ve svém blogu uvedla, že objevila jeden druh *malwaru*, který Janus využíval. *Malware* byl

[14] Pro druhý rok od vydání zařízení již není požadavek na konkrétní počet záplat [46].

k dispozici v internetovém obchodě s aplikacemi *Google Play* a vydával se nejprve za aplikaci, která čistí systém, a poté byl aktualizován a prezentoval se jako zpravodajská aplikace [40]. Postup akcí prováděných tímto *malwarem* je znázorněn na obrázku 1.13.

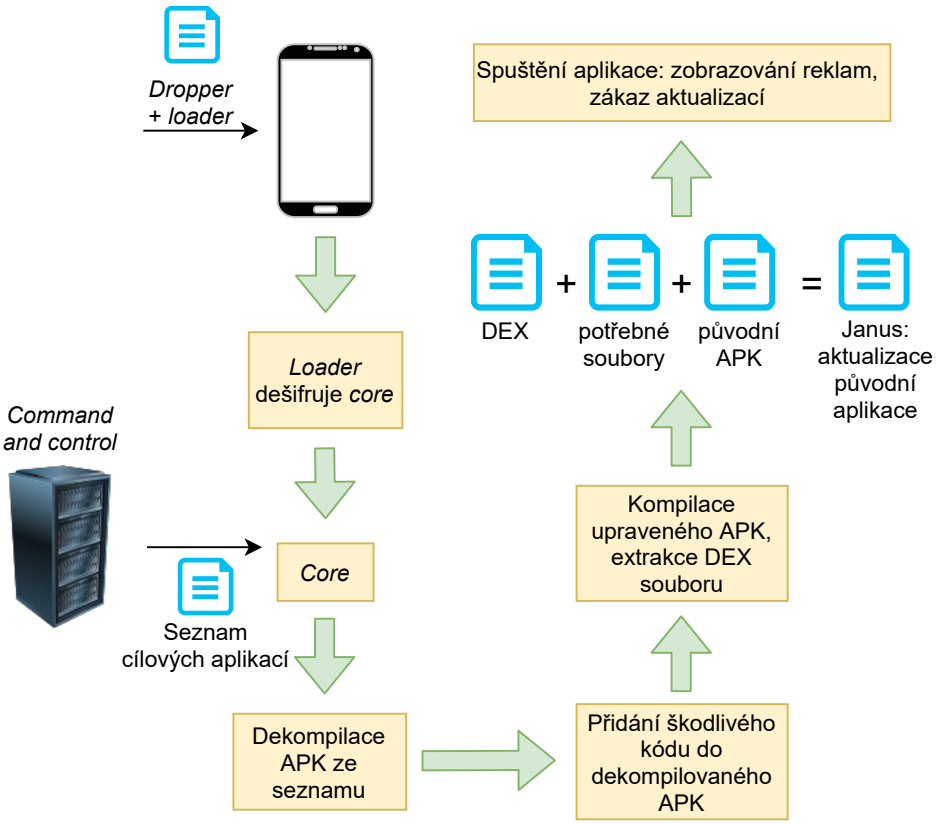


Obrázek 1.13: Postup škodlivého softwaru objeveného společností *TrendMicro*, který využívá zranitelnost Janus. Červenou barvou jsou znázorněny části souborů, které jsou v dané chvíli vnímány jako výplň [40].

Malware využíval Janus k tomu, aby obešel antivirové kontroly v obchodě s aplikacemi *Google Play* a případný antivirový software na zařízení. Instalační balíček *malwaru* je spojený DEX soubor s APK archivem. DEX soubor je prázdný a slouží pouze k tomu, aby antivirové aplikace považovaly celý spo-

jený soubor za neškodný prázdný DEX. Samotný škodlivý kód je zašifrovaný a v APK balíčku je skrytý v adresářích pro pomocné soubory aplikace. Po spuštění nainstalované aplikace je škodlivý kód dešifrován, dynamicky nahrán do paměti a spuštěn. *Malware* poté kontaktuje svůj *command and control* server (server, který *malwaru* posílá příkazy nebo dodatečné soubory) a stáhne si z něj další škodlivý kód [40].

Zranitelnost Janus byla využita i v komplexnějším *malwaru*. Tím je například *Agent Smith*, který své jméno dostal podle replikující se postavy z filmu *Matrix* [48]. Tento *malware* objevili v červenci 2019 bezpečnostní experti ze společnosti *Check Point*. *Agent Smith* využívá zranitelnost Janus, ale také *Bundle*, která umožňuje instalaci aplikací bez vědomí uživatele, a *Man-in-the-Disk*, která umožňuje sledovat změny na SD kartě a zaměnit na ní soubor s aktualizací jiné (legitimní) aplikace. Cílem *malwaru* je napadnout co nejvíc aplikací v zařízení a zobrazovat v nich reklamy [49]. Postup *malwaru Agent Smith* je na obrázku 1.14.



Obrázek 1.14: Diagram postupu škodlivé aplikace *Agent Smith* [49].

Agent Smith má modulární strukturu. Tím, že se vydává za jiné aplikace (hry, přehrávače), nejprve přiměje uživatele, aby sám nainstaloval základní

modul *malwaru*, tzv. *dropper*. Modul *dropper* obsahuje kód modulu *loader* a zašifrovaný kód modulu *core*, který se vydává za data souboru JPG. Modul *loader* dešifruje *core* a pomocí legitimních mechanismů OS Android pro práci s velkými DEX soubory ho přidá ke svému kódu a spustí [49].

Modul *core* kontaktuje *command and control* server a získá od něj seznam cílových aplikací – aplikací, pro které existuje připravený škodlivý kód. *Core* se pokusí nalézt tyto aplikace v systému, rozbalit jejich instalační APK archiv a *disassemblovat* jejich DEX soubor do jazyka *Smali* (viz sekce 2.2.2). Části *disassemblovaných* souborů poté nahradí škodlivým kódem pro konkrétní aplikaci, který získá od *command and control* serveru. Škodlivý kód přidaný do aplikace způsobí, že pokaždé, když aplikace vytvoří novou *aktivitu* (obrazovku grafického rozhraní), bude přes ni zobrazena reklama. *Smali* kód aplikace s přidaným škodlivým *Smali* kódem je následně zkompilován a zabalen do APK, ze kterého je extrahován soubor `classes.dex` [49].

K extrahovanému DEX souboru je pomocí mechanismů využití zranitelnosti Janus znázorněných na obrázku 1.12 na straně 26 připojen původní APK balíček aplikace. Protože zranitelnost Janus umožňuje změnu pouze u souborů, které se v původním APK již vyskytovaly, a *Agent Smith* potřebuje i nové pomocné soubory, jsou tyto soubory vloženy do výplně DEX před APK archiv, který je na úplném konci výplně DEX souboru [49].

Vytvořený soubor je pomocí několika zranitelností typu *Bundle* nainstalován bez vědomí uživatele. Systém ho díky zachovanému podpisu *APK Signature Scheme v1* původního APK považuje za aktualizaci nainstalované aplikace. Pokud zranitelnosti pro instalaci balíčku nelze využít, je balíček malwarem nainstalován pomocí metody *Man-in-the-Disk* [49].

Když je napadená aplikace spuštěna, *Agent Smith* zamezí instalaci jejích dalších aktualizací, aby jimi nebyl přepsán. Toho dosáhne tak, že sleduje adresář, kam se ukládají balíčky s aktualizacemi, a pokud se v něm objeví nějaký soubor, je *malwarem* ihned odstraněn [49].

Bezpečnostní experti ze společnosti *Check Point* odhadují, že díky tomu, že *Agent Smith* nenakazí pouze jednu aplikaci v zařízení, ale úplně všechny z jeho seznamu, které v zařízení nalezne, mohlo být *malwarem* nakaženo celkem 2,8 miliardy aplikací na zhruba 25 milionech zařízení. *Agent Smith* byl nalezen v internetovém obchodě s aplikacemi *9Apps*, kde bylo uživatelům nabízeno více než 360 různých variant aplikace obsahující jeho modul *dropper*. *Malware* cílil převážně na indické uživatele, ale byl nalezen i v USA, Saúdské Arábii, Austrálii a ve Spojeném království [49].

Kapitola 2

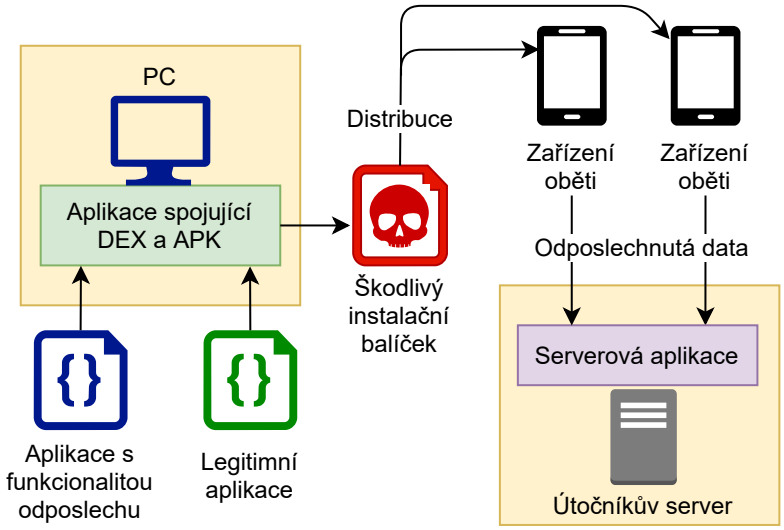
Návrh

Cílem této práce je provést útok na zařízení s operačním systémem Android, který využije zranitelnost Janus. V této kapitole je popsáno, z jakých částí se útok skládá a jak jsou navrženy jeho klíčové komponenty.

Útočník chce narušit soukromí uživatele zařízení, konkrétně provádět vzdálený odposlech. Jeho cílem je vytvořit takovou škodlivou aplikaci, kterou distribuuje do zařízení více obětí. Tato aplikace má získávat data z mikrofonu zařízení a odesílat je na útočníkův server. Na serveru útočníka je potom spuštěna aplikace, která data přijímá z více zařízení najednou, dokáže přijatá data přehrávat v reproduktorech a ukládat do souborů. Pro ukládání do souborů se přijatá data s předstihem ukládají do kruhového pole, které zaručí, že i když útočník spustí nahrávání až po tom, co z reproduktoru slyšel pro něj zajímavou informaci, bude tato informace obsažena i v souboru se záznamem. Samotný útok se potom skládá ze tří základních modulů – aplikace s logickou odposlechu, serverové aplikace pro příjem dat a aplikace spojující DEX a APK podle principů využití zranitelnosti Janus. Činnost těchto modulů je znázorněna na obrázku 2.1.

Na počátku musí útočník implementovat samotnou funkcionalitu odposlechu. Vytvoří tedy aplikaci pro OS Android, která ji obsahuje. Tuto funkcionalitu vloží do legitimní aplikace, čímž ale vznikne instalační balíček s narušeným podpisem (modifikovanými soubory s kódem). Útočník tedy kód upravené aplikace použije pro vytvoření škodlivého instalačního balíčku, který využívá zranitelnost Janus – spojí kód upravené aplikace s instalačním balíčkem neupravené legitimní aplikace. Tímto spojením vznikne škodlivý instalační balíček, který využívá zranitelnost Janus vydává se za legitimní aplikaci. Přitom ale odposlouchává uživatele zařízení a odposlechnutá data odesílá na server útočníka. Jak je vysvětleno v sekci 2.2.3, pro spojení souborů je potřeba samostatná aplikace.

Spolu se škodlivým balíčkem musí útočník vytvořit aplikaci pro svůj server, která bude od škodlivé aplikace pro OS Android přijímat odposlechnutá data a dále je zpracovávat (přehrávat, ukládat do souboru). Aplikaci pro příjem



Obrázek 2.1: Diagram znázorňující průběh útoku. Útočníkem vytvořená aplikace obsahující funkcionalitu odposlechu (modrá) se podle principů využití zranitelnosti Janus spojí s legitimní aplikací třetí strany (zelená). Vznikne škodlivý instalační balíček, který je distribuován na zařízení obětí útoku. Na těchto zařízeních je škodlivý balíček spuštěn a odposlouchávaná data jsou odesílána na útočníkův server, kde jsou zpracovávána serverovou aplikací.

dat musí útočník nasadit na svůj server, kde aplikace nepřetržitě poběží. Podrobnosti o této aplikaci jsou v sekci 2.3.

Útočník dále musí škodlivý instalační balíček rozšířit do zařízení svých vybraných obětí. V momentě, kdy se škodlivá aplikace nainstaluje a je spuštěna, začne na útočníkův server posílat data.

2.1 Volba cíle útoku

Při využití zranitelnosti Janus, jak bylo zmíněno v sekci 1.5.1, jsou možnosti útočníka omezeny oprávněními cílové legitimní aplikace. Protože útočník nemůže pozměnit soubor `AndroidManifest.xml`, ve kterém jsou žádosti o oprávnění vypsány, musí najít takový cíl útoku, který o oprávnění potřebná pro útočníka sám žádá. V této sekci jsou popsána dvě oprávnění, která je nutně potřeba získat pro úspěšné provedení útoku, který je cílem této práce. V sekcích 2.1.1 a 2.1.2 jsou popsány i další komponenty, které by útočník měl hledat u cílové legitimní aplikace, protože mu rozšíří možnosti a celkově zlepší účinnost útoku.

Prvním oprávněním, které je nutné pro provedení tohoto útoku, je `android.permission.RECORD_AUDIO`. Toto oprávnění umožňuje aplikaci přístup k audio vstupu zařízení. *Protection level* `RECORD_AUDIO` v tabulce

1.1 na straně 14 je *dangerous* [50], takže se uživateli, který na svém zařízení má operační systém Android verze 6.0 a vyšší, zobrazí při pokusu o přístup k mikrofonu dialog, ve kterém může přístup povolit, nebo zamítnout. Cílovou aplikaci je tedy třeba volit tak, aby žádost o přístup k mikrofonu nevzbudila v oběti podezření.

Druhým nutným oprávněním je `android.permission.INTERNET`. Toto oprávnění umožňuje aplikaci otevírat síťová spojení a je potřeba k tomu, aby odposlechnutá data mohla být odeslána na útočníkův server. *Protection level* tohoto oprávnění (viz tabulka 1.1 na straně 14) je *normal* [50]. Aplikace, která žádost o toho oprávnění má ve svém `AndroidManifest.xml`, tedy přístup k síti získá automaticky.

2.1.1 Odposlech na pozadí

Kromě dvou výše zmíněných oprávnění, bez kterých by vzdálený odposlech nefungoval, existují ještě další prvky, které napomůžou úspěšnému provedení útoku. Legitimní cílová aplikace je ale musí mít uvedeny ve svém `AndroidManifest.xml`. Jedním z nich je například tzv. služba (*service*). Následující popis služby je převzat z oficiální dokumentace OS Android [51].

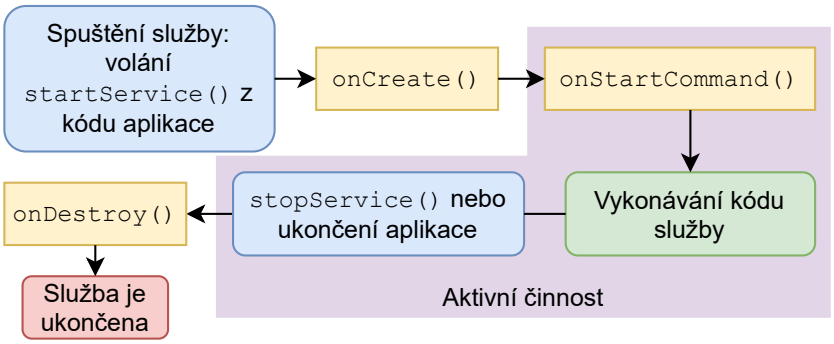
Služba je komponenta aplikace, která nemá grafické rozhraní. Využívá se k tomu, aby jako samostatný proces na pozadí prováděla činnosti, které nepotřebují interakci uživatele. *Service* může zůstat spuštěná i po tom, co uživatel ukončí aplikaci, která službu spustila.

Existují tři hlavní druhy služeb. Prvním je tzv. *background service*, tedy služba spuštěná na pozadí. Tyto služby se typicky používají pro provedení pomocných činností trvajících delší dobu, které nepotřebují činnost uživatele.

Druhým typem služeb je tzv. *foreground service*, tedy služba běžící na popředí. Tyto služby mají v systému typicky vyšší prioritu než služby na pozadí a zůstávají spuštěné i v momentě, kdy systém z paměti odstraňuje procesy, které jsou dlouhou dobu nečinné. Uživatel může se službou na popředí komunikovat pomocí grafické notifikace v hlavní liště, kterou *foreground service* musí po dobu své aktivity zobrazovat.

Třetím typem služby v operačním systému Android je tzv. *bound service*. Jde o poddruh služby na pozadí, na kterou se po jejím spuštění může navázat jiná komponenta aplikace než ta, která službu původně spustila. Po navázání se na službu (*bind*) může komponenta aplikace se službou interagovat pomocí rozhraní pro meziprocesovou komunikaci.

Každá služba v operačním systému Android je implementována jako třída dědící od třídy `Service`. Od ní zdědí *callback* metody pro ovládání služby volané při určitých událostech v jejím životním cyklu, jako např. `onStartCommand` (zavolá se po spuštění služby, obsahuje její hlavní kód), `onCreate` (volá se jednorázově při vytvoření objektu služby), `onBind` (volá se při navázání na službu) nebo `onDestroy` (volá se před zrušením objektu služby). Znázornění životního cyklu služby na pozadí je na obrázku 2.2.



Obrázek 2.2: Diagram znázorňující životní cyklus služby na pozadí.

V závislosti na tom, jakou hodnotu vrátí metoda `onStartCommand`, je se službou v operačním systému dále zacházeno. Pokud vrátí konstantu `START_NOT_STICKY`, systém službu zruší ihned po dokončení jejích činností. Pokud metoda vrátí `START_STICKY`, je služba systémem po jejím ukončení opět obnovena (`START_STICKY` říká, že o spouštění a ukončování této služby je rozhodováno v kódu aplikace).

Takto je chování popsáno v oficiální dokumentaci [51] a vývojáři aplikací pro OS Android ho takové očekávají. Někteří výrobci zařízení s operačním systémem Android ale tato pravidla nedodržují. Z důvodu úspory baterie jsou na zařízeních těchto výrobců služby ukončeny předčasně (např. po ukončení hlavní aplikace i přesto, že jejich `onStartCommand` vrací `START_STICKY`).

Pokud uživatel chce tomuto nezvyklému chování předejít, musí v nastavení systému explicitně zaškrtnout, že o úsporu baterie u konkrétní aplikace nestojí. Toto rozhodnutí je však na uživateli a vývojář aplikace s ním nic nezmůže, protože zrušení těchto opatření pro úsporu baterie nelze provést ani vyžádat z kódu aplikace. Proto vznikla iniciativa s názvem *Don't kill my app!* [52], pomocí které se vývojáři aplikací pro operační systém Android snaží tlačit na výrobce zařízení, aby ve svých verzích operačního systému implementovali procesy na pozadí korektně podle oficiální dokumentace.

Společnost *Google* se snažila tento způsob úspory baterie alespoň standardizovat a v OS Android 6.0 představila úsporný mód. Někteří výrobci na něj ale nedbají a dále používají svůj vlastní způsob úspory baterie. V žebříčku *Don't kill my app!*, který řadí výrobce zařízení s operačním systémem Android podle toho, jak nepřívětivým způsobem služby v systému implementují, jsou v době psaní této práce na předních místech společnosti *Huawei*, *Samsung*, *OnePlus* a *Xiaomi* [52].

Aplikace, která službu obsahuje (jedna z jejích tříd dědí od třídy `Service`), ji musí mít uvedenou ve svém souboru `AndroidManifest.xml`. Aplikace musí mít pod XML elementem `<application>` zapsán element `<service>`. Celá položka v manifestu pro službu jménem `ExampleService`

by byla `<service android:name=".ExampleService"/>` [51]. Při využití zranitelnosti Janus tedy nelze spojit útočníkův DEX soubor, který využívá služby, s APK archivem, který služby nemá. Pokud útočník chce, aby jeho odposlech běžel kontinuálně na pozadí, měla by cílová legitimní aplikace službu obsahovat.

2.1.2 Obnovení po restartu systému

Aplikace v operačním systému Android mají možnost reagovat na systémové události, např. změnu síťového připojení, instalaci nové aplikace nebo nabíjení baterie. Tyto události vyšlou v systému zprávu (*broadcast*), která může být přijata aplikacemi, které mají zaregistrovaný tzv. *broadcast receiver*, tedy třídu, která na zprávy reaguje [53].

Broadcast receiver se registruje v souboru `AndroidManifest.xml` pod elementem `<receiver>`. V registraci je nutno uvést, která třída patřící aplikaci bude na *broadcast* zprávy reagovat (ta musí být odvozena od třídy `BroadcastReceiver`) [53].

Dále je v registraci uvedeno, jaký *intent* (abstraktní popis operace, která se má provést) aplikace má přijímat. Tím aplikace docílí toho, že jí budou posílány pouze ty systémové zprávy, které jsou pro ni zajímavé. Jedním z těchto popisů zpráv, které lze do registrace *broadcast receiveru* uvést, je `android.intent.action.BOOT_COMPLETED`. Díky němu aplikace obdrží zprávu pokaždé, když se spustí operační systém [53]. Příklad položky v `AndroidManifest.xml` je ve výpisu 2.1.

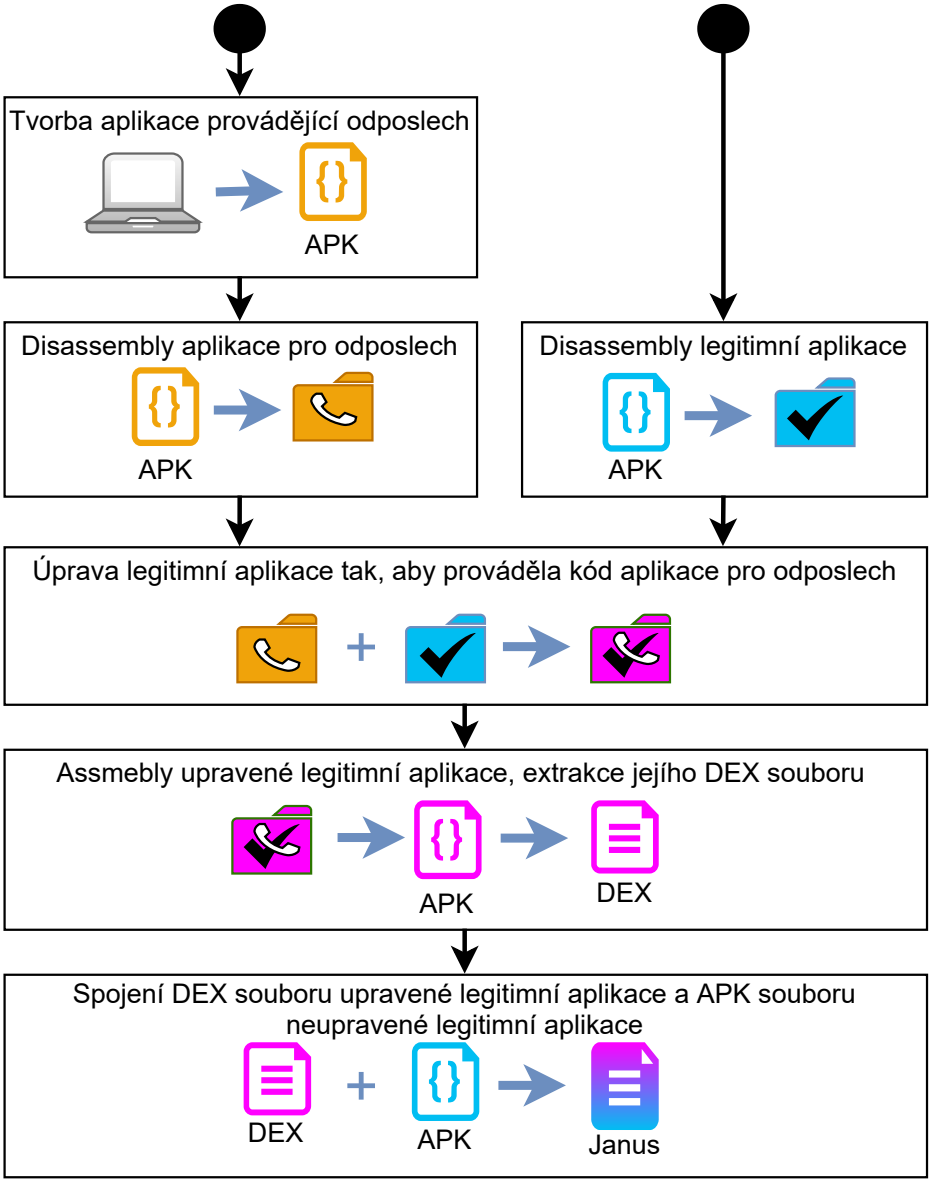
```
<receiver
    android:name="com.example.application.
       BootComplete"
    android:enabled="true"
    android:exported="false">
    <intent-filter>
       <action android:name="android.intent.action.
          BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

Výpis 2.1: Položka v `AndroidManifest.xml` – aplikace má registrovaný *broadcast receiver* implementovaný třídou jménem `BootComplete` a chce pomocí něj získávat zprávy o startu systému (`BOOT_COMPLETED`) [53].

Pro útok, který je cílem této práce, by bylo vhodné najít legitimní aplikaci, která má zaregistrovaný *broadcast receiver* na akci `BOOT_COMPLETED`. Reagující třída by mohla po startu systému znovu spustit proces, který bude odposlouchávat a odesílat data na útočníkův server.

2.2 Příprava škodlivého instalačního balíčku

Škodlivý instalační balíček je nejdůležitějším souborem tohoto útoku. Jak bylo vysvětleno v předchozí sekci, útočník musí pečlivě volit aplikaci, kterou použije jako cílovou. V této sekci je rozvedeno, jakou podobu by měl mít samotný odposlech a jakým způsobem se škodlivý balíček z cílové aplikace vytvoří. Diagram průběhu přípravy škodlivého souboru (DEX spojeného s APK) je na obrázku 2.3.



Obrázek 2.3: Diagram fází přípravy škodlivého instalačního balíčku.

Při přípravě škodlivého instalačního balíčku (DEX + APK) musí útočník nejprve vytvořit první ze tří hlavních modulů tohoto útoku – aplikaci pro OS Android, která provádí samotný odposlech. Přidáním její funkcionality je upravena legitimní aplikace, díky čemuž uživatel na první pohled nepozná, že je v jeho zařízení nainstalovaná aplikace škodlivá. Podrobnosti o návrhu aplikace s funkcionalitou odposlechu jsou v sekci 2.2.1.

Aplikaci pro odposlech vytvořenou útočníkem a legitimní aplikaci, která je použita jako cíl útoku, je nutné *disassemblovat*[15] – z APK balíčků se extrahují veškerá data a uloží se do strukturovaných adresářů. Následně se binární soubor `classes.dex` obou aplikací pomocí specializovaných nástrojů převede do čitelné podoby, konkrétně do jazyka *Smali*. O jazyku *Smali* je více podrobností v sekci 2.2.2.

Aby oběť útoku na první pohled nepoznala, že má ve svém zařízení s OS Android nainstalovanou jinou aplikaci, než původně zamýšlela, je škodlivý APK balíček vytvořen na základě původní aplikace. Legitimní cílová aplikace i aplikace pro odposlech jsou nyní *disassemblované* do jazyka *Smali*. Funkcionalitu aplikace pro odposlech je na úrovni jazyka *Smali* možné přenést do *Smali* kódu legitimní cílové aplikace. Více podrobností o přenosu funkcionality do cílové aplikace je v sekci 2.2.2.

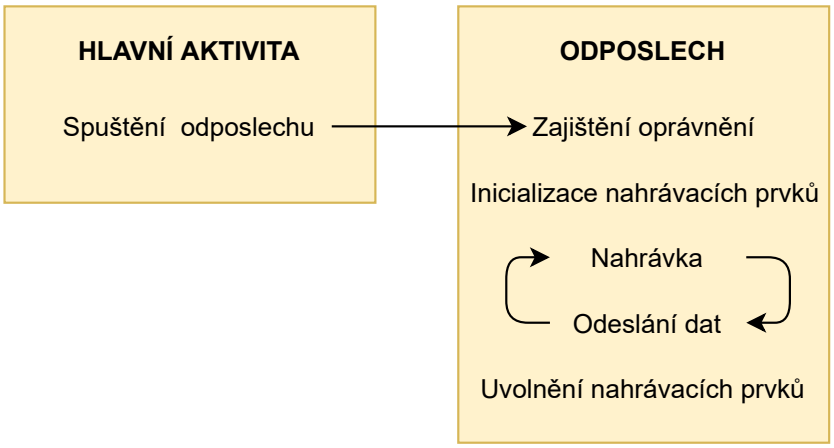
Po přidání funkcionality pro odposlech do legitimní cílové aplikace se její *Smali* kód převede zpět do *Dalvik* bytekódu (DEX souboru) a znovu se sestaví instalační balíček APK. Podpis tohoto balíčku je kvůli úpravám kódu zneplatněn, takže není možné jej nainstalovat do zařízení. Z balíčku lze ale extrahovat DEX soubor, který se použije pro využití zranitelnosti Janus. DEX soubor upravené legitimní aplikace se pomocí principů zranitelnosti Janus spojí s APK archivem neupravené legitimní aplikace. Při instalaci tohoto spojeného souboru se pak zkontroluje podpis neupravené legitimní aplikace, ale nainstaluje se DEX soubor upravené aplikace. Upravený kód obsahuje stejné třídy jako neupravený kód, takže nainstalovaná aplikace nebude mít při spuštění problém s nekonzistencí souboru `AndroidManifest.xml`, kde jsou vypsány služby, *broadcast receivery* a *content providery* obsažené v aplikaci. Více podrobností o spojení APK a DEX je v sekci 2.2.3.

2.2.1 Aplikace obsahující logiku odposlechu

Jak je vidět na obrázku 2.3, nejprve je nutné vytvořit aplikaci pro operační systém Android, která obsahuje funkcionalitu samotného odposlechu. Jediným úkolem této aplikace je co nejdelší dobu získávat data z mikrofonu zařízení a odesílat je na server útočníka. Jak bylo zmíněno výše, škodlivý DEX soubor

[15] Jak je popsáno v kapitole 3.4, zvolené legitimní aplikace mají otevřený zdrojový kód, takže by všechny úpravy legitimní aplikace bylo možné provést přímo na úrovni jazyka Java. Aby se otevřenost zdrojového kódu nestala podmínkou pro úspěch tohoto útoku, je aplikacemi s otevřeným zdrojovým kódem pracováno tak, jako by útočník měl přístup pouze k podepsanému instalačnímu APK archivu.

je vytvořen za pomoci aplikace legitimní (viz obrázek 2.3). Legitimní aplikace a aplikace pro odposlech se *disassemblují* a funkcionalita aplikace pro odposlech se vloží do *disassemblované* legitimní aplikace. Aby přesun byl univerzálně proveditelný pro co největší množství legitimních aplikací, je celý odposlech realizován jedinou třídou. Z tohoto důvodu má aplikace pro odposlech pouze dvě části – hlavní aktivitu a třídu realizující odposlech (viz obrázek 2.4).



Obrázek 2.4: Hlavní části aplikace realizující odposlech.

První částí aplikace je její hlavní aktivita. Hlavní aktivita se v aplikaci nachází pouze proto, aby úspěšně proběhla její kompilace a aplikace mohla být samostatně odladěna. Hlavní aktivita funguje pouze jako vstupní bod aplikace, který spustí kód třídy pro odposlech.

Druhou částí je samotná třída pro odposlech. Tato třída obsahuje veškerou funkcionalitu, která je pro odposlouchávání potřeba. Po spuštění hlavní aktivitou zajistí nejprve potřebná oprávnění (nahrávání zvuku, odesílání dat) a poté inicializuje objekty potřebné pro nahrávání. Následně v cyklu nahrává data z mikrofonu a odesílá je pomocí UDP paketů na útočníkův server. Pokud je cyklus ukončen, uvolní se všechny na začátku inicializované objekty. Veškerý kód této třídy se přidá do výše zmíněné *disassemblované* legitimní aplikace.

2.2.2 Úprava funkcionality legitimní aplikace v jazyce *Smali*

Bytekód *Dalvik* používaný operačním systémem Android je binární strojový kód pro virtuální stroje *Dalvik* a *Android Runtime*. Protože je binární, je pro člověka obtížně čitelný, a proto byly vytvořeny nástroje *Smali* a *Baksmali*[16], které převádějí *Dalvik* bytekód do jazyka *Smali* (čitelné podoby bytekódu) a zpět do binární podoby [54].

[16]Islandsky *assembler* a *disassembler* [54].

Na rozdíl od virtuálního stroje jazyka Java, který má zásobníkově orientovanou architekturu, je architektura virtuálního stroje *Dalvik* (i *Android Runtime*) registrově orientovaná. V jazyce *Smali* tedy existují 32bitové proměnné představující registry. Protože v DEX souboru jsou uvedena jména metod, členských proměnných a typů, pracuje s těmito jmény i jazyk *Smali* a jeho kód je tak blízký zdrojovému kódu aplikace v jazyce Java [54].

Během útoku, který je cílem této práce, se upravuje funkcionalita legitimní aplikace tím, že je do jejího *disassemblovaného* DEX souboru vložen *Smali* kód části útočníkovy aplikace pro odposlech. Pro útočníka je výhodné využít pro odposlech službu (*service*), protože ta, pokud její metoda `onStartCommand` vrací hodnotu `START_STICKY`, ke svojí činnosti nevyžaduje, aby v systému byla spuštěna její aplikace.

Jak je popsáno v sekci 1.5.1, zranitelnost Janus neumožňuje přidat do aplikace vlastní služby. Seznam služeb (tříd dědících od třídy `Service`) je totiž uložen v souboru `AndroidManifest.xml`, který bez zneplatnění podpisu APK balíčku nelze pozměnit. Pokud útočník chce, aby jeho kód běžel v procesu na pozadí, musí obětovat část funkcionality legitimní aplikace a některou z jejích služeb přepsat svou vlastní.

2.2.3 Spojení souborů DEX a APK

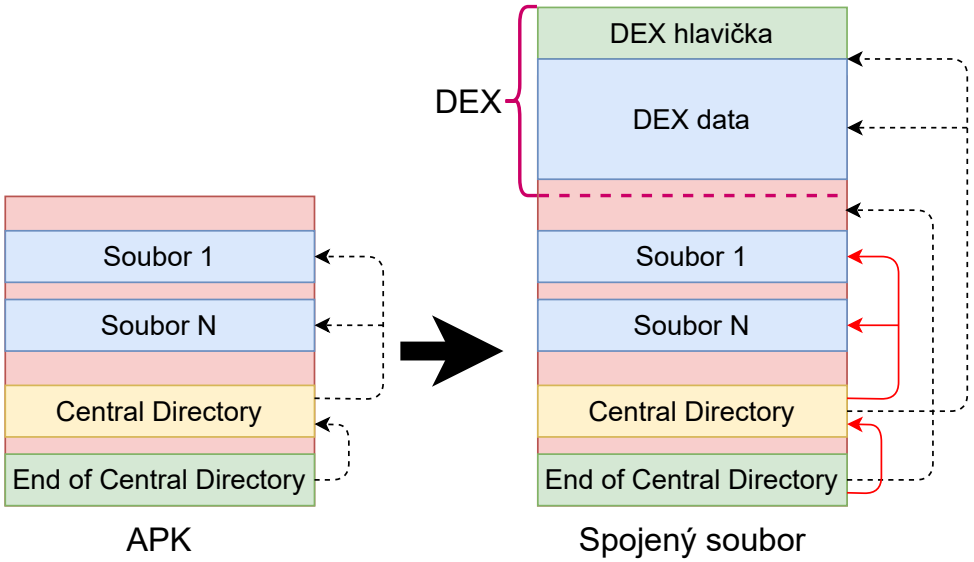
Jak bylo zmíněno v sekci 1.5.2, při samotném využívání zranitelnosti Janus, tedy při spojování DEX souboru a APK archivu, je potřeba APK balíček pozměnit. ZIP sekce souboru APK totiž obsahují ofsety, které jsou počítané od začátku souboru (viz sekce 1.3.1). Přidáním DEX souboru před APK archiv se ofsety zneplatní a je potřeba je opravit (viz obrázek 2.5).

Spojení souborů tedy není pouze triviálním spojením jejich dat. Proto je v rámci útoku vytvořena aplikace pro příkazovou řádku, která načte oba soubory, přepíše ZIP ofsety v APK, spojí data souborů a zapíše je do výstupního souboru. Ta se skládá ze čtyř hlavních tříd (viz obrázek 2.6).

První částí aplikace pro spojení DEX souboru s APK archivem je její hlavní modul, který slouží jako vstupní bod aplikace. V hlavním modulu jsou načteny soubory DEX a APK a jejich daty jsou inicializovány objekty stejnojmenných tříd. Poté jsou přepsány ofsety v objektu APK a data obou objektů jsou spojena a zapsána do výstupního souboru.

Třída DEX slouží jako datový typ pro jeden DEX soubor. Jediné, co třída DEX potřebuje, jsou data DEX souboru a jejich délka (nalezení jednotlivých sekcí nebo interpretace DEX hlavičky zde nejsou potřeba). Třída APK slouží jako datový typ pro jeden APK archiv. Objekt třídy APK obsahuje data APK archivu a jejich délku, ale navíc při své inicializaci nalezne ofsety ZIP sekcí *Central Directory*, *End of Central Directory* a *Local File Header*, které tak lze jednoduše přepsat.

Třída Janus obsahuje funkcionalitu potřebnou pro spojení obou souborů. V datech objektu APK dokáže přepsat ofsety sekcí *Central Directory*, *End of*



Obrázek 2.5: Oprava ofsetů v APK archivu po připojení za DEX soubor. Čárkovaně jsou znázorněny staré ofsety, červenou plnou čarou nové.

Central Directory a *Local File Header* každého v archivu obsaženého souboru. Dále objekt třídy Janus spojí upravená data obsažená v objektech DEX a APK a upraví jejich kontrolní součet.

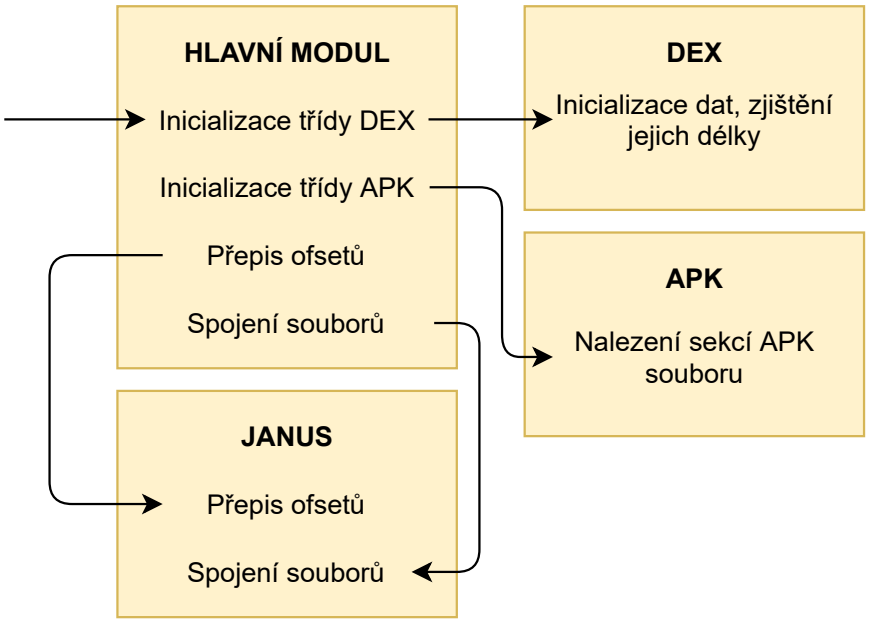
2.3 Aplikace pro příjem a zpracování odposlechnutých dat

Tato sekce se zabývá návrhem podoby aplikace, kterou má útočník nasazenou na svém serveru. Jedná se o aplikaci pro příkazovou řádku, která přijímá data od všech zařízení, která jsou v daném momentě schopna jí data poslat. Jedno zařízení odesílající data z jedné konkrétní IP adresy a portu bude dále označeno jako *spojení*.

Aplikace přijímá data od více spojení zároveň. V aplikaci lze vypisovat seznam aktuálních spojení a přepínat, které spojení má být aktivní (tedy které se má přehrávat v reproduktoru).

Spojení lze ze seznamu ručně odstraňovat, aby při omezení maximálního počtu spojení byla přijímána data pouze od těch, která jsou pro útočníka zajímavá. Spojení, od kterých po nějakou dobu data nepřijdou, jsou ze seznamu automaticky odstraňována.

Dále lze spustit a ukončit záznam dat spojení do souboru. Díky vnitřnímu kruhovému poli, do kterého jsou přijatá data průběžně ukládána, mohou do souboru být uložena i data, která byla přijata ještě před spuštěním nahrávání.



Obrázek 2.6: Hlavní třídy aplikace, která slouží pro spojení DEX souboru s APK archivem.

Při spuštění lze na příkazové řádce pomocí argumentů ovlivnit velikost kruhového pole, počet vláken zpracovávajících data a maximální počet spojení, z nichž jsou data zpracovávána. Aplikace se skládá ze tří stěžejních částí, viz obrázek 2.7.

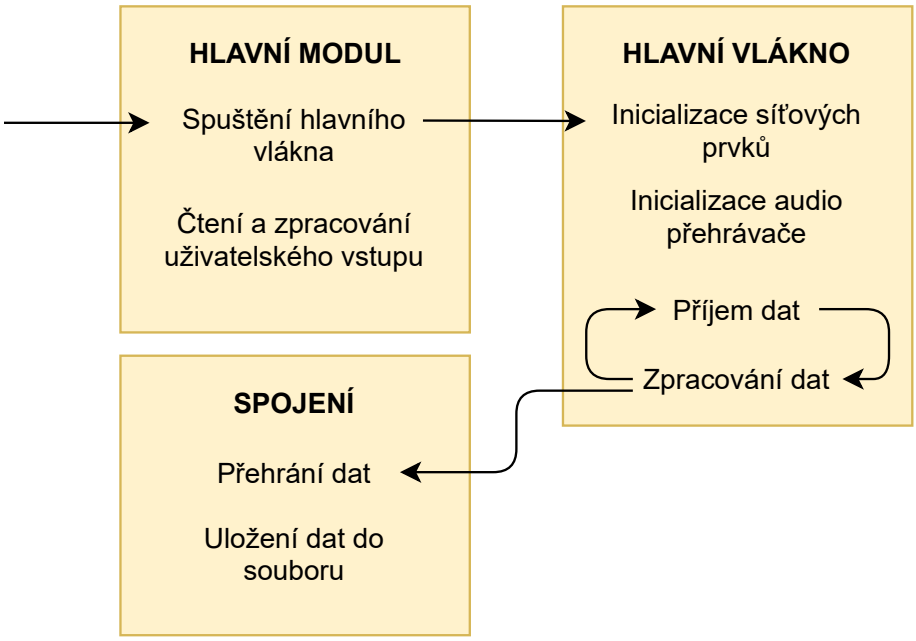
Hlavní modul aplikace slouží jako její vstupní bod. Na začátku spustí hlavní vlákno (viz dále) a poté ve smyčce čte a zpracovává vstup uživatele. Uživatel tak má nad aplikací kontrolu a může prohlížet a modifikovat seznam připojených zařízení a ovládat nahrávání či přehrávání vybraných spojení.

Objekt třídy reprezentující hlavní vlákno nejprve inicializuje síťové prvky potřebné pro příjem odposlechnutých dat a poté prvky potřebné pro přehrávání zvuku. Ve smyčce potom přijímá data ze sítě a v odděleném vlákně nechává objekt třídy reprezentující aktuální spojení data dále zpracovat.

Objekt třídy reprezentující aktuální spojení zpracovává data přijatá od jednoho konkrétního zařízení. Podle toho, jaké uživatelské příkazy objekt dostane, může data buď ignorovat, přehrát v reproduktorech, nebo je uložit do souboru.

2.4 Doručení škodlivé aplikace do zařízení oběti

Cílem útočníka je doručit škodlivou aplikaci do zařízení oběti tak, aby oběť nezaznamenala, že se jedná o škodlivou aplikaci a ne o legitimní. Známé internetové obchody s aplikacemi jako např. *Google Play* již využití zranitelnosti



Obrázek 2.7: Hlavní třídy aplikace, která slouží pro příjem a zpracování odposlechnutých dat.

Janus rozpoznají a škodlivé instalační balíčky nedistribuují [41]. Útočník tak musí aplikaci distribuovat jinou cestou, například pomocí vlastních webových stránek nebo internetového obchodu s aplikacemi, který využití zranitelnosti Janus nedetekuje. Škodlivou aplikaci může útočník díky zachování podpisu při využití zranitelnosti Janus vydávat například za dosud nezveřejněnou aktualizaci legitimní aplikace.

Útočník si dále musí zvolit, komu chce aplikaci do zařízení nainstalovat. Může se jednoduše vydat cestou instalace škodlivé aplikace do co největšího počtu zařízení, z nichž si poté vybere ty oběti, jejichž odposlech je pro něj zajímavý. Tento způsob má však tu nevýhodu, že při velkém množství napadených zařízení bude pro útočníka nepřehledné, které spojení chce v kterou chvíli odposlouchávat. Navíc mu (v závislosti na útočníkovi dostupné infrastruktuře) může velké množství příchozích dat zahltit síťové prvky nebo výpočetní prostředky serveru.

Alternativním způsobem distribuce škodlivé aplikace je distribuce cílená. Útočník si může zvolit konkrétní oběti, které přiměje, aby si škodlivou aplikaci do zařízení nainstalovaly. Pro distribuci může použít například tzv. *spear phishing*, tedy podvodné zprávy cílené na konkrétní uživatele.

Kapitola 3

Realizace

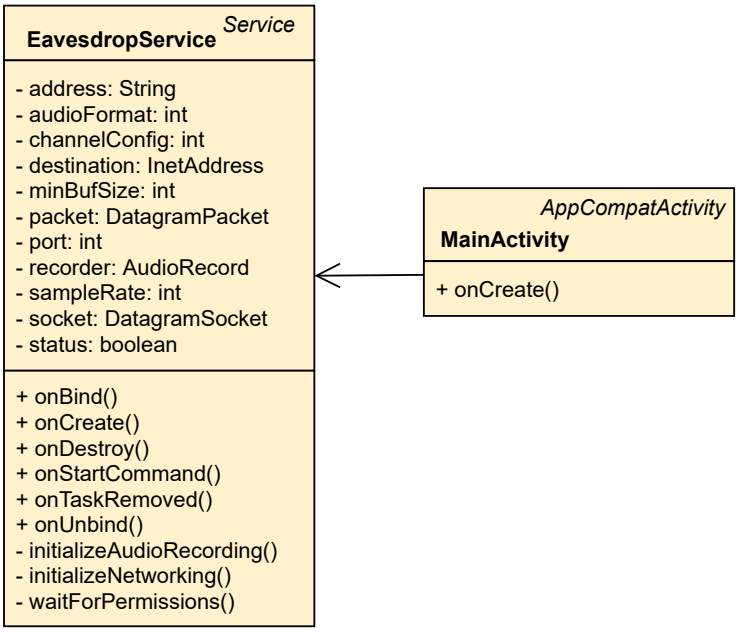
K provedení útoku je potřeba vytvořit tři aplikace – aplikaci pro OS Android obsahující logiku odposlechu, aplikaci pro útočníkův server přijímající odposlechnutá data a aplikaci, která dokáže spojit DEX soubor a APK balíček podle principů využití zranitelnosti Janus. V této kapitole je podrobně popsána implementace všech výše zmíněných modulů útoku. Aby oběť nepoznala, že si do svého zařízení nainstalovala škodlivou aplikaci, je funkcionalita útočníkovy aplikace s logikou odposlechu přesunuta do legitimní aplikace a DEX soubor této upravené aplikace je použit pro tvorbu škodlivého instalačního balíčku. Tato kapitola dále obsahuje podrobný popis, jak přesun funkcionality probíhal.

3.1 Aplikace obsahující logiku odposlechu

Pro implementaci aplikace, která obsahuje logiku odposlechu, byl zvolen jazyk Java. Aplikace byla implementována tak, aby po *disassemblování* zkompilovaného aplikačního kódu bylo možné jednoduše nalézt a vyjmout funkcionalitu, která je pro útok důležitá. Z tohoto důvodu obsahuje aplikace pouze dvě třídy – hlavní aktivitu a odposlouchávací službu (viz obrázek 3.1).

Aplikace třetích stran pro OS Android jsou zpravidla grafické. Jejich základem jsou tzv. *aktivity* – třídy, které reprezentují jednu obrazovku grafického rozhraní. Aplikace s logikou odposlechu má pouze jedinou aktivitu, a to tu hlavní, která se zobrazí po spuštění aplikace.

Na obrázku 3.2 je zobrazen průběh činností vykonávaných aplikací s logikou odposlechu. Hlavní aktivita je implementována pomocí třídy s názvem `MainActivity`. Tato třída slouží pouze jako vstupní bod aplikace při ladění. Hlavní aktivita má implementovanou jedinou metodu, a to `onCreate`. Metoda `onCreate` je jedna z tzv. *callback* metod (metod, s jejichž pomocí může aktivita reagovat na události způsobené systémem nebo uživatelem) a je volána při vytváření objektu třídy `MainActivity`.

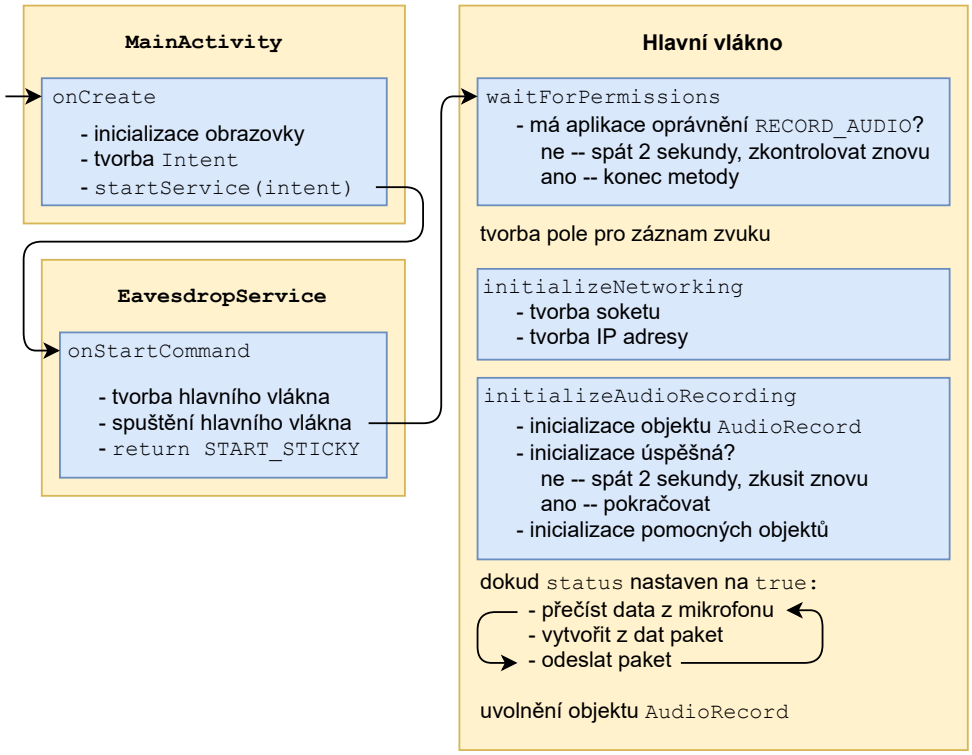


Obrázek 3.1: UML třídní diagram aplikace s logikou odposlechu.

V metodě `onCreate` je nejprve nastavena prázdná grafická obrazovka voláním funkce `setContentView`. Dále je v této metodě vyžádáno oprávnění k nahrávání zvuku (pro účely ladění aplikace – legitimní aplikace musí svá oprávnění vyžadovat samy, takže se do nich žádost nemusí přesouvat).

Jak bylo řečeno v sekci 2.1.1, některá zařízení s operačním systémem Android implementují vlastní způsob úspory baterie a navzdory specifikacím systému ukončují procesy na pozadí po ukončení jejich aplikace. Ukončování procesů na pozadí po ukončení aplikace lze u některých zařízení zamezit přidělením speciálního oprávnění nebo změnou nastavení pro danou aplikaci (každý výrobce zařízení má svůj vlastní způsob). Stav těchto oprávnění nebo nastavení nelze z aplikace ovlivnit ani zjistit a jediný způsob, jak se vývojář aplikace může pokusit o získání tohoto oprávnění nebo nastavení, je vypsat uživateli zprávu, aby toto nastavení změnil sám. Proto je do metody `onCreate` dále přidána podmínka, ve které je zjištěn výrobce zařízení, na kterém je aplikace spuštěna, a pokud jde o výrobce uvedeného v žebříčku iniciativy *Don't kill my app!* [52] (např. *Xiaomi, Huawei, OnePlus* apod.), vypíše se na obrazovku oznámení. Oznámení je typu `Toast` a je v něm vypsán text, který uživateli říká, aby v nastavení svého zřízení u této aplikace zakázal opatření úsporného módu. Oznámení se liší podle zjištěného výrobce zařízení.

Poté hlavní aktivita připraví odposlouchávací službu ke spuštění – vytvoří tzv. *intent* (abstraktní popis činnosti, která se má provést). *Intent* potřebuje dva parametry – tzv. *kontext* (rozhraní s globálními informa-



Obrázek 3.2: Diagram posloupnosti činností prováděných aplikací s logikou odposlechu.

cemi o aplikačním prostředí) a název třídy obsahující kód, který má být spuštěn (třídy `EavesdropService`). Vytvořený *intent* je použit jako argument funkce `startService`, která *intent* spustí.

Odposlech je realizován službou (viz sekce 2.1.1). To sice znamená, že část funkcionality legitimní aplikace musí být touto službou při tvorbě škodlivého APK balíčku přepsána (viz sekce 2.1.1), ale na druhou stranu útočníkovi použití služby přinese výhodu, protože díky ní odposlech funguje po delší dobu, i pokud v zařízení zrovna není spuštěna a používána škodlivá aplikace. Protože v cílových verzích OS Android (5.0 – 8.0) není zaručeno, že k mikrofonu může přistupovat více aplikací (procesů) zároveň [55], bylo by ve většině případů stejně nutné původní funkcionalitu legitimní aplikace upravit – pokud chce legitimní aplikace nahrávat zvuk, není možné, aby odposlech nahrával zároveň s ní.

Třída, která odposlech realizuje, se jmenuje `EavesdropService`. Tato třída dědí od třídy `Service`, což jí umožňuje fungovat jako proces na pozadí. Třída `EavesdropService` implementuje všechny povinné *callback* metody své nadtřídy. Metody `onCreate`, `onBind` a `onUnbind` však nejsou pro odposlech potřeba a jejich těla jsou prázdná.

Hlavní metodou třídy `EavesdropService` je `onStartCommand`, která je spuštěna zavoláním funkce `startService` v kódu aplikace. Metoda `onStartCommand` vytvoří hlavní vlákno odposlechu (objekt třídy `Thread`), spustí ho a vrátí hodnotu `START_STICKY`, která zajistí (viz sekce 2.1.1), že po ukončení aplikace, ze které byla volána funkce `startService`, systém službu znovu spustí.

V hlavním vláknu odposlechu se v metodě `waitForPermissions` nejprve počká, až aplikace získá od uživatele potřebná oprávnění. Jak bylo zmíněno v sekci 2.1, odposlech nutně potřebuje oprávnění `RECORD_AUDIO` a `INTERNET`. Oprávnění `INTERNET` je aplikaci přiděleno automaticky, ale oprávnění `RECORD_AUDIO` má *protection level* na úrovni *dangerous* (viz sekce 2.1) a uživatel ho musí od OS Android verze 6.0 aplikaci explicitně přidělit. Protože žádat o oprávnění lze v operačním systému Android pouze z aktivity, ale nikoliv ze služby [56], je nutné před přístupem k mikrofonu na oprávnění počkat. Čekání na oprávnění je implementováno jako smyčka, která každé dvě sekundy voláním `checkCallingOrSelfPermission` zkontroluje, zda aplikace oprávnění `RECORD_AUDIO` již získala. Pokud ano (volání vrátilo hodnotu `PackageManager.PERMISSION_GRANTED`), smyčka i metoda `waitForPermissions` končí.

Po získání potřebných oprávnění je vytvořeno bytové pole (proměnná `buffer`), které slouží k uložení dat získaných z mikrofonu. Při určování velikosti pole musí být brán ohled na doporučenou minimální velikost, která odpovídá zvoleným parametrům pro nahrávání zvuku (parametry podrobněji rozvedeny dále). Pro maximální velikost žádné doporučení není, ale velikostí pole je ovlivněna latence odposlechu. Latence nahrávky zvuku je totiž určena následujícím vztahem [57]:

$$\text{latence} = \frac{\text{velikost pole}}{\text{vzorkovací frekvence}}$$

Latence je při pevné vzorkovací frekvenci přímo úměrná velikosti pole. Kvůli minimalizaci latence odposlechu byla velikost pole zvolena jako minimální hodnota vhodná pro zvolené parametry. Tuto minimální hodnotu lze získat pomocí metody `AudioRecord.getMinBufferSize`.

V hlavním vláknu odposlechu následuje inicializace objektů poskytujících přístup k síti voláním metody `initializeNetworking`. Nejprve je vytvořen UDP soket (proměnná `socket` typu `DatagramSocket`) a poté je inicializována proměnná `destination`. Tato proměnná je typu `InetAddress` a při jejím vytváření je použita IP adresa útočníkova serveru, která je ve formě textového řetězce uložena v proměnné `address`, členské proměnné služby `EavesdropService`.

Následuje inicializace objektů potřebných pro získávání zvukových dat z audio vstupu zařízení v metodě `initializeAudioRecording`. Nejprve je vytvořen objekt typu `AudioRecord`, který jako argumenty dostane zdroj zvuku (`AudioSource.MIC`), vzorkovací frekvenci (zde zvolena hod-

nota 44 100 vzorků za sekundu, protože je jedinou zaručenou frekvencí podporovanou všemi Android zařízeními [58], a také zaručí vysokou kvalitu odposlechu – jde o standardní hodnotu pro hudební CD [59]), konfiguraci kanálu (`AudioFormat.CHANNEL_IN_MONO`), formát zvukových vzorků (`AudioFormat.ENCODING_PCM_16BIT`) a velikost výše zmíněného pole (`buffer`) pro uložení dat. Odposlech tedy z mikrofonu monofonně nahrává 44 100 vzorků za sekundu, kde každý ze vzorků má dva byty.

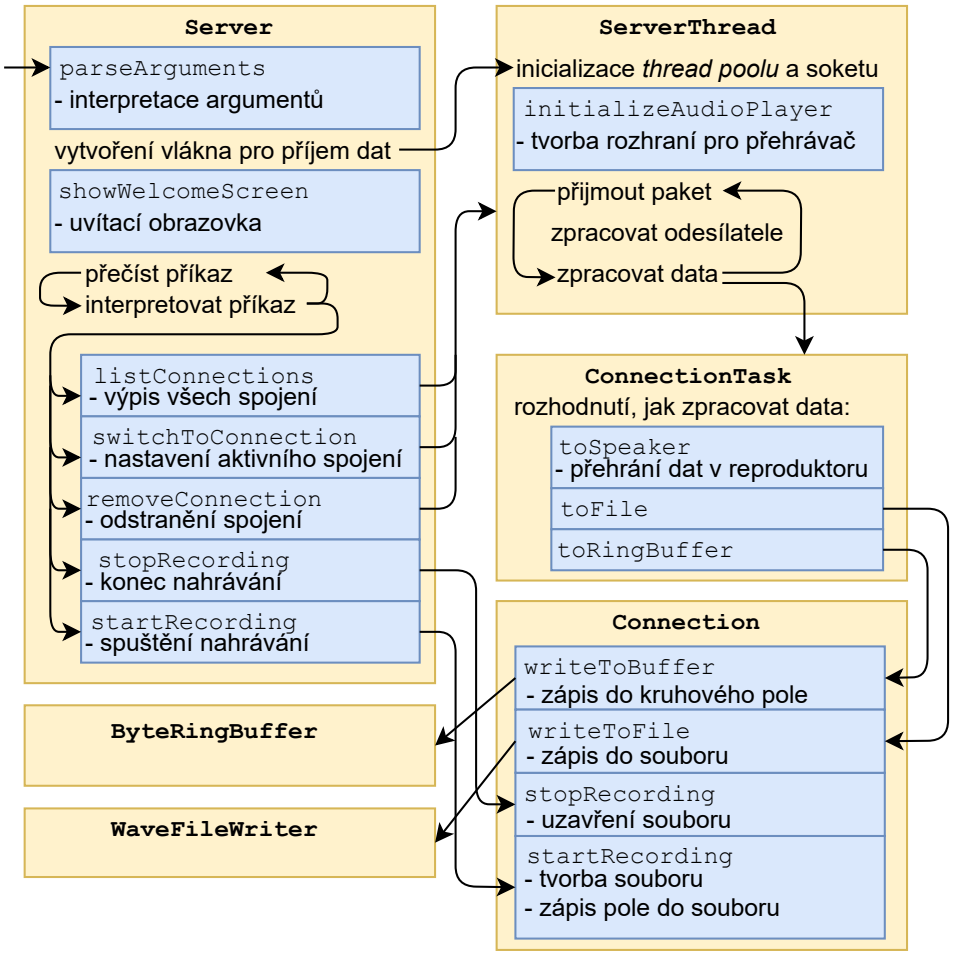
Jak bylo zmíněno výše, není zaručeno, že k mikrofonu může přistupovat více aplikací zároveň. Je tedy nutné s objektem `AudioRecord` zacházet tak, jako by ho v jeden moment mohla mít inicializovaný pouze jedna aplikace. V metodě `initializeAudioRecording` se tedy ve smyčce čeká, dokud vnitřnímu stavu rekordéru není systémem nastavena hodnota `STATE_INITIALIZED`. Po inicializaci rekordéru je zkontrolováno, zda jsou na zařízení k dispozici objekty pro potlačení šumu (`NoiseSuppressor`) a vyrušení ozvěny (`AcousticEchoCanceller`) a pokud ano, jsou inicializovány. Tím metoda `initializeAudioRecording` končí.

Dále je v hlavním vláknu odposlechu inicializovaným rekordérem spuštěno nahrávání a ve smyčce (dokud je příznak `status` nastavený na hodnotu `true`) jsou čtena data z mikrofonu (`recorder.read`). Ze získaných zvukových dat je vytvořen UDP paket (`DatagramPacket`) a ten je odeslán prostřednictvím dříve inicializovaného soketu. Pokud při odesílání nastane výjimka (cílová adresa je nedosažitelná), je ignorována a odposlech i přesto dále běží a snaží se data odesílat.

Po ukončení nahrávací smyčky nastavením příznaku `status` na hodnotu `false` je uvolněn objekt rekordéru a hlavní vlákno končí. Příznak `status` se na hodnotu `false` nastavuje z metod `onDestroy` a `onTaskRemoved`. Tyto metody patří mezi *callback* metody třídy `Service`. Nahrávání je tedy zastaveno, pokud se systém chystá službu zrušit (`onDestroy`), nebo pokud uživatel ukončil aplikaci, která službu spustila (`onTaskRemoved`). Díky tomu, že `onStartCommand` vrací hodnotu `START_STICKY`, je služba sice ihned obnovena, ale je nutné veškeré inicializované prostředky uvolnit a všechny výše popsané inicializace provést po obnovení znovu. Pokud by rekordér při ukončování služby nebyl uvolněn, po obnovení by jej kvůli výlučnému přístupu k němu nemuselo být možné používat.

3.2 Aplikace pro příjem a zpracování odposlechnutých dat

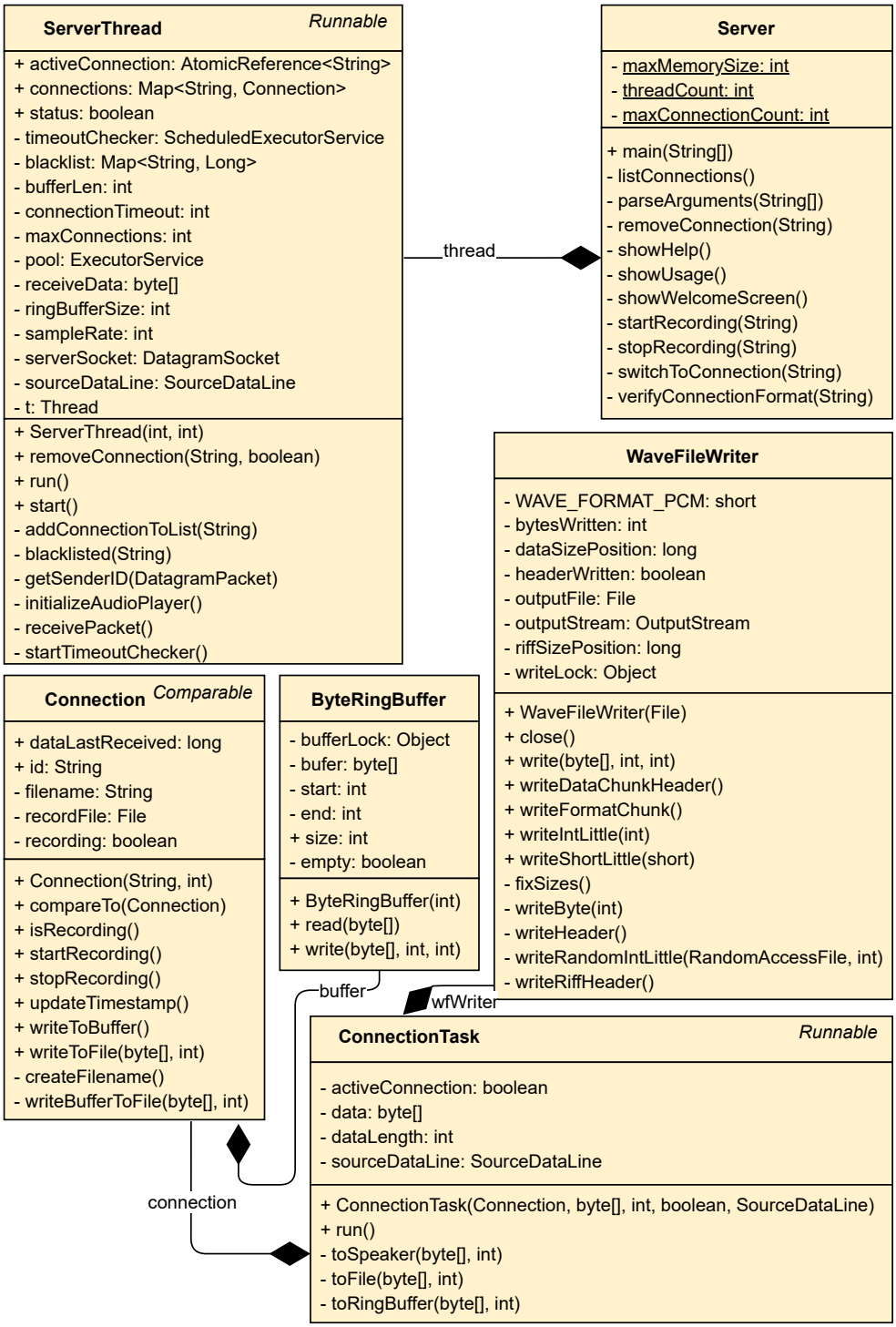
V této sekci je podrobně popsána implementace aplikace, která je spuštěna na útočníkově serveru a přijímá data od odposlouchávajících aplikací. Celá serverová aplikace byla implementována v jazyce Java. Diagram všech tříd, ze kterých se skládá, je na obrázku 3.4. Diagram znázorňující dále popsanou činnost této aplikace je na obrázku 3.3.



Obrázek 3.3: Diagram činností prováděných serverovou aplikací sloužící pro příjem a zpracování dat odposlechu.

Aplikace podporuje připojení více *spojení* zároveň. Jako *spojení* je v tomto textu a v kontextu zde popisované aplikace označováno jedno zařízení s operačním systémem Android, které odesílá odposlechnutá data na útočníkův server. Toto spojení je reprezentováno objekty třídy `Connection`. Každé spojení má svůj identifikátor – textový řetězec ve formátu `IP:port` (např. `192.168.0.1:55663`).

Každé spojení má dále své cyklické pole (objekt třídy `ByteRingBuffer`), které slouží pro uchovávání nejaktuálnější části záznamu ještě před tím, než je uživatelem spuštěno nahrávání do souboru. Soubor se záznamem tak obsahuje odposlechnutá data přijatá ještě předtím, než bylo spuštěno nahrávání. Kruhové pole `ByteRingBuffer` je běžné pole bytů. Ve chvíli, kdy je pole plné a je potřeba do něj zapsat další data, nahradí nová data nejstarší data, která



Obrázek 3.4: UML třídní diagram serverové aplikace sloužící pro příjem a zpracování dat odposlechu.

jsou v poli obsažena. Objekt `ByteRingBuffer` proto kromě samotného pole obsahuje i ukazatel na pozici v poli, která obsahuje nejstarší zapsaný byte a počet bytů, které jsou v poli obsazeny.

Pokud je spojení nahráváno, obsahuje jeho objekt i výstupní datový proud (objekt typu `WaveFileWriter`) napojený na jeho výstupní soubor. U každého spojení je dále zaznamenáván čas, kdy od něj byla naposledy přijata data, aby spojení, od kterých již více než dvě sekundy data nepřišla, mohla být odstraněna[17].

Vstupním bodem serverové aplikace je metoda `main` třídy `Server`. Na jejím začátku jsou v metodě `parseArguments` zpracovány argumenty předané při spuštění aplikace přes příkazovou řádku. Aplikaci je možné předat čtyři různé argumenty – spuštění aplikace s argumentem `-h` zobrazí nápovědu ke spouštění s ostatními třemi argumenty, argument `-t` určuje počet vláken zpracovávajících data, argument `-c` určuje maximální počet spojení, od nichž jsou data zpracovávána (prevence proti zahlcení paměti serveru), a argument `-m` udává velikost (v megabytech) kruhového pole sloužícího pro uchovávání záznamu ještě před spuštěním nahrávání.

Po zpracování argumentů je vytvořeno vlákno pro příjem dat (objekt třídy `ServerThread`), které je následně spuštěno. Hlavní vlákno aplikace běží dále – na konzoli v metodě `showWelcomeScreen` vykreslí uvítací obrazovku a poté ve smyčce čte a zpracovává příkazy uživatele.

Uživatelských příkazů je celkem sedm. Příkaz `q` ukončí celou aplikaci, příkaz `h` zobrazí nápovědu. Příkaz `l` vypíše uživateli všechna spojení, od kterých aktuálně přicházejí odposlouchávaná data. V metodě `listConnections` se od vlákna `ServerThread` získá seznam všech spojení (podrobněji dále) a ty se vypíší na obrazovku.

Příkazem `s <SPOJ>` uživatel zvolí, které spojení (`<SPOJ>`) má být označeno jako *aktivní*, tedy které se má přehrávat v reproduktorech. Volaná metoda `switchToConnection` nejprve voláním metody `verifyConnectionFormat` zkontroluje, zda uživatel zadal název spojení ve správném formátu, tedy `IP:port`. Tato kontrola probíhá za pomoci regulárního výrazu, který kontroluje korektnost IP adresy a následně je zkontrolován i rozsah čísla portu. Pokud je identifikátor spojení platný a vlákno `ServerThread` má dané spojení ve svém seznamu, je toto spojení vláknu nastaveno jako aktivní.

Příkaz `d <SPOJ>` dále slouží pro odstranění spojení `<SPOJ>`. Identifikátor `<SPOJ>` je buďto textový řetězec ve formátu `IP:port`, nebo písmeno `a`, které představuje aktuálně aktivní spojení. Při zadání příkazu `d <SPOJ>` je volána metoda `removeConnection`, ve které je zkontrolován formát identifikátoru

[17] Pokud spojení přestane posílat data, znamená to buď, že jeho aplikace byla odinstalována (tím skončí i činnost služby na pozadí), nebo že napadené zařízení s OS Android nemá přístup k internetu. V obou případech je v pořádku odstranit toho spojení ze seznamu aplikace, protože při obnovení služby bude odposlouchávající aplikaci přiřazena jiná IP adresa nebo port a v serverové aplikaci tak pro toto spojení vznikne objekt s novým identifikátorem.

<SPOJ> v metodě verifyConnectionFormat a samo odstranění je delegováno vláknu, konkrétně metodě ServerThread.removeConnection. V té je nejprve zkontrolováno, zda je dané spojení právě přehráváno nebo nahráváno a v případě, že ano, jsou tyto činnosti zastaveny. Spojení je poté odstraněno ze seznamu. Po odstranění je identifikátor spojení přidán do seznamu blacklist – protože spojení jsou UDP a jsou do seznamu serveru přidávána automaticky, seznam blacklist zaručí, že než bude ručně odstraněné spojení znovu automaticky přidáno do aplikace, uběhne alespoň minuta. To z toho důvodu, aby bylo možné při omezeném počtu spojení (argument -c) ručně promazat seznam a aby do něj odstraněná spojení nebyla automaticky přidána zpět ihned po odstranění.

Uživatelské příkazy r <SPOJ> a c <SPOJ> potom spustí a ukončí nahrávání spojení <SPOJ> do souboru. Spuštění nahrávání zprostředkovává metoda startRecording, ve které je po kontrole formátu identifikátoru <SPOJ> spuštění nahrávání delegováno na objekt reprezentující spojení, které chce uživatel nahrávat, konkrétně na jeho metodu Connection.startRecording. V této metodě je vytvořen výstupní soubor, do kterého je zapisováno přes výstupní datový proud WaveFileWriter. Soubor je ve formátu WAV a ihned při vytváření jsou do něj zapsány příslušné hlavičky tohoto formátu. Následně je do výstupního souboru zapsán obsah cyklického pole (nahrávka z doby, než bylo nahrávání spuštěno). Objekt Connection si potom nastaví příznak recording, aby vláknům zpracovávajícím příchozí data indikoval, že přijatá data mají být zapsána do jeho výstupního souboru.

Metoda stopRecording hlavního vlákna aplikace, která nahrávání zastavuje, je implementována obdobně jako výše popsaná metoda startRecording. Hlavní vlákno opět deleguje zastavení nahrávání přímo na objekt Connection, který si ve své metodě stopRecording nastaví příznak recording na false a zavře výstupní soubor. Po ukončení nahrávání je záznam uložen v aktuálním adresáři v souboru pojmenovaném IP.port_<čas>.wav, kde <čas> je časová známka počátku nahrávání ve formátu rok.měsíc.den.hodina.minuta.sekunda. Celé jméno souboru se záznamem spojení přicházejícího z IP adresy 192.168.0.1 a portu 55663, které se začalo nahrávat 28. 1. 2020 ve 12:40:59, tedy bude 192.168.0.1.55663_2020.01.28.12.40.59.wav.

Při vytváření vlákna pro příjem dat (objektu třídy ServerThread) je nejprve inicializována skupina pomocných vláken, která zpracovávají přijatá data (jejich počet lze ovlivnit pomocí výše zmíněného argumentu -t). Pomocná vlákna jsou uložena ve struktuře, která se nazývá *thread pool*, což je zásobárna obsahující pevný počet předem vytvořených vláken, díky níž aplikace funguje efektivněji, protože vláknům stačí pouze předat data ke zpracování a není nutné je s každými příchozími daty znovu vytvářet a inicializovat a po dokončení jejich činnosti je uvolňovat. V aplikaci byla tato zásobárna vláken implementována objektem Executors.newFixedThreadPool.

V hlavní metodě (`run`) vlákna pro příjem dat (`ServerThread`) je nejprve inicializován UDP soket, přes který aplikace přijímá data. Stejně jako v aplikaci s logikou odposlechu je soket implementován objektem typu `DatagramSocket`.

Následuje volání metody `initializeAudioPlayer`, ve které je inicializován objekt typu `SourceDataLine`, který slouží jako rozhraní pro přehrávání dat v reproduktorech. Inicializace je provedena se stejným zvukovým formátem, jaký je použit pro nahrávání zvuku (viz sekce 3.1), tedy 44 100 šestnáctibitových vzorků za sekundu.

Dále je v metodě `run` inicializován výše již několikrát zmíněný seznam všech spojení. Tento seznam je implementován jako hashovací tabulka, tedy tabulka, která obsahuje dvojice klíč–hodnota. Jako klíč do ní byl zvolen výše zmíněný identifikátor spojení (řetězec IP:port) a jako hodnota je použit celý objekt typu `Connection`. Protože aplikace je vícevláknová a při přístupech do tabulky by hrozily nežádoucí souběhy vláken, je seznam spojení implementován jako objekt typu `ConcurrentHashMap`, tedy hashovací tabulka s vícevláknovou synchronizací. Jako hashovací tabulka je implementován i seznam `blacklist`, do kterého jsou ukládána ručně odstraněná spojení. Klíčem v tomto seznamu je identifikátor spojení a hodnotou čas přidání do seznamu.

Ještě před spuštěním hlavní smyčky, která přijímá data, je ve vláknu `ServerThread`, v metodě `startTimeoutChecker` vytvořeno a spuštěno ještě jedno vlákno, které každé dvě sekundy zkontroluje celý seznam spojení a odstraní z něj ta, od kterých již nepřicházejí data.

Ve vláknu `ServerThread` následuje hlavní smyčka, ve které je nejprve volána metoda `receivePacket`. V této metodě je vytvořen objekt typu `DatagramPacket` a pomocí metody `socket.receive` jsou do něj uložena přijatá data. Z přijatého paketu je následně zjištěna adresa a port odesílatele, ze kterých je vytvořen identifikátor spojení. Pokud se v seznamu `connections` nenachází spojení s tímto identifikátorem, počet spojení ještě nedosáhl argumentem určeného limitu a zároveň dané spojení není v seznamu `blacklist` (popř. tam není po dobu kratší než jedna minuta), je vytvořen nový objekt typu `Connection` a je do seznamu `connections` přidán. Pokud identifikátor spojení v seznamu spojení již existuje, nalezne se objekt patřící danému spojení a dále se v této smyčce pracuje pouze s ním.

Objektu, který představuje aktuální spojení, je aktualizován čas posledních přijatých dat. Následuje tvorba objektu `ConnectionTask`, který obsahuje kód pro zpracování přijatých dat. Objektu `ConnectionTask` se do konstruktoru předají přijatá data a objekt aktuálního spojení a celý objekt `ConnectionTask` je následně předán *thread poolu*, který zařídí spuštění jeho kódu v samostatném vláknu a zpracuje tak přijatá data.

Thread pool potom v samostatném vláknu spustí metodu `run` třídy `ConnectionTask`, ve které je rozhodnuto, jak se data mají zpracovat. Pokud má objekt `Connection` představující spojení, od kterého byla data přijata, nastaven příznak pro nahrávání, jsou přijatá data

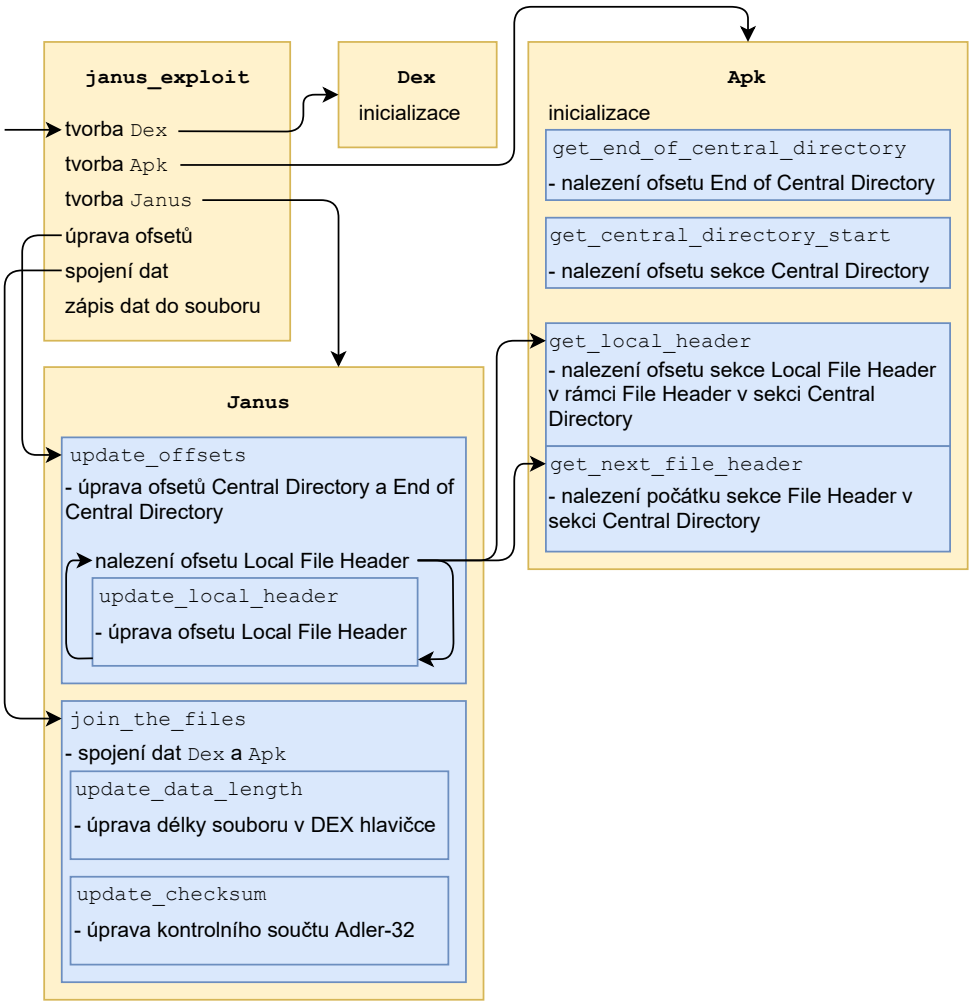
zapsána do výstupního datového proudu, který objekt `Connection` obsahuje (metoda `Connection.writeToFile`). V opačném případě jsou data zapsána do kruhového pole patřícího objektu `Connection` (metoda `Connection.writeToBuffer`). V případě, že spojení předané objektu `ConnectionTask` je aktivní, jsou přijatá data v metodě `toSpeaker` zapsána do objektu `SourceDataLine`, který zprostředkuje jejich přehrání v reproduktorech.

Jak bylo popsáno výše, aplikace je vícevláknová. Aby data byla zpracovávána konzistentně a nedocházelo k souběhům vláken, které by mohly způsobovat chyby vedoucí k pádu aplikace, bylo nutné synchronizovat přístup k prostředkům, které jsou v aplikaci využívány. Výše již bylo řečeno, že seznam všech spojení obsažený v objektu `ServerThread` je implementovaný jako `ConcurrentHashMap`, tedy hashovací tabulku, která má interně řešenou synchronizaci přístupu do jejích částí. Tato synchronizace zamezí souběhu vlákna pro příjem dat (`ServerThread`) a pomocných vláken pro jejich zpracování (`ConnectionTask`). Protože více pomocných vláken `ConnectionTask` může v jednom okamžiku zpracovávat data od stejého odesílatele, je nutné synchronizovat i samotné zpracování těchto dat. V metodě `ConnectionTask.toSpeaker` je vytvořením bloku `synchronized(sourceDataLine)` uzamčeno rozhraní pro přehrávání v reproduktorech `SourceDataLine`, takže přijatá data nemohou být v reproduktorech více vlákny prokládána mezi sebou. Ze stejného důvodu obsahuje výstupní datový proud `WaveFileWriter` objekt `writeLock`, který se v bloku `synchronized(writeLock)` uzamkne a povolí zápis do datového proudu pouze vláknu, které zámek drží. Cyklické pole `ByteRingBuffer` podobně obsahuje objekt `bufferLock`, který je uzamčen při zápisu do pole a při čtení z pole. Uzamčení je opět provedeno pomocí bloku `synchronized(bufferLock)` a zaručuje tak, že k poli může v jednu chvíli přistupovat pouze jedno vlákno.

3.3 Aplikace pro spojení souborů DEX a APK

Jak bylo popsáno v sekci 2.2.3, pro spojení souborů DEX a APK je potřeba přepsat ty části jejich dat, které obsahují relativní ofsety, kontrolní sumu nebo délku souboru. V této sekci je podrobně popsána implementace aplikace, která připojí instalační balíček APK za DEX soubor tak, aby vzniklý spojený soubor využíval zranitelnost Janus a prošel kontrolami podpisu a integrity při jeho instalaci na operačním systému Android. Aplikace spojující DEX a APK je implementována v jazyce Python 3. UML třídní diagram této aplikace je na obrázku 3.6. Diagram činností, které aplikace provádí a které jsou blíže popsány dále, je na obrázku 3.5.

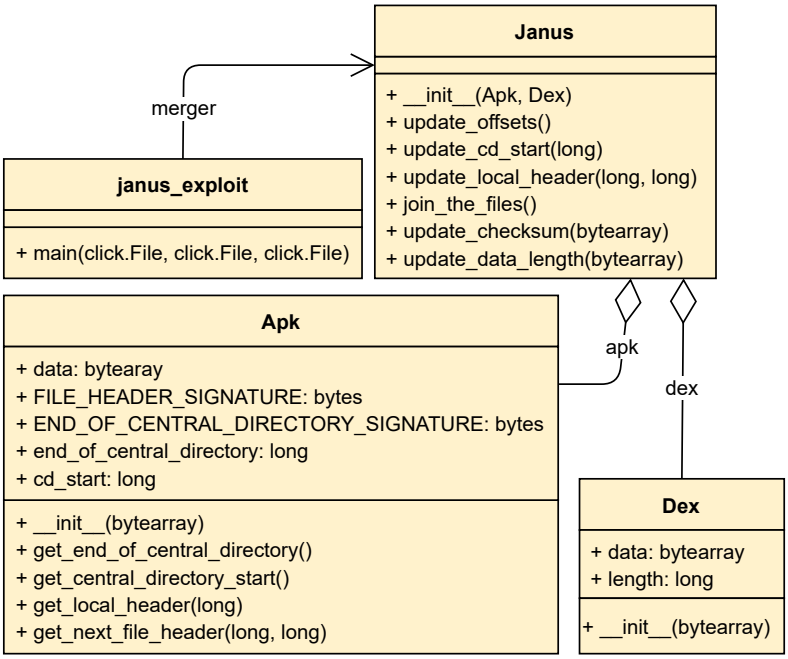
Aplikace pro spojení DEX a APK je aplikace pro příkazovou řádku. Při spuštění je nutné jí předat tři argumenty – prvním je název DEX souboru,



Obrázek 3.5: Diagram činností aplikace pro spojení DEX a APK.

druhým název APK archivu a třetím název souboru, do kterého bude zapsán výstup aplikace (spojení souborů z prvních svou argumentů). Aplikace používá knihovnu *Click* a v argumentech předané soubory jsou typu `click.File`, takže knihovna obstará jejich otevření, zavření i případné výjimky.

Vstupním bodem aplikace je soubor `janus_exploit.py`. Ten obsahuje pouze funkci `main`, ve které jsou pomocí dat souborů DEX a APK z argumentů nejprve inicializovány objekty typu `Dex` a `Apk`. Objekt `Dex` slouží jako zástupná struktura pro data DEX souboru a obsahuje pouze tato data a jejich délku. Objekt `Apk` též obsahuje data APK souboru a jejich délku, ale navíc při své inicializaci volá metodu `get_end_of_central_directory` a poté metodu `get_central_directory_start`. První z metod, `get_end_of_central_directory`, v datech objektu nalezne ofset kon-



Obrázek 3.6: UML třídní diagram aplikace pro spojení DEX a APK.

stanty `PK 05 06`, poznávací značky ZIP sekce *End of Central Directory* (viz sekce 1.3.1). Metoda `get_central_directory_start` potom nalezne ofset počátku ZIP sekce *Central Directory*. Ten je podle specifikace PKZIP formátu [60] uložen v 16. až 20. bytu sekce *End of Central Directory*.

Ve funkci `main` je dále vytvořen objekt třídy `Janus`. Tomu jsou jako argumenty předány dříve vytvořené objekty `Dex` a `Apk`.

Z funkce `main` je dále volána metoda `update_offsets` objektu `Janus`. Z té je volána metoda `update_cd_start`, ve které je upraven ofset počátku sekce *Central Directory* uložený v sekci *End of Central Directory*, který byl nalezen při inicializaci objektu `Apk`. K ofsetu je přičtena délka dat objektu `Dex` (tedy celého souboru DEX), protože archiv APK je připojen za soubor DEX a ofsety, které je v metodě `update_offsets` třeba upravit, jsou počítány od počátku souboru.

V metodě `update_offsets` následuje cyklus, ve kterém se prohledává celá sekce *Central Directory* a jsou v ní přepsány všechny ofsety ukazující na sekce *Local File Header* jednotlivých souborů obsažených v APK archivu. Sekce *Central Directory* začíná jednou ze sekcí *File Header*. Sekce *File Header* obsahuje v bytech 42 až 46 ofset příslušné sekce *Local File Header* [60], který je pomocí metody `Apk.get_local_header` nalezen, je k němu přičtena délka DEX souboru a voláním `update_local_header` je upravený ofset zapsán zpět na své místo. V cyklu se dále volá metoda

`apk.get_next_file_header`, ve které je nalezena následující sekce *File Header* v *Central Directory* (začíná značkou `PK 01 02` [60]) a celý cyklus je opakován.

Po skončení metody `Janus.update_offsets` je z funkce `main` volána metoda `Janus.join_the_files`. Na jejím počátku jsou připojením dat objektu `Apk` za data objektu `Dex` spojena data souboru DEX a archivu APK s upravenými ofsety. Následuje volání metody `update_data_length`. V hlavičce DEX je totiž v bytech 32–36 [37] uvedena délka souboru, kterou je potřeba přepsat z délky DEX na délku spojeného souboru. Po úpravě délky je volána metoda `update_checksum`. V této metodě se nejprve upraví SHA-1 hash celého souboru zapsaná v DEX hlavičce v bytech 12–32 [37]. Následně je upraven kontrolní součet Adler-32, který se nachází také v DEX hlavičce v bytech 8–12 [37].

Po skončení metody `Janus.join_the_files` jsou ve funkci `main` spojená a upravená data obou souborů zapsána do výstupního souboru, jehož jméno si uživatel volí pomocí třetího argumentu aplikace.

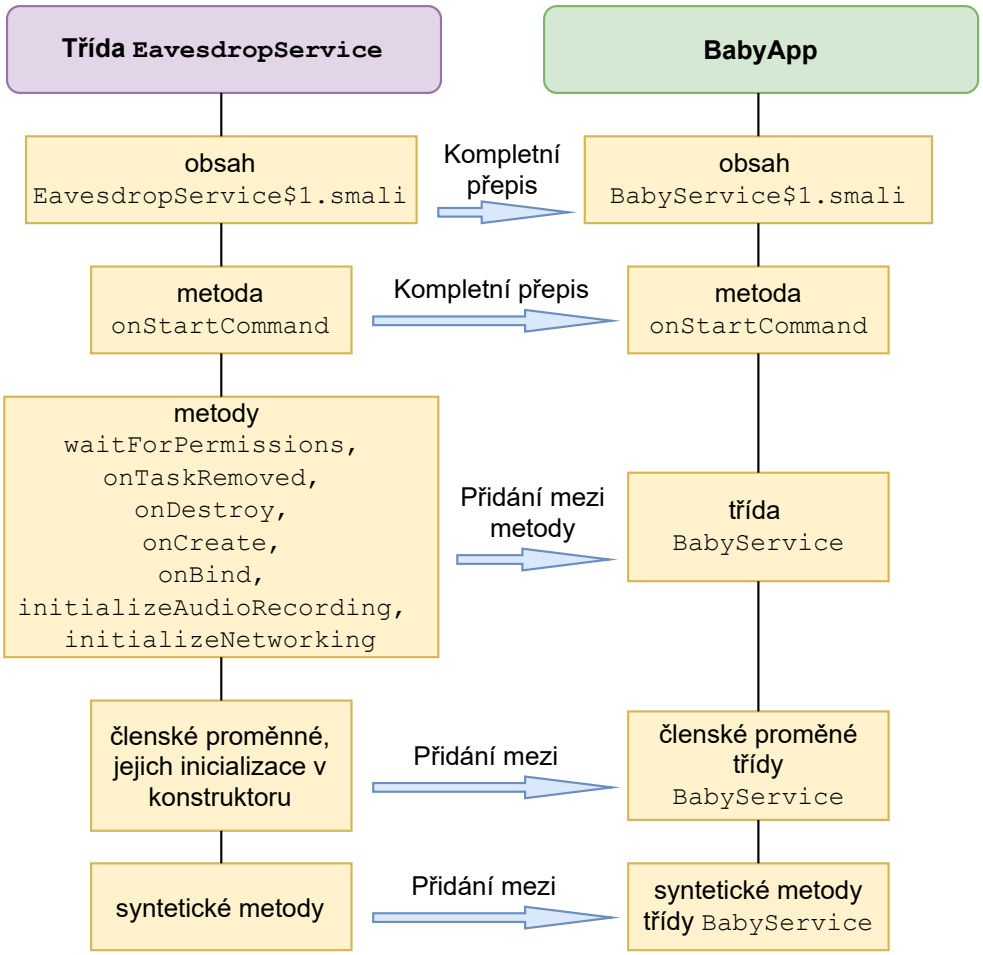
3.4 Tvorba škodlivého instalačního balíčku

V této kapitole již byla popsána implementace aplikace obsahující logiku odposlechu, implementace aplikace, která přijímá odposlechnutá data a implementace aplikace, která spojí APK archiv s DEX souborem a umožňuje tak vytvořit škodlivý instalační balíček využívající zranitelnost Janus. Jak je znázorněno na obrázku 2.3 na straně 40, pro vytvoření škodlivého instalačního balíčku je nyní potřeba přenést funkcionalitu aplikace pro odposlech do zvolené legitimní aplikace, z upravené legitimní aplikace extrahovat DEX soubor a spojit ho s jejím neupraveným APK. Celý proces tvorby škodlivého instalačního balíčku je popsán v této sekci. Veškeré úpravy, které je potřeba provést v *disassemblované* legitimní aplikaci, jsou zobrazeny na obrázku 3.7.

Postup nahrazení funkcionality legitimní aplikace je (za dále uvedených podmínek) univerzální a jde použít na různých aplikacích. V této sekci je pro popis procesu nahrazení zvolena aplikace *BabyApp* [61]. *BabyApp* je *open-source* aplikace, tedy aplikace s otevřeným zdrojovým kódem, která nahrává zvuk z mikrofonu a pokud hlasitost nahraného zvuku překročí určenou hranici (například se rozpláče spící dítě), aplikace začne nahraný zvuk odesílat na uživatelův server, kde je zvuk možné dále zpracovat.

V sekci 2.1 byly popsány podmínky, které by měla splňovat cílová legitimní aplikace – musí žádat o oprávnění `RECORD_AUDIO` a `INTERNET`, musí obsahovat alespoň jednu službu[18] a může obsahovat třídu fungující jako *broadcast receiver*, která reaguje na událost `BOOT_COMPLETED` a zajišťuje tak start

[18] V sekci 2.1 je služba uvedena jako nepovinná komponenta, která pouze vylepší fungování odposlechu, ale protože aplikace pro odposlech je jako služba již od začátku implementována (viz sekce 3.1), stala se v přítomnost služby v legitimní cílové aplikaci podmínkou.



Obrázek 3.7: Diagram znázorňující veškeré úpravy, které je potřeba provést v legitimní aplikaci *BabyApp*, aby do ní byla převedena funkcionalita služby `EavesdropService`.

odposlechu po spuštění systému. Aplikace *BabyApp* splňuje všechny výše uvedené podmínky. V jediné službě, kterou *BabyApp* obsahuje, a která bude nahrazena službou pro odposlech, navíc probíhá veškerá interakce s mikrofonem, takže kvůli výlučnému přístupu k němu není potřeba zasahovat do jiných míst v kódu legitimní aplikace. V době psaní této práce je aktuální verze *BabyApp* navíc podepsána pouze *APK Signature Scheme v1* (výpis 4.3 na straně 73), takže škodlivý instalační balíček lze nainstalovat i na nezazáplatované verze OS Android 7.0 až 8.0[19].

[19]Viz sekce 1.4.3 – pokud je APK balíček podepsán schématem verze 2 (popř. 3), je na verzích OS Android 7.0 a vyšší kontrolován pouze podpis pomocí schématu nejvyšší systémem podporované verze a zranitelnost Janus, která se týká *APK Signature Scheme v1*, na zmíněných verzích OS Android nelze využít.

Celý proces nahrazení funkcionality legitimní aplikace začíná *disassemblováním* instalačního balíčku APK legitimní aplikace a útočníkovy aplikace pro odposlech. Jak bylo zmíněno v sekci 2.2.2, z APK archivu nejprve musí být extrahovány všechny soubory a poté je *disassemblován* soubor `classes.dex`, který obsahuje zkompilovaný aplikační kód (*Dalvik* bytekód). Bytekód je při *disassemblování* převeden do jazyka *Smali*, což je textový popis jednotlivých *Dalvik* instrukcí.

Pro práci s APK archivy byl použit nástroj *APKtool* [62]. *APKtool* je nástroj pro reverzní inženýrství instalačních balíčků aplikací pro operační systém Android. Pomocí něj je možné APK balíček rozbalit, *disassemblovat* jeho kód, *assemblovat* ho a všechny rozbalené soubory zpět zabalit a vytvořit instalovatelný balíček (ten je pro instalaci ještě nutné podepsat). APK balíček aplikace *BabyApp* byl rozbalen a *disassemblován* pomocí příkazu

```
$ apktool d babyApp.apk
```

Nástroj *APKtool* při rozbalování APK aplikace *BabyApp* vytvořil adresář `babyApp`, jehož struktura je znázorněna na obrázku 3.8. Aplikace s logikou odposlechu (`eavesdropservice.apk`) byla *disassemblována* obdobně.

```
original ..... původní nezměněné soubory
  META-INF ..... soubory s metadaty
    CERT.RSA
    CERT.SF
    eclipse.inf
    MANIFEST.SF
  AndroidManifest.xml ..... manifest v binární podobě
res ..... soubory s rozvržením GUI
smali ..... disassemblovaný soubor classes.dex
  android
  com
    hollandhaptics
      babyapp ..... Java balíček aplikace BabyApp
        BabyService$1.smali
        BabyService$2.smali
        BabyService.smali
        MainActivity.smali
        ...
  org
unknown ..... ostatní soubory
AndroidManifest.xml ..... manifest v textové podobě
apktool.yml ..... konfigurace APKtool
```

Obrázek 3.8: Struktura adresáře, který byl vytvořen nástrojem *APKtool* po rozbalení a *disassemblování* aplikace *BabyApp*.

Podle záznamu v `AndroidManifest.xml` se třída, která implementuje službu využívanou aplikací *BabyApp*, nazývá `BabyService`. Jak je vidět na obrázku 3.8, mezi *disassemblovanými* soubory aplikace *BabyApp* se nachází soubory `BabyService.smali`, `BabyService$1.smali` a `BabyService$2.smali`. To, že skutečně jde o implementaci třídy, která dědí od třídy `Service`, lze jednoduše ověřit – nadtřída třídy `BabyService` je ve *Smali* kódu `BabyService` uvedena (viz výpis 3.1).

```
.super Landroid/app/Service;
```

Výpis 3.1: Direktiva `.super` jazyka *Smali*, která v tomto případě značí, že třída, v jejímž zdrojovém kódu se direktiva nachází, dědí od třídy `Service`.

Kód ve výpisu 3.1 znamená, že třída `BabyService` skutečně dědí od třídy `Service`. Adresář `smali` útočníkovy *disassemblované* aplikace pro odposlech (`eavesdropservice.apk`) obsahuje (mimo jiné) soubory `EavesdropService.smali` a `EavesdropService$1.smali`, které obsahují implementaci třídy `EavesdropService` (viz sekce 3.1), která obsahuje veškerou logiku pro provádění odposlechu.

I když je útočníkova funkcionalita pro odposlech implementována jako jediná pojmenovaná třída (služba `EavesdropService`), její *Smali* reprezentace je rozdělena do dvou souborů – `EavesdropService.smali` a `EavesdropService$1.smali`. To proto, že v rámci metody `onStartCommand` je v odposlechové službě vytvořen objekt vlákna, který je implementován jako tzv. *anonymní třída* dědící od třídy `Thread`. Anonymní třída je třída, která nemá jméno, a v celém kódu aplikace existuje pouze jedna její instance. Příklad implementace anonymní třídy je ve výpisu 3.2. *Smali* anonymní třídy označuje názvem tříd, ve kterých jsou vytvářeny, a za tento název přidá znak `$` a číslo. Útočníkova služba pro odposlech se tedy ve skutečnosti skládá ze dvou tříd a služba `BabyService` rovnou ze tří.

```
Thread streamThread = new Thread(new Runnable() {
            @Override
            public void run() { /*...*/ }
});
```

Výpis 3.2: Anonymní třída dědící od třídy `Thread` s překrytou metodou `run`. Objekt `streamThream` je jediný objekt této třídy v celém programu.

Cílem útočníka v této chvíli je nahradit funkcionalitu služby `BabyService`[20] funkcionalitou služby `EavesdropService`. Anonymní třída `EavesdropService$1` by navíc měla nahradit anonymní třídu

[20]Legitimní služba `BabyService` v tomto případě funguje velice podobně, jako útočníkova služba `EavesdropService`. Tato podobnost však není podmínkou – jakoukoliv službu v legitimní aplikaci je možné nahradit službou pro odposlech nehledě na to, jakou činnost legitimní služba vykonává.

BabyService$1 nebo BabyService$2. Zde byla konkrétně zvolena třída BabyService$1, protože plní stejnou úlohu jako objekt třídy EavesdropService$1 – tvoří hlavní vlákno služby (BabyService$2 je pomocné vlákno pro nahrávání souborů na server).

Po *disassemblování* instalačního balíčku APK útočníkovy aplikace pro odposlech a legitimní aplikace *BabyApp* byl tedy nejprve celý obsah souboru EavesdropService$1.smali překopírován do souboru BabyService$1.smali. V jazyce *Smali* jsou použita pro všechny reference na objekty plná jména, například objekt typu String v jazyce Java je typu Ljava/lang/String v jazyce *Smali*. V souboru BabyService$1.smali bylo nutné všechny reference, které odkazují na třídu com/example/eavesdropservice/EavesdropService a jsou uvedené u jejích členských proměnných a metod, přepsat na com/hollandhaptics/babyapp/BabyService. Reference na anonymní třídu EavesdropService$1 je nutné přepsat na reference na anonymní třídu BabyService$1. Příklad nahrazení referencí je ve výpisu 3.3.

```
# Před nahrazením
Lcom/example/eavesdropservice/EavesdropService;->
    onStartCommand(Landroid/content/Intent;II)I

# Po nahrazení
Lcom/hollandhaptics/babyapp/BabyService;->
    onStartCommand(Landroid/content/Intent;II)I
```

Výpis 3.3: Příklad nahrazení reference na metodu onStartCommand s plným jménem útočníkovy třídy pro odposlech (EavesdropService) referencí s plným jménem služby legitimní aplikace (BabyService) v souboru se *Smali* kódem anonymní třídy BabyService$1.smali.

Po přepsání anonymní třídy BabyService$1 anonymní třídou EavesdropService$1, která obsahuje hlavní funkcionalitu odposlechu, byla ve třídě BabyService přepsána metoda onStartCommand, která je volána po spuštění služby z kódu aplikace a ve které je vytvořen a spuštěn objekt přepsané anonymní třídy. Celá tato metoda byla ve třídě BabyService kompletně nahrazena metodou onStartCommand ze třídy EavesdropService – metoda onStartCommand ze třídy EavesdropService byla zkopírována (od jejího počátku, .method public onStartCommand(Landroid/content/Intent;II)I, až po direktivu .end method, která ji ukončuje) a její kód byl vložen místo kódu stejnojmenné metody v souboru BabyService.smali. Stejným způsobem jako metoda onStartCommand byly přepsány i metody onCreate (aplikace *BabyApp* by zbytečně inicializovala prostředky, které nevyužije), onDestroy (korektní uvolnění prostředků inicializovaných během fungování služby)

a `onBind` (aplikace *BabyApp* tuto metodu implementuje a bez jejího přepsání by bylo možné se ke službě připojit z kódu aplikace).

Následně byly do souboru `BabyService.smali` přidány metody, které odposlechová služba `EavesdropService` obsahuje navíc. Těmi jsou `onTaskRemoved`, ve které jsou uvolněny prostředky po ukončení aplikace, `waitForPermissions`, ve které služba čeká na přidělení potřebných oprávnění, a `initializeAudioRecording` s `initializeNetworking`, ve kterých jsou inicializovány objekty zajišťující přístup k mikrofonu a k síti.

Po tom, co byly přesunuty všechny výše uvedené metody ze souboru `EavesdropService.smali` do souboru `BabyService.smali`, byly ze služby pro odposlech do služby `BabyService` přesunuty všechny členské proměnné. Ty, které služba `BabyService` již obsahovala, byly v souboru ponechány, a proměnné útočníkovy služby `EavesdropService` byly přidány k nim. Po přesunu je nutné zkontrolovat, zda se název některé z nově přidaných proměnných neshoduje s již existující členskou proměnnou. V takovém případě je nutné jednu z proměnných ve všech souborech, kde je k ní přistupováno, přejmenovat (v případě aplikace *BabyApp* taková shoda nenastala). Členské proměnné jsou v jazyce *Smali* značeny direktivou `.field`, jako například ve výpisu 3.4.

```
.field private final port:I
.field private destination:Ljava/net/InetAddress;
```

Výpis 3.4: Označení členských proměnných v jazyce *Smali*. Zde konkrétně jde o finální privátní (`private final`) členskou proměnnou (`.field`) s názvem `port`, která je typu `int` (`:I`) a privátní proměnnou typu `InetAddress` s názvem `destination`.

Jak je vidět ve výpisu 3.4, členské proměnné jsou pomocí direktiv `.field` pouze deklarovány. Jejich počáteční hodnoty jsou jim přiřazeny až v konstruktoru třídy. Ukázka tohoto přiřazení v jazyce *Smali* je v výpisu 3.5.

Z konstruktoru třídy `EavesdropService` následně byla do konstruktoru třídy `BabyService` přesunuta všechna přiřazení hodnot všem členským proměnným, které do `BabyService` byly přidány dříve. Konstruktor třídy `BabyService` má pouze jeden lokální registr, kdežto konstruktor třídy `EavesdropService` využívá tři (viz výpis 3.5), takže tuto hodnotu bylo v `BabyService` nutné změnit. V konstruktoru `BabyService` bylo dále ponecháno volání konstruktoru třídy `Service` a přiřazení hodnot proměnných `TAG` a `serverResponeCode`. Za tato přiřazení byla přidána všechna přiřazení hodnot členským proměnným z konstruktoru `EavesdropService`.

Další součástí třídy `EavesdropService`, kterou bylo potřeba přesunout do třídy `BabyService`, byly tzv. *syntetické metody*. Syntetické metody jsou metody uměle vytvořené kompilátorem. Protože anonymní třída v jazyce Java může přistupovat k privátním členům třídy (proměnným i metodám), ve které byl vytvořen její objekt, Java kompilátor pro tyto privátní členy vytvoří umělé

```
31 .method public constructor <init>()V
32     .locals 3
       # ...
63     invoke-static {v0, v1, v2}, Landroid/media/
          AudioRecord;->getMinBufferSize(III)I
64     move-result v0
65     iput v0, p0, Lcom/example/eavesdropservice/
          EavesdropService;->minBufSize:I
       # ...
69     return-void
70 .end method
```

Výpis 3.5: Část konstruktoru `EavesdropService` v jazyce *Smali*. Konstruktor má 3 lokální registry (proměnné, `locals`). Na řádku 63 je volána metoda `AudioRecord.getMinBufferSize` s argumenty v registrech `v0`, `v1` a `v2`, jejíž výsledek je přes registr `v0` přiřazen členské proměnné `minBufSize` (registr `p0` při tomto přiřazení obsahuje referenci na objekt, jemuž `minBufSize` patří, tedy `this`).

```
127 .method static synthetic access$100(Lcom/example/
       eavesdropservice/EavesdropService;)I
128     .locals 1
129     .param p0, "x0"    # Lcom/example/
           eavesdropservice/EavesdropService;
130     .line 28
131     iget v0, p0, Lcom/example/eavesdropservice/
           EavesdropService;->minBufSize:I
132     return v0
133 .end method
```

Výpis 3.6: Syntetická metoda v jazyce *Smali* zpřístupňující privátní členskou proměnnou `EavesdropService.minBufSize` anonymní třídě `EavesdropService$1`. Hodnota proměnné je nahrána do registru `v0` a ten je poté použit jako návratová hodnota metody.

metody, pomocí kterých k nim anonymní třída může přistupovat [63]. Podoba syntetické metody v jazyce *Smali* je ukázána ve výpisu 3.6.

Názvy všech syntetických metod jsou ve tvaru `access$X`, kde `X` je unikátní číslo – identifikátor zvolený kompilátorem. Všechny metody s tímto názvem byly přesunuty z třídy `EavesdropService` do třídy `BabyService`. Protože se názvy některých syntetických metod obou tříd shodovaly, bylo nutné nově přidané syntetické metody přejmenovat. Protože se čísla identifikátorů syntetických metod ve třídě `BabyService` vždy liší alespoň o 2, pro

zamezení kolizím názvů stačí k identifikátorům shodně pojmenovaných syntetických metod třídy `EavesdropService` vždy přičíst číslo 1 (například z `access$100` se stane `access$101`). Dále bylo nutné v přepsaném souboru `BabyService$1.smali` najít všechna volání syntetických metod, které byly přidány do `BabyService.smali` a přejmenovány, a přepsat je, aby měly odpovídající jména.

Stejně jako po přepsání souboru `BabyService$1.smali` bylo nutné po všech výše uvedených náhradách a přesunech metod a členských proměnných z třídy `EavesdropService` do třídy `BabyService` přepsat v souboru `BabyService.smali` všechny reference vedoucí na `EavesdropService` na správné reference, tedy `BabyService`.

Po přesunu odposlechové funkcionality do legitimní aplikace byl navíc z hlavní aktivity `EavesdropService` přesunut ještě výpis hlášky, která uživatele zařízení s vlastní implementací úsporného módu žádá, aby úsporný mód pro tuto aplikaci zakázal. Výpis byl přesunut na úplný konec metody `onCreate` hlavní aktivity aplikace *BabyApp*. Přesunut byl celý kód zajišťující výpis hlášky – od volání metody `getApplicationContext` až po ukončení podmínky. Po přesunu bylo nutné jednak opět přepsat reference na objekty, ale také změnit název návěští `cond_0`, které je součástí přenášeného kódu, ale v metodě `onCreate` se již vyskytovalo. Jeho název byl jednoduše změněn na `cond_00` – jak v místě, kde se návěští nachází, tak i v místě skoku na něj.

Pro korektní běh služby odposlechu je vhodné, když z kódu aplikace není volána metoda `startService`, pokud je služba již spuštěna. V kódu aplikace *BabyApp* je před spuštěním `BabyService` kontrolováno, zda služba již byla spuštěna. Pokud by v kódu aplikace tato kontrola chyběla, jak je tomu například v aplikaci *Sound Recorder* [64], která je dostupná na internetovém obchodě s aplikacemi *F-Droid* [65] a která byla v rámci tohoto útoku využita jako alternativa k *BabyApp*, bylo by vhodné ji tam doplnit (podrobněji popsáno dále).

Protože aplikace *BabyApp* má zaregistrovaný tzv. *broadcast receiver* na událost typu `BOOT_COMPLETED`, obsahuje třídu `BootComplete`, která aplikaci v metodě `onReceive` umožňuje reagovat na událost spuštění operačního systému. Metoda `onReceive` třídy `BootComplete` obsahuje pouze kód, který spustí službu `BabyService`, což je pro útok ideální a metodu není potřeba nijak měnit. Pokud by ovšem tento kód neobsahovala, bylo by do ní vhodné přidat kód jako např. ve výpisu 3.7.

Po tom, co byla do *Smali* kódu aplikace *BabyApp* přenesena veškerá požadovaná funkcionalita služby `EavesdropService`, byl z *disassemblované* aplikace *BabyApp* znovu vytvořen balíček APK. K tomu byl, stejně jako k jejímu *disassemblování*, použit nástroj *APKtool*, konkrétně příkaz

```
$ apktool b babyApp
```

Tento příkaz vytvořil ze *Smali* kódu binární soubor s *Dalvik* bytekódem (`classes.dex`) a společně s ostatními soubory ho zabalil do APK archivu.

```
39 new-instance v0, Landroid/content/Intent;
40 const-class v1, Lcom/hollandhaptics/babyapp/
       BabyService;
41 invoke-direct {v0, p1, v1}, Landroid/content/
       Intent;-><init>(Landroid/content/
       Context;Ljava/lang/Class;)V
42 .local v0, "sIntent":Landroid/content/Intent;
43 invoke-virtual {p1, v0}, Landroid/content/Context;->
       startService(Landroid/
       content/Intent;)Landroid/content/ComponentName;
```

Výpis 3.7: *Smali* kód, který volá metodu `startService`. Nejprve je vytvořena instance třídy `Intent`, které je na řádku 41 předán název třídy `BabyService`. Na řádku 43 je volána metoda `startService`, která spustí službu `BabyService`. Pokud aplikace obsahuje *broadcast receiver* reagující na událost `BOOT_COMPLETED`, je vhodné tento kód (s upraveným názvem služby) vložit do jeho metody `onReceive`.

Pokud by útočník chtěl distribuovat přímo tento nový archiv, musel by ho podepsat (archiv se jinak nenainstaluje do zařízení). Výhodnější pro útočníka je využít zranitelnost Janus a ze znovuvytvořeného APK archivu použít pouze DEX soubor, který připojí před legitimní APK archiv. Při instalaci tohoto souboru do zařízení s operačním systémem Android se zkontroluje podpis (popř. struktura) legitimního APK, ale bude použit kód z nového DEX souboru.

Nový DEX soubor je možné buďto extrahovat z nově vytvořeného APK archivu, který se nachází v adresáři `babyApp/dist`, nebo ho lze najít v adresáři `babyApp/build/apk`. Ke spojení DEX souboru s legitimním (podepsaným) APK archivem byla použita aplikace `janus_exploit`, jejíž implementace je popsána v sekci 3.3. Konkrétní příkaz použitý při vytváření škodlivého balíčku je popsán v sekci 4.1. Soubor `exploited.apk`, který je výstupem aplikace `janus_exploit`, je finální podobou škodlivého instalačního balíčku, jehož vytvoření je cílem této sekce. Jde o spojení DEX souboru s APK archivem, které lze nainstalovat do zranitelného zařízení s operačním systémem Android.

Jak bylo řečeno na začátku této sekce, klíčové části výše popsaného postupu tvorby škodlivého instalačního balíčku jsou za určitých podmínek univerzální, což bylo předvedeno na další legitimní aplikaci. V rámci této práce byla kromě *BabyApp* využita i aplikace *Sound Recorder* [64], která funguje jako diktafon, pomocí kterého lze nahrát a poté i přehrát zvukový záznam. Aplikace obsahuje službu s názvem `RecorderService`, která po *disassemblování* obsahuje dvě anonymní třídy.

Přidání odposlechu do aplikace *Sound Recorder* proběhlo velice podobným postupem jako u aplikace *BabyApp*. Stejně jako u aplikace *BabyApp* byla nejprve přepsána jedna z anonymních tříd (`RecorderService$1`)

útočníkovou anonymní třídou `EavesdropService$1`. Následně byla z třídy `RecorderService` odstraněna proměnná `mPhoneStateListener`, protože přepsaná anonymní třída patřila jí. Dále byla (opět stejně jako při úpravách *BabyApp*) přepsána metoda `onStartCommand` v `RecorderService` stejnojmennou metodou z útočníkovy služby pro odposlech. Z útočníkovy služby byly dále do `RecorderService` přesunuty metody `onCreate`, `onDestroy`, inicializace sítě a objektů pro nahrávání zvuku, metoda čekající na získání oprávnění, všechny členské proměnné (včetně jejich inicializace v konstruktoru) a syntetické metody `access`.

Protože odposlech v aplikaci *Sound Recorder* je spuštěn po stisknutí tlačítka nahrávání, muselo být chování tlačítka upraveno, aby jeho opětovné stisknutí odposlech neukončilo. V hlavní třídě aplikace `SoundRecorder` byla v metodě `onClick` smazána všechna volání metod `startPlayback`, `pausePlayback` (záznam odposlechu se na rozdíl od záznamu diktafonu neukládá do souboru, takže by nebylo možné jej přehrát) a metod `stop` a `reset`. Ve třídě `Recorder`, která zařizuje spuštění a zastavení služby pro nahrávání, bylo v metodě `stopRecording` odstraněno volání metody `RecorderService.stopRecording`. Nakonec byla v metodách třídy `RecorderService` odstraněna veškerá volání metody `localStopRecording`. Metoda `isRecording` byla upravena tak, aby vracela vždy hodnotu `0`, což zamezí pokusům o pauzu nebo zpracovávání (neexistujícího) souboru se záznamem.

Jak bylo zmíněno výše, aplikace *BabyApp* před spuštěním služby již kontroluje, zda tato služba již nebyla spuštěna dříve. V aplikaci *Sound Recorder* takováto kontrola neprobíhá a je nutné ji do *Smali* kódu aplikace přidat. Do *disassemblované* třídy `Recorder` byla přidána metoda, která provede kontrolu, zda služba již byla spuštěna. Ukázka této metody je ve výpisu 3.8.

Ke všem voláním metody `startService` byla poté do *Smali* kódu aplikace doplněna podmínka, která na základě návratové hodnoty metody z výpisu 3.8 rozhodne, zda má být odposlechová služba spuštěna, či nikoliv. Ukázka podmínky, která zamezí opětovnému spuštění služby, je ve výpisu 3.9.

Po všech výše zmíněných úpravách *Smali* kódu aplikace *Sound Recorder* byl do její hlavní aktivity (`SoundRecorder`) přidán výpis s žádostí o zakázání úsporného módu na některých zařízeních. Následně byla tato aplikace (stejně jako výše zmíněná aplikace *BabyApp*) zpět sestavena nástrojem *APKtool* a pomocí nástroje `janus_exploit` byl upravený DEX soubor spojen s legitimním podepsaným APK archivem tak, aby vznikl instalační balíček, který využije zranitelnost Janus a nainstaluje škodlivý kód do zařízení.

```
27 private boolean isServiceRunning() {
28     ActivityManager manager = (ActivityManager) ctx.
          getSystemService(ACTIVITY_SERVICE);
29     for (ActivityManager.RunningServiceInfo service
          : manager.getRunningServices(Integer.
          MAX_VALUE)){
30         if(service.service.getClassName().equals(
              "com.example.app.myService")){
31             return true;
32         }
33     }
34     return false;
35 }
```

Výpis 3.8: Metoda (z důvodu přehlednosti v jazyce Java, přesunuta byla na úrovni jazyka *Smali*), která ověří, zda služba s názvem `myService` aplikace `app` je již spuštěna. Pokud je metoda volána ze třídy, která není *aktivitou*, je nutné jí pro zjištění spuštěných služeb předat tzv. *kontext* nějaké *aktivity* (zde členská proměnná `ctx`). V metodě je prohledán seznam služeb spuštěných v systému získaný od komponenty `ActivityManager`. Pokud je v seznamu nalezena hledaná služba `myService`, je vrácena hodnota `true`. V opačném případě je vrácena hodnota `false`.

```
973 invoke-direct {p0}, Lcom/example/app/myClass;->
       isServiceRunning()Z
974 move-result v7
975 if-nez v7, :cond_9
976 invoke-virtual {p0, v0}, Landroid/content/Context;->
       startService(Landroid/content/Intent;)Landroid/
       content/ComponentName;
977 :cond_9
```

Výpis 3.9: Příklad implementace podmínky, která povolí pouze jedno spuštění služby. Nejprve je volána metoda `isServiceRunning` z výpisu 3.8 a pokud vrátí hodnotu `0` (`false`), je volána metoda `startService`, která spustí službu (řádek 976). Pokud metoda `isServiceRunning` vrátí nenulovou hodnotu, je spuštění služby přeskočeno.

Kapitola 4

Testování

Útok implementovaný v rámci této závěrečné práce se skládá ze tří modulů – aplikace pro odposlech, serverové aplikace a aplikace spojující DEX a APK soubory. V předchozích kapitolách je popsán návrh a implementace těchto modulů. V této kapitole je popsáno, jak probíhaly testy jednotlivých komponent útoku. Testy jsou na závěr vyhodnoceny a jsou diskutována možná rozšíření nebo vylepšení útoku.

4.1 Tvorba škodlivého instalačního balíčku

Na začátku útoku je nutné vytvořit škodlivý instalační balíček, tedy spojení DEX souboru upravené legitimní aplikace (s přidanou funkcionalitou aplikace pro odposlech) a neupraveného (vývojářem podepsaného) APK archivu legitimní aplikace.

V sekci 3.3 je popsána implementace aplikace `janus_exploit`. Tato aplikace je implementovaná jako Python modul, který se spouští se třemi argumenty – DEX souborem, APK archivem a názvem výstupního souboru. Aplikaci `janus_exploit` byly jako argumenty předány DEX soubor upravené legitimní aplikace (viz sekce 3.4) a neupravený archiv legitimní aplikace a z nich byl vytvořen soubor `exploited.apk`. Tento soubor je spojeným souborem využívajícím zranitelnost Janus způsobem jako na obrázku 1.12 na straně 26. Průběh generování škodlivého instalačního balíčku aplikací `janus_exploit` je ve výpisu 4.1.

Jak bylo popsáno v sekci 1.5, zranitelnost Janus spočívá v obcházení kontroly podpisu instalačního balíčku. Podpis instalačního balíčku garantuje, že balíček nebude pozměněn a že pokud se u dvou různých balíčků shoduje certifikát podepisujícího, pochází tyto balíčky od stejného vývojáře. APK archiv se do zařízení s operačním systémem Android bez podpisu nenainstaluje.

Vývojář může svůj vytvořený APK archiv podepsat a podpis následně ověřit pomocí nástroje `apksigner` [66], který je součástí standardní vývojářské sady pro operační systém Android. Před OS Android verze 7.0,

```
$ python -m janus_exploit classes.dex babyApp.apk
    exploited.apk
Merging files classes.dex and babyApp.apk, result
    will be written to exploited.apk
Start of the Central Directory offset updated:
    1200606 ---> 3153406
Updated local header offsets
Updating data length to 3181966
Updating checksum
exploited.apk generated
```

Výpis 4.1: Spuštění aplikace `janus_exploit`, která za soubor `classes.dex` připojí instalační balíček `babyApp.apk`. Data spojených souborů jsou zapsána do souboru `exploited.apk`.

kdy existovalo pouze *APK Signature Scheme v1*, bylo možné APK archivy podepisovat i nástrojem `jarsigner` [67], což je nástroj pro podepisování JAR archivů jazyka Java. Protože APK a JAR mají velmi podobný formát (viz sekce 1.4.2) a *APK Signature Scheme v1* je založeno na podepisování JAR archivů, lze nástroj `jarsigner` použít pro tvorbu a ověření podpisu *APK Signature Scheme v1*.

Jak bylo řečeno v předchozí kapitole, v této práci byly využity dvě legitimní aplikace – *Sound Recorder* a *BabyApp*. Kontrola podpisu škodlivého balíčku vytvořeného aplikací `janus_exploit` z DEX souboru s upraveným kódem aplikace *BabyApp* a jejího legitimního APK proběhne bez problémů. Nástroj `jarsigner` podpis spojeného souboru ověří stejně, jako by ověřoval legitimní balíček této aplikace (viz výpis 4.2) a nástroj `apktool` též (viz výpis 4.3). To, že škodlivý balíček využívá zranitelnost Janus, ani jeden z nástrojů neodhalil.

```
- Signed by "CN=pavlov, OU=pavlov, O=pavlov,
   L=pavlov, ST=pavlov, C=pavlov"
  Digest algorithm: SHA1
  Signature algorithm: SHA1withRSA, 2048-bit key

jar verified.
```

Výpis 4.2: Část výstupu vypsaného při ověření podpisu škodlivého APK archivu založeného na aplikaci *BabyApp* nástrojem `jarsigner` (příkaz `jarsigner -verify -verbose exploited.apk`). Legitimní instalační balíček je podepsán přímo vývojářem (`pavlov`), protože byl stažen z jeho *GitHub* repozitáře. Díky zranitelnosti Janus tento podpis zůstal po spojení legitimního APK s upraveným DEX souborem neporušen.

```
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2):
   false
Verified using v3 scheme (APK Signature Scheme v3):
   false
Number of signers: 1
Signer #1 certificate DN: CN=pavlov, OU=pavlov,
   O=pavlov, L=pavlov, ST=pavlov, C=pavlov
```

Výpis 4.3: Část výstupu vypsaného při ověření podpisu škodlivého APK založeného na aplikaci *BabyApp* nástrojem `apksigner` (příkaz `apksigner verify -v --print-certs exploited.apk`). Podpis archivu je ověřen (`Verifies`), ale pouze pomocí podpisového schématu verze 1.

```
- Signed by "CN=FDroid, OU=FDroid, O=fdroid.org,
   L=ORG, ST=ORG, C=UK"
    Digest algorithm: SHA1
    Signature algorithm: MD5withRSA (weak),
       2048-bit key
WARNING: The jar will be treated as unsigned,
   because it is signed with a weak algorithm that
   is now disabled by the security property:
jdk.jar.disabledAlgorithms=MD2, MD5, RSA keySize <
   1024, DSA keySize < 1024
```

Výpis 4.4: Část výstupu ověření podpisu škodlivého instalačního balíčku založeného na aplikaci *Sound Recorder* nástrojem `jarsigner`. Instalační balíček je sice podepsán internetovým obchodem s aplikacemi *F-Droid*, ale `jarsigner` ho označil jako nepodepsaný, protože používá hashovací algoritmus MD5, který je u JAR archivů považován za zastaralý.

Aplikace *Sound Recorder* je již dlouhou dobu neaktualizovaná (naposledy v roce 2012). Ověření jejího podpisu nástrojem `jarsigner` je ve výpisu 4.4. Výstup ověření podpisu byl stejný jak pro škodlivý balíček, tak i pro samotný legitimní instalační balíček aplikace *Sound Recorder*. Jak je vidět ve výpisu 4.4, nástroj `jarsigner` v obou případech označil balíček jako nepodepsaný, protože z pohledu standardu jazyka Java používá zastaralý hashovací algoritmus. Standardy pro JAR archivy však nemusí být shodné s požadavky na podpis archivů APK, takže použití zastaralého algoritmu v podpisu legitimní aplikace nemusí být pro útočníka překážkou (jak dokazuje nástroj `apksigner`). Ověření podpisu škodlivého balíčku vytvořeného z aplikace *Sound Recorder* nástrojem `apksigner` je ve výpisu 4.5.

```
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2):
   false
Verified using v3 scheme (APK Signature Scheme v3):
   false
Number of signers: 1
Signer #1 certificate DN: CN=FDroid, OU=FDroid,
   O=fdroid.org, L=ORG, ST=ORG, C=UK
```

Výpis 4.5: Ověření podpisu škodlivého instalačního balíčku založeného na aplikaci *Sound Recorder* nástrojem `apksigner`. Podpis je ověřen (`Verifies`), ale pouze pomocí podpisového schématu verze 1.

Jak je zřejmé z výpisu 4.5, zastaralost hashovacího algoritmu není pro ověření oficiálním nástrojem operačního systému Android problematická. Protože je legitimní APK archiv podepsán pouze podpisovým schématem verze 1, nástroj `apksigner` při kontrole útočníkova balíčku `exploited.apk` využití zranitelnosti Janus též neodhalí (výstup pro škodlivý balíček je totožný jako výstup pro legitimní APK archiv).

4.2 Testy škodlivé aplikace

Po vytvoření škodlivých instalačních balíčků, jejichž podpis zůstal díky zranitelnosti Janus zachován, byly tyto balíčky otestovány na dvou reálných zranitelných zařízeních – nejprve na *Xiaomi Redmi 2* a poté na *Lenovo K5 Note* (viz obrázek 4.1). Při testech bylo pozorováno, jak proběhne instalace balíčku, zda škodlivá aplikace svým vzhledem připomíná legitimní aplikaci, jak jsou odesílána nahraná data z mikrofonu a také, zda odposlech pokračuje i po ukončení aplikace, případně po restartu zařízení.

Při těchto testech byla k příjmu a zpracování dat využita serverová aplikace, jejíž implementace je popsaná v sekci 3.2. Testování a funkce této aplikace jsou popsány v samostatné sekci 4.3.

4.2.1 Xiaomi Redmi 2

Prvním zařízením, na kterém byly škodlivé balíčky otestovány, je *Xiaomi Redmi 2*, model *HM 2LTE-CMCC*. Toto zařízení má operační systém MIUI 7.3, který je založen na OS Android 5.1.1. Poslední bezpečnostní záplatu obdrželo toto zařízení 1. dubna 2016, takže zranitelnost Janus obsahuje.

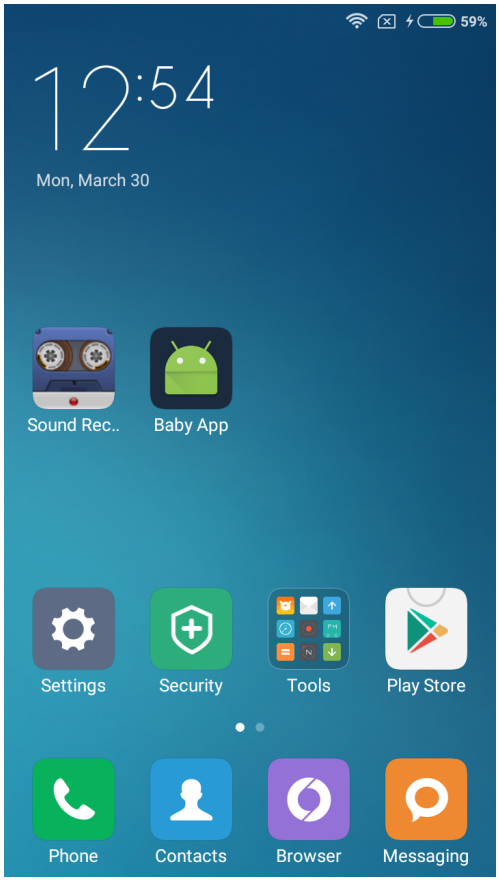
Instalace škodlivých balíčků vytvořených na základě aplikací *BabyApp* a *Sound Recorder* proběhla bez problému. Škodlivé balíčky se nainstalovaly a jejich aplikace bylo možné spustit. Spustila se upravená verze aplikace, takže

Obrázek 4.1: Zařízení, na kterých byly škodlivé instalační balíčky testovány. Vlevo je *Xiaomi Redmi 2*, vpravo *Lenovo K5 Note*.

využití zranitelnosti Janus zafungovalo a jako soubor s kódem se do zřízení nainstaloval DEX soubor, kterým začíná škodlivý instalační balíček.

Následně bylo otestováno, zda se škodlivý balíček nainstaluje jako aktualizace legitimní aplikace. Nejprve byla do zařízení nainstalována legitimní aplikace a poté byl nástrojem *Android Debug Bridge* [68], konkrétně příkazem `adb install -r exploited.apk`, do zařízení nainstalován škodlivý balíček `exploited.apk`. Přepínač `-r` značí, že nejde o novou instalaci, ale aktualizaci již nainstalované aplikace. I tato instalace proběhla v pořádku, *log* zařízení neobsahoval žádnou informaci, která by značila chybu nebo nekonzistenci škodlivého instalačního balíčku. Na obrázku 4.2 je snímek domovské obrazovky s ikonami obou škodlivých aplikací.

Jak bylo zmíněno v sekci 2.1.1, *Xiaomi* je jedním z výrobců zařízení s operačním systémem Android, kteří do systému implementovali svůj vlastní způsob úspory baterie, a služby na pozadí se tak na jejich zařízeních nechovají korektně podle oficiální dokumentace (jsou kvůli úspoře častěji ukončovány a nejsou obnovovány po ukončení jejich aplikací). Podle žebříčku sestaveného iniciativou *Don't kill my app!* [52] je *Xiaomi* v době psaní této práce čtvrtým

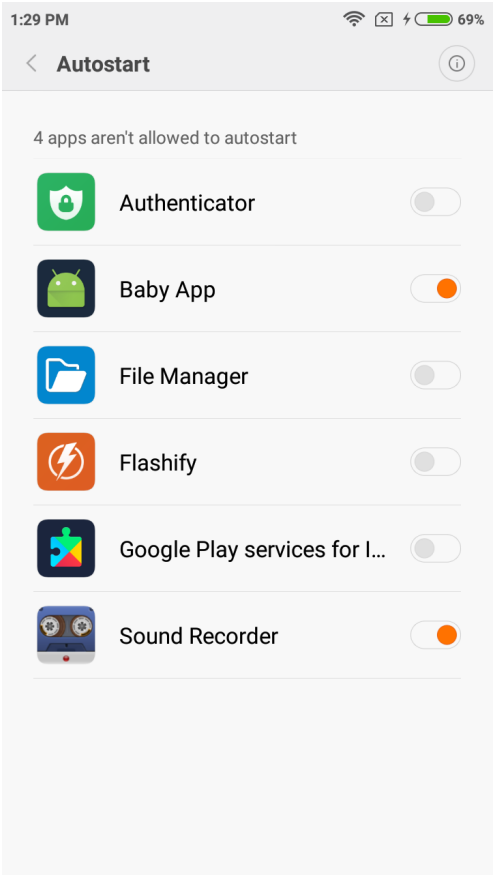


Obrázek 4.2: Snímek obrazovky z testování na *Xiaomi Redmi 2*. Domovská obrazovka s ikonami obou nainstalovaných škodlivých aplikací a také ikonou systémové aplikace *Security*.

nejhůře hodnoceným výrobcem zařízení s OS Android, což značí, že implementace služeb v zařízeních *Xiaomi* je pro vývojáře aplikací (a v důsledku toho i pro koncové uživatele) nepřívětivá.

Konkrétně na zařízení *Redmi 2* je služba spuštěna pouze do ukončení její aplikace (nehledě na návratovou hodnotu metody `onStartCommand`). Tomuto chování lze zamezit přidělením oprávnění na tzv. *autostart*, což aplikaci umožní obnovení služby po ukončení aplikace a získávání systémových *broadcast* zpráv s *intent* `BOOT_COMPLETED`, aby aplikace mohla reagovat na spuštění systému.

Oprávnění *autostart* nepatří mezi běžná oprávnění operačního systému Android, která jsou popsána v sekci 1.3.4, a žádost o něj nelze uvést do `AndroidManifest.xml`. Nelze ani žádat o jeho přidělení přímo z kódu aplikace. Na zařízení *Xiaomi Redmi 2* se toto oprávnění přiděluje v systémové aplikaci *Security* pod nabídkou *Permissions – Autostart*. Snímek obrazovky

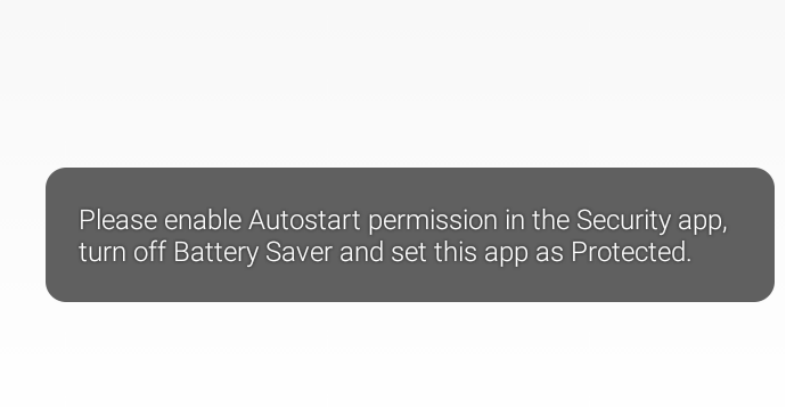


Obrázek 4.3: Snímek obrazovky z testování na *Xiaomi Redmi 2*. Přidělení speciálního oprávnění *autostart*, které službám umožní obnovení po ukončení jejich aplikací.

přidělování tohoto oprávnění je na obrázku 4.3. Jak bylo zmíněno v sekci 3.1, jediný možný způsob, jak toto oprávnění získat, je přimět uživatele, aby ho ručně přidělil. Proto byl do hlavní aktivity obou upravovaných legitimních aplikací přidán výpis hlášky, která o toto oprávnění uživatele požádá. Tato hláška se zobrazí po každém spuštění škodlivé aplikace na zařízení z žebříčku *Don't kill my app!* – viz obrázek 4.4.

Bez přiděleného oprávnění *autostart* byly odposlechové služby obou škodlivých aplikací skutečně ukončeny spolu s jejich aplikacemi. Následující testy byly tedy provedeny s přiděleným oprávněním *autostart*.

Po spuštění aplikace *BabyApp* byl na obrazovku pouze vypsán text *„Service State: Running“* (viz obrázek 4.5). Toto chování je správné a je totožné s chováním legitimní aplikace *BabyApp* – tato aplikace nemá grafické rozhraní, pouze spustí službu, která zaznamenává zvuky okolí a při překročení určeného prahu hlasitosti začne zvuky odesílat na server uživatele.



Obrázek 4.4: Snímek obrazovky z testování na *Xiaomi Redmi 2*. Po spuštění škodlivé aplikace je uživatel požádán, aby aplikaci přidělil oprávnění *autostart*.

Ihned po spuštění škodlivé aplikace *BabyApp* byla spuštěna upravená služba `BabyService`, do které byla vložena funkcionalita útočníkovy služby `EavesdropService` (viz sekce 3.4). Data z mikrofonu byla škodlivou aplikací odesílána na útočníkův server, kde bylo možné je přehrát. Přehrávaný zvuk byl srozumitelný, v kvalitě odpovídající zvoleným parametrům nahrávání.

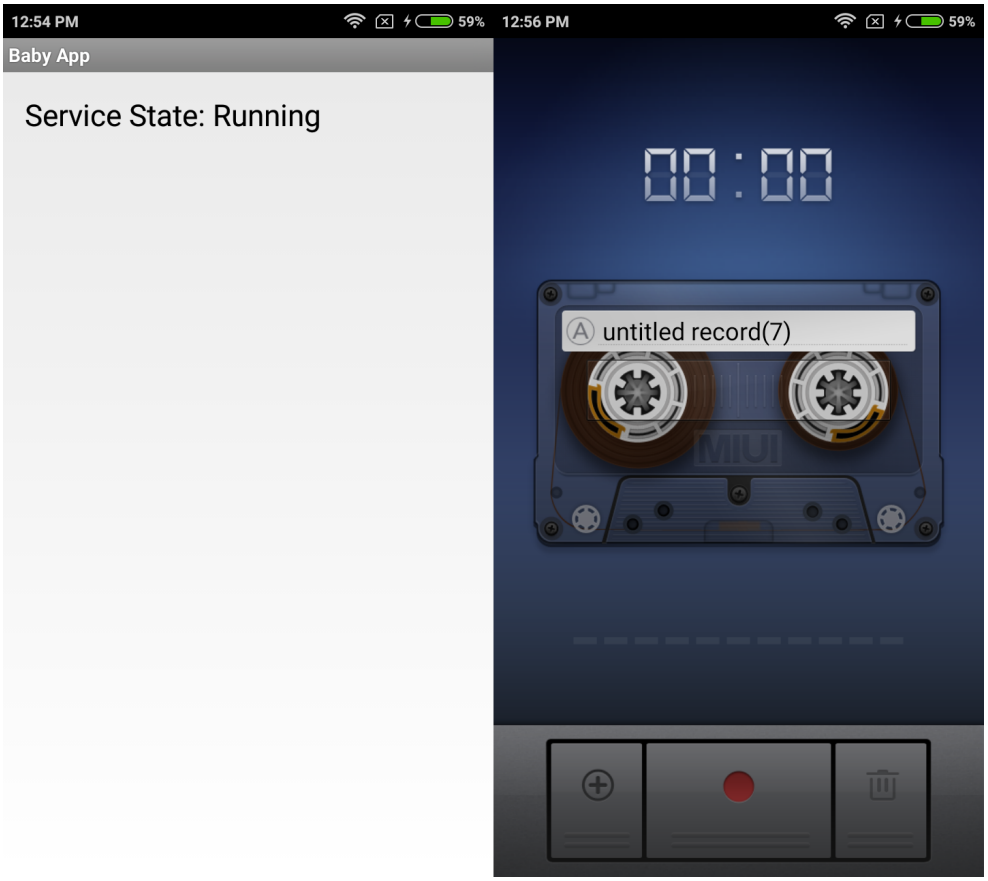
Po minimalizaci aplikace (stisknutí tlačítka *domů*) služba zůstala spuštěná a její činnost pokračovala. Po zamčení obrazovky též. Po ukončení aplikace (odstranění z přehledu spuštěných aplikací) byla služba ukončena, ale po zlomku sekundy byla systémem opět obnovena[21]. Bylo jí ale přiděleno nové číslo portu, což se v serverové aplikaci projevilo jako nové spojení.

Aplikace *BabyApp* má zaregistrovaný *broadcast receiver* a díky němu může reagovat na události typu `BOOT_COMPLETED`, tedy na spuštění systému. Škodlivá aplikace na něj reaguje spuštěním odposlechové služby. Po restartu zařízení byla odposlechová služba škodlivé aplikace založené na *BabyApp* automaticky spuštěna, opět s novým číslem portu.

Služba v operačním systému Android může sice být spuštěna na pozadí i po ukončení její aplikace, ale nelze ji v systému zachovat po odinstalování aplikace. Po odstranění škodlivé aplikace *BabyApp* ze systému byla odposlechová služba ukončena a již nebyla obnovena.

Škodlivá aplikace *Sound Recorder* fungovala velice podobně, jako škodlivá aplikace *BabyApp*. Po jejím spuštění se na obrazovce vykreslilo grafické rozhraní legitimní aplikace *Sound Recorder* (obrázek 4.5). Odposlechová služba `EavesdropService` v této aplikaci nahradila službu `RecorderService`,

[21] Pouze díky přidělenému oprávnění *autostart*. Pokud by ho aplikace neměla, odposlech by v této chvíli končil.



Obrázek 4.5: Snímky obrazovky z testování na *Xiaomi Redmi 2*. Vlevo je spuštěná aplikace *BabyApp*, vpravo je spuštěná aplikace *Sound Recorder*.

jejíž spuštění je navázáno na stisknutí tlačítka *„nahrávání“* – prostředního tlačítka s červeným kruhem. Po stisknutí tohoto tlačítka se služba úspěšně spustila a začala odesílat data získaná z mikrofonu na útočníkův server. Serverová aplikace dokázala získaná data přehrát v reproduktorech.

Stejně jako v případě *BabyApp*, odposlech prostřednictvím škodlivé aplikace *Sound Recorder* fungoval i po minimalizaci aplikace (stisknutí tlačítka *domů*) a zamčení obrazovky. S přiděleným oprávněním *autostart* byla odposlechová služba po ukončení aplikace ukončena též, ale po zlomku sekundy byla opět obnovena – také s nově přiděleným číslem portu. Po restartu zařízení odposlechová služba obnovena nebyla, protože aplikace *Sound Recorder* nemá zaregistrovaný žádný *broadcast receiver* a nemůže tak reagovat na spuštění systému ani na jiné systémové události. Po odinstalování škodlivé aplikace *Sound Recorder* byla stejně jako v případě škodlivé aplikace *BabyApp* odstraněna i spuštěná odposlechová služba.

4.2.2 Lenovo K5 Note

Druhým reálným zařízením, které bylo k dispozici pro testování škodlivých aplikací, bylo *Lenovo K5 Note* (model *A7020a40*). Toto zařízení obsahuje operační systém Android 6.0 a poslední bezpečnostní záplatu obdrželo 5. října 2016, takže zranitelnost Janus též obsahuje.

Instalace obou škodlivých balíčků proběhla bez problémů – systém nerozpoznal, že instalační balíčky škodlivých aplikací *BabyApp* a *Sound Recorder* jsou ve skutečnosti spojené soubory DEX a APK. Úspěšně proběhla i simulovaná aktualizace, kdy do zařízení byla nejprve nainstalována legitimní aplikace a poté byl nástrojem *Android Debug Bridge* [68] (příkazem `adb install -r exploited.apk`) nainstalován škodlivý balíček jako aktualizace již nainstalované legitimní aplikace. Na obrázku 4.6 je domovská obrazovka s ikonami obou nainstalovaných škodlivých aplikací.

Obrázek 4.6: Snímek obrazovky z testování na *Lenovo K5 Note*. Domovská obrazovka s ikonami (nahoře) obou škodlivých aplikací.

Po spuštění škodlivé aplikace *BabyApp* bylo zobrazeno stejné grafické rozhraní jako při testování na zařízení *Xiaomi Redmi 2* (viz obrázek 4.5). Odposlechová služba byla ihned automaticky spuštěna a začala získávat data z mikrofonu a odesílat je na útočníkův server. Serverová aplikace získaná data dokázala přehrát v reproduktorech, zvuk byl srozumitelný a jeho kvalita odpovídala zvoleným parametrům nahrávání.

Po minimalizaci aplikace *BabyApp* (stisknutí tlačítka *domů*) i po uzamčení obrazovky služba zůstala spuštěná a dále získávala data z mikrofonu a odesílala je na útočníkův server. Po ukončení aplikace *BabyApp* (odstranění aplikace z přehledu spuštěných aplikací, který se zobrazí po stisknutí tlačítka *přehled*), byla služba ukončena, ale po zlomku sekundy byla operačním systémem opět obnovena. Stejně jako na zařízení *Xiaomi* obdržela obnovená služba nové číslo portu. Na rozdíl od zařízení *Xiaomi* nebylo na zařízení *Lenovo* potřeba povolovat *autostart* ani jiné neobvyklé oprávnění či nastavení a odposlechová služba byla po ukončení aplikace obnovena přesně podle specifikace v oficiální dokumentaci OS Android.

Dále bylo otestováno obnovení odposlechové služby škodlivé aplikace založené na aplikaci *BabyApp* po restartu zařízení. Škodlivá aplikace *BabyApp* obsahuje třídu, která je v jejím souboru `AndroidManifest.xml` zaregistrována jako *broadcast receiver* na událost `BOOT_COMPLETED` a ze které je odposlechová služba spouštěna, což škodlivé aplikaci umožňuje spuštěním odposlechové služby reagovat na spuštění systému. Když byla odposlechová služba spuštěna, zařízení bylo restartováno a po spuštění systému byla odposlechová služba během několika sekund obnovena. Odposlechová služba byla ukončena až ve chvíli, kdy byla odinstalována škodlivá aplikace (stejně jako při testech na zařízení *Xiaomi*).

Po spuštění škodlivé aplikace založené na aplikaci *Sound Recorder* bylo, stejně jako při testování na zařízení *Xiaomi*, na obrazovce vykresleno grafické rozhraní legitimní aplikace *Sound Recorder*. Protože je útočníkovou službou pro odposlech nahrazena služba pro nahrávání, je spuštění odposlechu vázáno na stisknutí tlačítka *nahrávání* (tlačítko s červeným kruhem). Po jeho stisknutí se odposlechová služba skutečně spustila a začala odesílat data útočníkovu serveru. Data v serverové aplikaci bylo možné přehrát, kvalita zvuku byla stejná jako v případě *BabyApp*.

Po minimalizaci aplikace (stisknutí tlačítka *domů*) služba zůstala spuštěna a odposlech pokračoval, po zamčení obrazovky též. Když byla škodlivá aplikace ukončena, odposlechová služba byla ukončena s ní, ale po zlomku sekundy byla systémem opět obnovena. Stejně jako v ostatních případech bylo službě po obnovení přiděleno nové číslo portu. Po restartu zařízení odposlechová služba obnovena nebyla, protože aplikace *Sound Recorder* nemá zaregistrovaný žádný *broadcast receiver* a tím pádem nemůže reagovat na zprávy o události spuštění systému. Po opětovném spuštění škodlivé aplikace a aktivaci odposlechu byla škodlivá aplikace odinstalována. Stejně jako v předchozích případech byla v tu chvíli odposlechová služba ukončena a již nebyla obnovena.

4.2.3 Testy v emulátoru

Přestože podle zadání této práce má útok být demonstrován na reálném zařízení, byly provedeny i testy škodlivého instalačního balíčku v oficiálním emulátoru operačního systému Android, který je součástí vývojového prostředí *Android Studio*. Hlavním účelem těchto testů bylo zjištění, jestli se škodlivý balíček nainstaluje na různé verze OS Android, které by měly obsahovat zranitelnost Janus, a jak se na nich chová. Výsledky těchto testů jsou shrnuty v tabulce 4.1.

Verze	Instalace	Odesílání dat	Perzistence
5.0	OK	OK, ale špatná vzorkovací frekvence	OK
5.1	OK	OK, ale špatná vzorkovací frekvence	OK
6.0	OK	OK	OK
7.0	OK	OK	OK
7.1.1	Chyba	—	—
8.0	Chyba	—	—

Tabulka 4.1: Souhrn výsledků testů provedených na různých verzích OS Android v emulátoru *Android Studio*.

Nejstarší testovanou verzí operačního systému Android byla verze 5.0. Jak bylo zmíněno v sekci 1.5.3, OS Android 5.0 je první verze tohoto systému, ve které byl aplikační virtuální stroj *Dalvik Runtime* nahrazen virtuálním strojem *Android Runtime* a jelikož Janus je zranitelností *Android Runtime*, má smysl začít testovat právě od verze 5.0. Škodlivé instalační balíčky vytvořené z obou legitimních aplikací (*BabyApp* a *Sound Recorder*) se do emulovaného zařízení s OS Android 5.0 podařilo úspěšně nainstalovat. Data odposlechu se ze zařízení v obou případech odesílala, ale serverová aplikace přehrávala pouze nesrozumitelný šum. V *logu* emulovaného zařízení se po spuštění aplikace vypsala hláška, která je ve výpisu 4.6.

```
E/audio_hw_generic: Error opening input stream
   format 1, channel_mask 0010, sample_rate 16000
```

Výpis 4.6: Chybová hláška vypsaná operačním systémem Android 5.0 po spuštění škodlivé aplikace. Při inicializaci objektů pro nahrávání zvuku s daným formátem se vyskytla chyba.

Chyba ve výpisu 4.6 vypsaná systémem znamená, že použitý obraz systému v emulátoru nepodporuje zvukový formát, který je využíván jak škodlivou aplikací, tak serverem. V aplikaci *Android Emulator* totiž některé obrazy starších verzí OS Android podporují pouze jeden nebo několik málo formátů nahrávání (kombinací kanálu, vzorkovací frekvence a velikosti vzorku). Například u vzorkovací frekvence je většinou podporováno pouze 8 000 nebo 16 000 vzorků za

sekundu [69] [70]. Podle hlášky ve výpisu 4.6 se systém snaží vytvořit audio vstup s 16 000 vzorky za sekundu, ale škodlivá aplikace je nastavena na 44 100. I když audio vstup byl nakonec vytvořen (konkrétní parametry již v hlášce vypsány nebyly) a data se nahrávala a odesílala, serverová aplikace též očekává vzorkovací frekvenci 44 100 vzorků za sekundu a přijatá data tak při přehrávání tvoří pouze šum. Nastavenou vzorkovací frekvenci kvůli této chybě nemá smysl měnit, protože je zaručeno, že stávající frekvenci 44 100 vzorků za sekundu podporují všechna reálná zařízení s OS Android [58].

Kromě výše popsané chybné vzorkovací frekvence však obě škodlivé aplikace fungovaly přesně podle očekávání – odposlechové služby obou škodlivých aplikací byly po ukončení jejich aplikací obnoveny a služba aplikace *BabyApp* byla obnovena i po tom, co bylo emulované zařízení restartováno.

Jako další byla testována verze operačního systému Android 5.1. Její test dopadl totožně jako test verze 5.0. Instalace též proběhla v pořádku, data se odesílala, ale systém vypsal do *logu* stejnou chybu, jako ve výpisu 4.6. Služby obou škodlivých aplikací byly po ukončení jejich aplikací obnoveny a služba aplikace *BabyApp* byla obnovena i po restartu zařízení.

Test verzí OS Android 6.0 a 7.0 dopadl lépe než testy předchozích verzí. Oba škodlivé balíčky se do systému nainstalovaly a škodlivé aplikace po spuštění začaly odesílat data serveru. Protože systémové obrazy OS Android 6.0 a 7.0 podporují i vyšší vzorkovací frekvence, nenastala v tomto případě chyba jako při testech OS Android 5.0 a 5.1 – nahrávaný zvuk byl čistý a srozumitelný. Služby obou škodlivých aplikací byly po ukončení jejich aplikací obnoveny a služba aplikace *BabyApp* byla obnovena i po restartu zařízení.

Následně byly testovány verze operačního systému Android 7.1.1 a 8.0. Protože jejich systémové obrazy jsou aktualizované a mají bezpečnostní záplaty, které Janus opravují (Android 7.1.1 má poslední záplatu z 1. ledna 2018 a Android 8.0 má poslední záplatu z 5. dubna 2018), instalace škodlivých balíčků se nezdařila. Při instalaci byla systémem do *logu* vypsána hláška, která je ve výpisu 4.7. Podle hlášky selhalo zpracování instalačního balíčku `base.apk`, což je název, který dostane instalační balíček každé aplikace po uložení do příslušného adresáře v systému. Protože škodlivé instalační balíčky nezačínají ZIP sekcí *Local File Header*, systém je nepovažuje za platné (viz sekce 1.5.2 a výpis 1.4).

4.3 Test serverové aplikace

V této sekci je popsáno testování serverové aplikace, která slouží k příjmu a zpracování odposlechnutých dat ze zařízení s operačním systémem Android, na kterých je nainstalovaná škodlivá aplikace.

Serverová aplikace je aplikace pro příkazovou řádku. Po jejím spuštění se na konzoli vypsal uvítací text, který obsahoval informace o aplikaci a nápovědu k jejímu ovládání. Tento uvítací text je na obrázku 4.7.

```
2020-03-26 17:39:21.375 1763-1817/system_process W/
   zipro: Error opening archive /data/app/
   vmdl2083047757.tmp/base.apk: Iteration ended

2020-03-26 17:39:21.375 1763-1817/system_process D/
   asset: failed to open Zip archive '/data/app/
   vmdl2083047757.tmp/base.apk'

2020-03-26 17:39:21.376 1763-1817/system_process E/
   PackageInstaller: Commit of session 2083047757
   failed: Failed to parse /data/app/vmdl2083047757.
   tmp/base.apk: AndroidManifest.xml
```

Výpis 4.7: Chybová hláška vypsaná systémem do *logu* při pokusu nainstalovat škodlivé instalační balíčky na operační systém Android, ve kterém byla zranitelnost Janus již opravena (verze 7.1.1 a 8.0). Systém při zpracování škodlivého instalačního balíčku rozpoznal, že nejde o ZIP archiv – instalace souboru, který nezačíná hlavičkou sekce *Local File Header* je podle výpisu 1.4 na straně 30 odmítnuta, což se projeví hláškou, že soubor nelze otevřít, a následně i hláškou, že se nepodařilo přečíst soubor `AndroidManifest.xml`, který je v archivu obsažen.

Při spuštění bez zadání argumentů pro počet vláken, maximální počet spojení a velikost kruhového pole byly použity jejich výchozí hodnoty – 10 megabytů pro kruhové pole každého spojení, 2 vlákna zpracovávající přijatá data a neomezený počet spojení, jejichž data jsou zpracovávána. Pro test byla využita tři zařízení zároveň – výše testované *Lenovo K5 Note*, *Xiaomi Redmi 2* a emulátor s operačním systémem Android 7.0. Po spuštění serverové aplikace byla spuštěna i škodlivá aplikace na emulátoru. Serverová aplikace detekovala nové příchozí spojení (přijatá data od odesílatele, kterého ještě neměla ve svém seznamu) a vypsala o něm na obrazovku upozornění (viz výpis 4.8). Protože spojení bylo první, serverová aplikace z něj automaticky udělala spojení aktivní a začala ho přehrávat v reproduktorech.

```
[+] New connection: 127.0.0.1:56986
[*] Active connection: 127.0.0.1:56986
```

Výpis 4.8: Serverová aplikace hlásí nové spojení z IP adresy 127.0.0.1 a portu 56986. Protože nové spojení bylo zároveň prvním přijatým spojením, aplikace z něj automaticky udělala spojení aktivní (přehrávané v reproduktorech).

Následně byly spuštěny škodlivé aplikace na obou reálných zařízeních. Na zařízení *Xiaomi Redmi 2* byla spuštěna škodlivá aplikace založená na aplikaci

Obrázek 4.7: Snímek obrazovky z testování serverové aplikace. Uvítací text s nápovědou pro ovládání.

BabyApp a na zařízení *Lenovo K5 Note* byla spuštěna škodlivá aplikace postavená na aplikaci *Sound Recorder*. Serverová aplikace rozpoznala dvě nová spojení a přidala je do svého seznamu. Uživatel může v serverové aplikaci zadat příkaz `l`, který mu vypíše všechna spojení ze seznamu aplikace a zvýrazní spojení, které je aktivní (viz výpis 4.9).

```
---------------------------------------
Active connection: 192.168.0.102:49531
---------------------------------------
Other connections:
 - 192.168.0.103:40859
 - 127.0.0.1:54332
```

Výpis 4.9: Spojení aktuálně posílající data serverové aplikaci. Jejich seznam v této podobě je uživateli vypsán po zadání příkazu `l`.

Mezi spojeními v serverové aplikaci bylo možné přepínat pomocí příkazu `s SPOJ`, kde `SPOJ` je identifikátor spojení ve stejném formátu jako v seznamu ve výpisu 4.9, tedy `IP:port`. Příkaz `s` způsobil, že se spojení `SPOJ` stalo aktivním, tedy že začalo přehrávat v reproduktorech.

Když byla ukončena škodlivá aplikace na zařízení, které bylo v daný moment aktivní, služba škodlivé aplikace byla obnovena a bylo jí systémem přiděleno nové číslo portu. To se v serverové aplikaci projevilo jako nové spojení. Protože z původního spojení (kombinace IP adresy a portu) přestala přicházet nová data, bylo toto spojení serverovou aplikací po dvou sekundách automaticky odstraněno (viz výpis 4.10).

```
[*] Active connection: 127.0.0.1:54332
[+] New connection: 127.0.0.1:51363
[-] Connection 127.0.0.1:54332 removed.
```

Výpis 4.10: Chování serverové aplikace při obnovení odposlechové služby aktivního spojení. Spojení `127.0.0.1:54332` bylo aktivní, ale jeho služba byla obnovena a získala nové číslo portu – z pohledu serverové aplikace vzniklo nové spojení `127.0.0.1:51363`. Protože z původního spojení přestala přicházet data, bylo odstraněno.

Následně bylo otestováno chování aplikace s omezenými parametry. Aplikace byla nejprve spuštěna pouze s jedním zpracovávajícím vláknem a 1 MB paměti pro kruhové pole. Pro zpracování tří spojení, která aplikaci posílala data, jedno zpracovávající vlákno stačilo a aplikace fungovala plynule. Poté byla aplikace spuštěna znovu a bylo jí přidáno omezení na jedno akceptované spojení (argument `-c 1`). Aplikace skutečně zpracovávala data pouze od jednoho spojení. Zbylá spojení se neprojevila ani v seznamu (příkaz `l`). Po ručním odstranění tohoto spojení (příkaz `d a`) začala aplikace zpracovávat spojení z jiného zařízení (opět pouze jedno). Když byly škodlivé aplikace na dvou zařízeních ze tří ukončeny a zbylé aktivní spojení bylo z aplikace ručně odstraněno (opět příkaz `d a`), trvalo přesně jednu minutu, než aplikace začala příchozí data ze spojení opět zpracovávat.

Po výše popsaných základních testech chování serverové aplikace bylo otestováno nahrávání do souboru a práce s kruhovým polem, které slouží pro uložení nejaktuálnější části záznamu ještě předtím, než uživatel k nahrávání zadá příkaz.

Serverová aplikace byla pro účely těchto testů nejprve spuštěna s argumentem `-m 1`, který způsobil, že vnitřní kruhové pole mělo velikost 1 megabyte. Tato velikost při 44 100 dvoubytových vzorcích za sekundu odpovídá zhruba 11 sekundám záznamu.

Následně bylo příkazem `r 127.0.0.1:51363` spuštěno nahrávání spojení `127.0.0.1:51363` (emulátoru). Nahrávání bylo spuštěno 20 sekund od spuštění serverové aplikace a po dalších 20 sekundách bylo příkazem `c 127.0.0.1:51363` ukončeno. Během nahrávání byla přepínána aktivní spojení – v jednu chvíli bylo aktivní spojení to nahrávané, v jinou zase spojení z jiného zařízení.

Po ukončení nahrávání byl v aktuálním adresáři vytvořen soubor `127.0.0.1.51363_2020.04.01.14.49.42.wav`. Název souboru se skládal z identifikátoru spojení (dvojtečka byla nahrazena tečkou) a časové značky počátku nahrávání (zde konkrétně 1. 4. 2020, 14:49:42). Soubor bylo možné přehrát v přehrávači multimédií. Jeho délka byla 31 sekund – 11 sekund bylo zaznamenáno do kruhového pole a 20 sekund uběhlo mezi zadáním příkazů pro počátek a konec nahrávání. Nahrávka i přes přepínání aktivních spojení pocházela pouze z emulátoru.

4.4 Testy s antiviry

Fungování škodlivé aplikace v zařízení s operačním systémem Android se nijak výrazně neliší od činnosti běžných aplikací – škodlivá aplikace pracuje s mikrofonem a síťovými komponentami, ale veškerá její činnost probíhá pouze v rámci jejích získaných oprávnění. Škodlivost této aplikace tkví převážně v tom, že provádí něco jiného, než od ní uživatel očekává. Proto byl proveden menší test, ve kterém bylo sledováno, zda škodlivou aplikaci v zařízení odhalí antivirové aplikace.

Test proběhl na zařízení *Xiaomi Redmi 2*. Instalační balíček škodlivé aplikace založené na aplikaci *BabyApp* byl uložen do úložiště zařízení, konkrétně do adresáře *Download*, kam se ukládají soubory stažené z internetu, a následně byl antivirovou aplikací prozkoumán. Antivirové aplikace by při prozkoumání instalačního balíčku teoreticky měly mít největší šanci na odhalení škodlivé aplikace, protože instalační balíček má typický formát pro využití zranitelnosti Janus – na začátku je hlavička DEX souboru, na konci je ZIP sekce *End of Central Directory*.

Po kontrole instalačního balíčku v úložišti byla škodlivá aplikace nainstalována a po její instalaci byla v antivirové aplikaci spuštěna kontrola aplikací. Poté byla škodlivá aplikace spuštěna a znovu zkontrolována antivirem.

K testování byla použita aplikace *Google Play Protect* a potom celkem deset dalších antivirových aplikací. Všechny byly testovány v jejich bezplatné verzi, případně zkušební plné verzi, pokud tuto možnost nabízely. Antivirové aplikace byly vybrány z žebříčku doporučovaných antivirových aplikací pro rok 2020 magazínem *Techradar Pro* [71].

Aplikace *Google Play Protect*, která je předinstalovaná v každém zařízení s OS Android, zablokovala instalaci škodlivé aplikace. Vypsala hlášku, která říkala, že škodlivá aplikace je podvrh a že se snaží převzít kontrolu nad zařízením nebo ukrást uživatelova data.

Jako první z antivirových aplikací třetích stran byla otestována zkušební plná verze antiviru *Bitdefender Mobile Security & Antivirus* [72]. Tato aplikace nehlásila žádné problémy – při kontrole instalačního balíčku, jeho instalaci ani při spuštění aplikace. Žádné problémy na zařízení nenašly ani antivirové aplikace *McAfee Mobile Security: VPN Proxy and Anti Theft Safe Wi-Fi* [73], *Kaspersky Mobile Antivirus: AppLock & Web Security* [74], *Avira Antivirus 2020 - Virus Cleaner & VPN* [75], zkušební plná verze *ESET Mobile Security & Antivirus* [76] a *Malwarebytes Security: Virus Cleaner, Anti-Malware* [77].

Jako další byly otestovány aplikace *Avast Antivirus – Mobile Security & Virus Cleaner* [78] a *AVG AntiVirus 2020 for Android Security Free* [79]. Na instalačním balíčku škodlivé aplikace, její instalaci ani jejím spuštění obě antivirové aplikace žádné problémy neobjevily. Upozornily však na to, že na zařízení jsou dvě bezpečnostní rizika – je spuštěno ladění přes USB (nástroj *Android Debug Bridge*) a je povolena instalace aplikací z neznámých (nedůvěryhodných) zdrojů.

Dále byla otestována antivirová aplikace *Antivirus & Virus Cleaner, Applock, Clean, Booster* od *TAPI Security Labs* [80]. Tato aplikace při kontrole škodlivého instalačního balíčku žádný problém neobjevila, ale po jeho instalaci nahlásila varování, že útočníkova aplikace představuje bezpečnostní riziko, protože používá nebezpečná oprávnění – konkrétně vypsala hlášku *„RISK – Application uses dangerous permission“*. V hlášení však nebylo uvedeno, o jaká konkrétní oprávnění se jedná. Šlo také pouze o varování a škodlivá aplikace nebyla označena jako *malware*.

Nejúspěšnější během tohoto testu byla antivirová aplikace *Mobile Security & Antivirus* od společnosti *Trend Micro* [81]. Škodlivá aplikace byla antivirem objevena již při prohlídce úložiště zařízení. Antivirová aplikace po kontrole instalačního balíčku škodlivé aplikace vypsala uživateli hlášku *„Removal Critical“* – je nezbytně nutné soubor odstranit. Po instalaci škodlivé aplikace do zařízení a následné kontrole nainstalovaných aplikací byla útočníkova aplikace opět odhalena. Antivirus tentokrát vypsal hlášku *„Uninstallation Critical – This app may threaten your security or privacy“*, která říká, že je nezbytně nutné tuto aplikaci odinstalovat, protože může narušovat uživatelovu bezpečnost nebo soukromí. Ani u jedné z vypsaných hlášek však nebyl uveden důvod – tedy zda je aplikace označena za škodlivou, protože využívá Janus, nebo z nějakého jiného důvodu, např. proto, že má zaregistrovaný

broadcast receiver na událost start systému. Legitimní aplikaci však antivirus za škodlivou neoznačil, takže u útočníkovy aplikace bylo využití zranitelnosti Janus pravděpodobně detekováno.

Na závěr testů s antiviry byl škodlivý instalační balíček nahrán na server *VirusTotal* [82], který nahraný soubor nechal zkontrolovat větším množstvím antivirů. Soubor byl jako škodlivý označen pouze dvěma antiviry z šedesáti. Antivirus *DrWeb* označil soubor jako `Android.Janus.1` – odhalil, že jde o spojení DEX a APK. Druhá úspěšná detekce byla od antiviru *Sophos AV*, který soubor označil jako `Exp/201713156-A`. Číslo tohoto označení je CVE identifikátor zranitelnosti Janus (CVE-2017-13156), takže antivirus *Sophos AV* spojený soubor též odhalil. Souhrn výsledků testů je v tabulce 4.2.

Antivirus	Úložiště	Instalace	Spuštění
Google Play Protect	X	✓	X
Bitdefender	X	X	X
McAffee	X	X	X
Kaspersky	X	X	X
Avira	X	X	X
ESET	X	X	X
Malwarebytes	X	X	X
Avast	X	obecné upozornění	X
AVG	X	obecné upozornění	X
TAPI	X	varování, ale ne *malware*	X
Trend Micro	✓	✓	X

Server *VirusTotal*: 2/60 (*DrWeb, Sophos AV*)

Tabulka 4.2: Souhrn detekcí škodlivé aplikace při testech provedených s antivirovými aplikacemi a testu na serveru *VirusTotal*. Testována byla detekce škodlivého souboru v úložišti zařízení, detekce instalace a spuštění škodlivé aplikace. ✓– detekováno, X – nedetekováno.

4.5 Vyhodnocení testů, možnosti dalšího vývoje

Aplikace, které byly v rámci této práce implementované, měly společně vytvořit komponenty pro útok, díky kterému je možné vzdáleně odposlouchávat oběť a tím narušit její soukromí. Útok se skládá ze tří aplikací – aplikace pro operační systém Android obsahující logiku odposlechu, aplikace přijímající odposlechnutá data na útočníkově serveru a aplikace, která podle principů zranitelnosti Janus vytvoří škodlivý instalační balíček obsahující funkcionalitu aplikace pro odposlech, jehož podpis ale patří legitimní aplikaci.

4.5.1 Aplikace pro spojení DEX a APK

Testy aplikace `janus_exploit` pro spojení souboru DEX s archivem APK skončily s kladným výsledkem. Aplikace z DEX souboru upravené legitimní aplikace s přidanou funkcionalitou odposlechu a neupraveného APK archivu té samé legitimní aplikace vytvořila spojený soubor, který využíval zranitelnost Janus. Tento škodlivý instalační balíček prošel kontrolami podpisu pomocí nástrojů `jarsigner` a `apksigner`. Na všechna zranitelná zařízení, která byla k dispozici (*Xiaomi Redmi 2*, *Lenovo K5 Note* a emulátory) se škodlivý balíček podařilo nainstalovat, jako by šlo o balíček legitimní.

Aplikace `janus_exploit` je přímočará a pokud by si měla zachovat stávající funkcionalitu (spojení souborů DEX a APK), k jejímu dalšímu vývoji nebo rozšiřování by nebyl důvod. Prostor pro rozšíření aplikace ale existuje, a to například v přidávání využití dalších zranitelností týkajících se instalačních balíčků APK – například zranitelnosti *Masterkey* zmíněné v sekci 1.5 nebo jakýchkoliv dalších, které teprve budou objeveny.

4.5.2 Serverová aplikace

Testy serverové aplikace též potvrdily, že aplikace funguje korektně. Aplikace přijímala data ze škodlivých aplikací, mezi jednotlivými spojeními dokázala přepínat, přehrávat je v reproduktorech a nahrávat je do souboru. V aplikaci fungovalo i kruhové pole, které slouží pro záznam spojení ještě předtím, než uživatel zadá příkaz k nahrávání.

Tato aplikace má větší potenciál pro další vývoj. Jde o aplikaci pro příkazovou řádku, takže by pro ni v rámci jejího rozšíření bylo možné vytvořit grafické rozhraní. Dále by aplikaci bylo možné rozšířit například umožněním uživateli více pracovat s aktuálními spojeními, aby si uživatel jednotlivá spojení mohl pojmenovat nebo třeba třídit do skupin. Aplikace momentálně podporuje pouze nahrávání do souboru ve formátu WAV, takže by mohla být rozšířena o podporu nahrávek v jiném zvukovém formátu – například MP3 z důvodu úspory datového provozu a místa na úložišti serveru.

4.5.3 Škodlivá aplikace pro OS Android

Testy škodlivé aplikace pro operační systém Android, tedy aplikace vzniklé přidáním funkcionality pro odposlech do legitimní aplikace, prokázaly úspěšné využití zranitelnosti Janus, funkční implementaci odposlechu a korektní přesun funkcionality odposlechu do legitimní aplikace na úrovni jazyka *Smali* – aplikaci bylo možné nainstalovat na každý pro testování dostupný zranitelný systém (ať už reálný či emulovaný), po spuštění měla aplikace stejné grafické rozhraní jako aplikace legitimní a odposlechová služba v zařízení zůstala spuštěná i po minimalizaci aplikace, uzamčení obrazovky, ukončení aplikace (s výjimkou *Xiaomi*, pokud aplikaci nebylo přiděleno oprávnění *autostart*) a v případě škodlivé aplikace *BabyApp* byla odposlechová služba automaticky

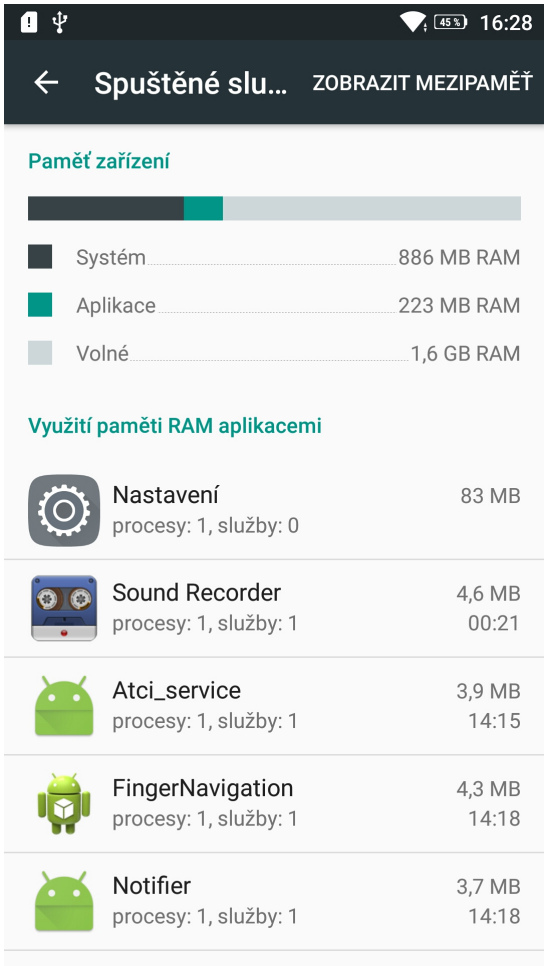
obnovena i po restartu zařízení. Jediným problémem bylo zmíněné oprávnění *autostart*, které ale podle iniciativy *Don't kill my app!* na zařízeních *Xiaomi* tvoří překážku pro všechny vývojáře aplikací pro OS Android.

Testy s antiviry ukázaly, že přestože je škodlivý instalační balíček typickým spojením DEX a APK pro využití zranitelnosti Janus, byl balíček jako škodlivý označen pouze malým počtem antivirových aplikací. Služba pro odposlech je v systému skryta podobně, jako *spyware* zvaný *xHelper* (viz sekce 1.1) – služba není vypsána v přehledu spuštěných aplikací a nezobrazuje ani žádné upozornění v horní liště. Přítomnost spuštěné služby lze v systému zjistit pouze ze seznamu spuštěných procesů, který je dostupný jen pokud je v zařízení povolen vývojářský režim (*Nastavení – Pro vývojáře – Spuštěné služby*). Snímek obrazovky se seznamem spuštěných služeb je na obrázku 4.8.

Škodlivá aplikace poskytuje mnoho prostoru pro rozšíření a další vývoj. Hlavní nevýhodou zvukového odposlechu na operačním systému Android verze starší než 10 (tedy ve všech zranitelných verzích) je to, že v systému není zaručena možnost přístupu k mikrofonu z více aplikací zároveň. Když je spuštěna služba pro odposlech, na některých zařízeních obsadí mikrofon a ostatní aplikace k němu nemohou přistupovat, což znemožní celou řadu aktivit, které uživatel na svém zařízení může chtít provádět. Uživatel tak nemůže využívat například telefon, videochat, kameru, diktafon nebo asistenta pro ovládání hlasem. Všechny tyto a ještě další aplikace využívající mikrofon uživateli nahlásí problém s obsazením mikrofonu, což může vést k odhalení odposlechu a odinstalování škodlivé aplikace.

Při pokusu o pořízení videonahrávky na zařízení *Xiaomi Redmi 2* ve chvíli, kdy byla spuštěna odposlechová služba, byla vypsána chybová hláška, která je na obrázku 4.9. Na zařízení *Lenovo* však ostatní aplikace využívající mikrofon fungovaly současně s odposlechovou službou bez problému. Videokamera i předinstalovaný legitimní diktafon na zařízení *Lenovo* fungovaly spolu s odposlechovou službou perfektně – odposlechová služba odesílala zvuková data na server a obě legitimní aplikace zvuk zaznamenávaly též.

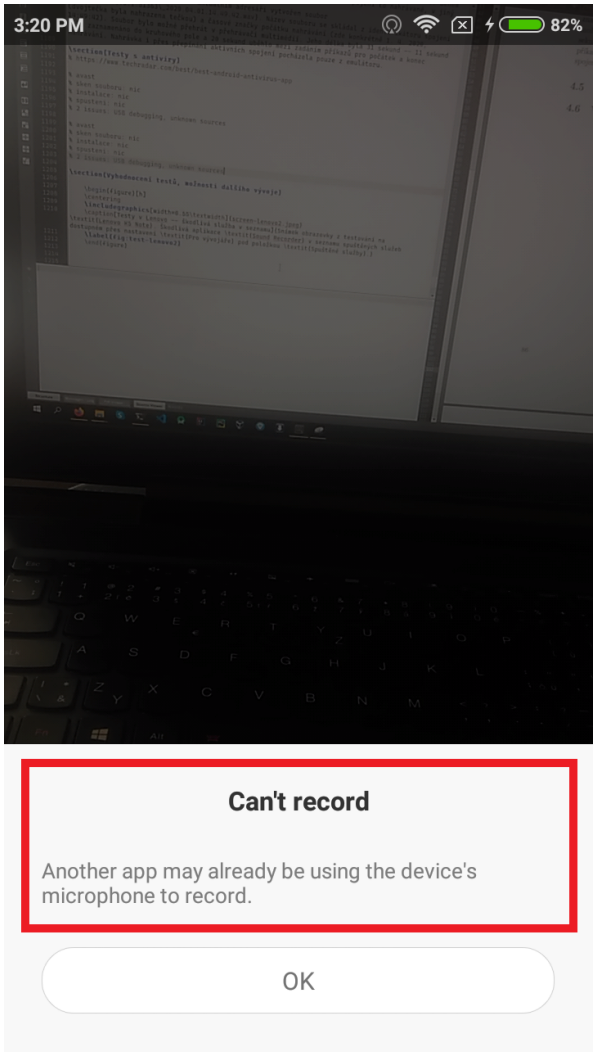
Některé aplikace (například diktafony) se problém s obsazením mikrofonu snaží vyřešit získáním oprávnění `READ_PHONE_STATE` [50], což je oprávnění, které aplikaci umožní získat informace o stavu telefonu – telefonní číslo, připojení k síti, ale i aktuální příchozí nebo odchozí hovory. S tímto oprávněním by služba pro odposlech mohla při příchozím nebo odchozím hovoru uvolnit mikrofon a telefonování by tak na zařízení blokováno nebylo. Na druhou stranu by šlo o další oprávnění potřebné pro úspěch útoku a zúžil by se tak seznam potenciálních kandidátů na využitelnou legitimní aplikaci. Oprávnění `READ_PHONE_STATE` navíc neřeší celý problém obsazení mikrofonu – služba pro odposlech by s ním mohla reagovat pouze na požadavky telefonu, ale kamera, hlasový asistent a ostatní aplikace využívající mikrofon by stále byly zablokovány. Řešení tohoto problému záleží na útočníkových preferencích – například pokud chce útočník odposlechem zaznamenat nějaké konkrétní události, odposlech by mohl probíhat pouze v určitých časových intervalech.



Obrázek 4.8: Snímek obrazovky z testování na *Lenovo K5 Note*. Služba škodlivé aplikace postavené na aplikaci *Sound Recorder* vypsaná v seznamu spuštěných služeb dostupném přes nastavení *Pro vývojáře* pod položkou *Spuštěné služby*.

Další možností, jak vyřešit problém s blokováním mikrofonu na některých zařízeních, je rozšířit škodlivou službu o další špionážní techniky. Pokud by legitimní aplikace měla příslušná oprávnění, mohla by škodlivá služba útočníkovi odesílat i fotografie, videonahrávky, seznam navštívených webových stránek nebo třeba obsah SMS či jiných zpráv. Pokud by škodlivá služba udržovala spojení s útočníkovým serverem, mohl by útočník zadat příkaz ke zvukovému odposlechu ručně pouze ve chvílích, kdy sám uzná za vhodné (např. v závislosti na GPS poloze nebo získaných SMS zprávách).

Aplikace s logikou odposlechu, jejíž funkcionalita je přenesena do legitimních aplikací, byla implementována s ohledem na vyšší míru univerzálnosti



Obrázek 4.9: Pokus o pořízení videonahrávky na *Xiaomi Redmi 2*, když je spuštěna služba pro odposlech. Aplikace kamery hlásí, že nahrávku nelze vytvořit, protože jiná aplikace již používá mikrofon.

– útočník tak může jednoduše stejným způsobem využít velké množství různých legitimních aplikací a dosáhnout tak vyššího počtu napadených zařízení. V důsledku toho ale škodlivé aplikace postrádají funkcionalitu aplikací legitimních. Škodlivé aplikace po spuštění sice budí zdání, že jde o aplikace legitimní, ale skutečná funkcionalita legitimních aplikací je nahrazena odposlechem. Zda, případně za jak dlouho, si uživatel všimne, že aplikace nefunguje podle jeho očekávání, závisí na povaze legitimní aplikace a konkrétním způsobu jejího zamýšleného používání. Například u škodlivé aplikace založené na diktafonu si její nefunkčnosti uživatel všimne ve chvíli, kdy si bude chtít

přehrát vytvořené záznamy. V případě škodlivé aplikace založené na aplikaci *BabyApp*, která má hlídat, zda uživatelovo dítě pláče ze spaní, si uživatel ztráty legitimní funkcionality dlouhou dobu všimnout nemusí. Legitimní aplikace totiž hlídá hlasitost zvuků v místnosti a projevuje se (odesílá data na uživatelův server) pouze v případě překročení daného prahu hlasitosti. Pokud tedy je uživatelovo dítě v klidu, legitimní aplikace data neodesílá.

Pokud se škodlivá aplikace nechová podle uživatelových očekávání, může být odinstalována a tím pádem skončí odposlech. Další možností pro rozšíření odposlechu je tedy jeho integrace do legitimní aplikace tak, aby škodlivá aplikace kromě odposlechu prováděla i činnosti, kvůli kterým si ji uživatel nainstaloval. U tohoto způsobu tvorby škodlivé aplikace by nebylo možné mít jednu třídu (službu), jejíž *Smali* kód by se přesouval do *Smali* kódu legitimních aplikací – u každé legitimní aplikace zvlášť by bylo nutné upravit její specifický způsob záznamu zvuku.

Protože odinstalování škodlivé aplikace znamená konec odposlechu, bylo by možné škodlivou aplikaci rozšířit o implementaci některého ze způsobů ochrany proti odinstalování, které byly zmíněny v sekci 1.1 – ať už falešný dialog odinstalace nebo kontrolu svých komponent v úložišti. Aplikaci, jejíž služba již byla spuštěna, by též bylo možné rozšířit tak, aby uměla skrýt svoji ikonu v nabídce aplikací. Pokud bude objeven způsob skrývání škodlivého softwaru *xHelper* (viz sekce 1.1), mohl by posloužit jako inspirace pro rozšíření škodlivé aplikace.

Ve škodlivé aplikaci je zapsána IP adresa útočníkova serveru, což by mohlo vést k jeho vyzrazení. Tento problém by bylo možné vyřešit například tak, že by škodlivé aplikace odesílaly odposlouchávaná data na proxy server, ze kterého by data byla dále odesílána na server útočníka. Komunikace by byla šifrovaná a skutečnou IP adresu útočníkova serveru by tak nebylo možné zjistit.

Potenciál pro další vývoj je také ve využívání dalších legitimních aplikací, aby byla škodlivou aplikací zasažena co největší skupina uživatelů. Útočník musí legitimní aplikaci pečlivě volit, ať už kvůli potřebným oprávněním nebo způsobu, jakým legitimní aplikace nahrává zvuk. Legitimní aplikace *BabyApp* a *SoundRecorder* použité v této práci byly zvoleny proto, že splňují požadavky na vyžadovaná oprávnění, a také proto, že na internetu jsou dostupné jejich instalační balíčky podepsané pouze *APK Signature Scheme v1* buď samotnými vývojáři, nebo internetovým obchodem, na kterém je balíček k dispozici. Dalším důvodem použití právě těchto aplikací byla jejich volná licence – obě aplikace mají otevřený zdrojový kód a permisivní licenci, takže k reverznímu inženýrství instalačního balíčku nebylo nutné shánět povolení autora aplikace. Pokud by však tento útok měl být skutečně nasazen a proveden na reálných uživatelích, útočník by na licence pravděpodobně nehleděl a mohl by si zvolit libovolnou aplikaci, která by nejlépe splňovala jeho požadavky na oprávnění a formu nahrávání. Tím by se útočníkovi značně rozšířil seznam legitimních aplikací, které může využít.

Závěr

Cílem této práce bylo analyzovat principy fungování a využití zranitelnosti Janus a také současný stav využití této zranitelnosti – známé útoky a jejich dopad na současná zařízení. Hlavním úkolem bylo na základě této analýzy navrhnout, implementovat a otestovat útok, který zranitelnost Janus využívá – konkrétně sadu aplikací, které útočníkovi umožní vzdáleně získávat data ze zvukového vstupu zařízení oběti.

Útok se skládá ze tří modulů. Prvním modulem útoku je aplikace `janus_exploit`, která spojí DEX soubor a APK archiv do jednoho souboru, který může být interpretován jako platný DEX i platné APK zároveň a využívá tak zranitelnost Janus – při instalaci tohoto souboru je úspěšně kontrolován podpis APK, ale do zařízení se uloží kód z DEX části spojeného souboru. Druhým modulem je aplikace pro OS Android zvaná `EavesdropService` obsahující funkcionalitu odposlechu, která byla přidána do legitimních aplikací. Třetím modulem je serverová aplikace, která přijímá data ze všech napadených zařízení a dokáže je přehrát v reproduktorech nebo uložit do souboru.

Hlavní cíl práce, tedy implementaci útoku cíleného na vzdálený odposlech oběti, se podařilo splnit. Nejprve byla navržena a implementována aplikace `EavesdropService` obsahující logiku odposlechu, která se v rámci útoku přesune do aplikace legitimní. Ta byla navržena tak, aby tento přesun bylo možné provést univerzálním způsobem u různých legitimních aplikací. Dále byla navržena a implementována aplikace `janus_exploit`, která na vstupu přečte DEX soubor a APK archiv, ve struktuře těchto souborů přepíše potřebné informace a data těchto souborů spojí do škodlivého instalačního balíčku, který využívá zranitelnost Janus. Následně byla navržena a implementována aplikace, která na serveru útočníka přijímá a zpracovává data z napadených zařízení.

Po seznámení se s principy zranitelnosti Janus byl vytvořen seznam požadavků, které musí splňovat legitimní aplikace, na které má být založen škodlivý instalační balíček. Na základě těchto požadavků byly zvoleny dvě

legitimní aplikace, které byly pro útok využity. Bytekód těchto aplikací byl převeden do *Smali* kódu, který byl upraven – byla do něj přidána funkcionalita aplikace `EavesdropService` obsahující logiku odposlechu. *Smali* kód byl následně převeden zpět na bytekód a soubor s tímto bytekódem (DEX) byl za pomoci aplikace `janus_exploit` spojen s neupraveným instalačním archivem (APK) legitimní aplikace.

Takto vytvořené škodlivé instalační balíčky využívaly zranitelnost Janus – úspěšně prošly kontrolou podpisu, jako by se jednalo o neupravené instalační balíčky legitimních aplikací, ale místo legitimního bytekódu se do zařízení nainstalovaly soubory s bytekódem s přidanou funkcionalitou odposlechu.

Všechny tři vytvořené moduly útoku byly otestovány. Test aplikace `janus_exploit` spojující soubor s bytekódem a instalační balíček proběhl úspěšně. Škodlivé instalační balíčky vytvářené touto aplikací neměly z hlediska *APK Signature Scheme v1* narušený podpis a při jejich instalaci se na zařízení nainstaloval DEX soubor s upraveným bytekódem.

Test serverové aplikace dopadl též úspěšně. Aplikace přijímala odposlechnutá data ze škodlivých aplikací a dokázala je přehrávat v reproduktorech. Mezi jednotlivými spojeními z různých zařízení bylo možné přepínat a nahrávat je do souboru ve formátu WAV. Při nahrávání v aplikaci korektně fungovalo i kruhové pole, které slouží pro uchování nejaktuálnější části záznamu ještě předtím, než uživatel spustí nahrávání daného spojení.

Samotné škodlivé aplikace pro OS Android byly testovány na dvou reálných zařízeních – *Xiaomi Redmi 2* a *Lenovo K5 Note*. Škodlivé aplikace na první pohled budily zdání, že jde o aplikace legitimní, na kterých byly založeny. Služba s odposlechem se spustila na pozadí a odesílala na server data z mikrofonu. Na zařízení *Lenovo* se služba chovala přesně podle očekávání – po ukončení škodlivé aplikace a po restartu zařízení byla služba obnovena a odposlech pokračoval. Na zařízení *Xiaomi* se ale služby na pozadí chovají jinak – pro docílení tohoto korektního chování na zařízení *Xiaomi* muselo být škodlivým aplikacím uživatele ručně přiděleno oprávnění *autostart*, kterým *Xiaomi* řeší úsporu baterie a které je podle iniciativy vývojářů aplikací pro OS Android zvané *Don't kill my app!* překážkou pro všechny vývojáře aplikací pro OS Android.

Následně bylo otestováno, jak jsou škodlivé aplikace vyhodnoceny různými antivirovými aplikacemi. Jako jediné z testovaných antivirových aplikací uspěly *Google Play Protect* a antivirová aplikace společnosti *TrendMicro*. Na serveru *VirusTotal* byly škodlivé aplikace detekovány pouze dvěma antiviry z šedesáti – aplikacemi *DrWeb* a *Sophos AV*.

Všechny tři moduly útoku je možné dále rozvíjet. Do aplikace `janus_exploit` je možné přidat podporu dalších zranitelností, které se týkají APK archivů. Serverové aplikaci je možné přidat například grafické rozhraní, podporu jiných zvukových formátů nebo větší míru uživatelské interakce se seznamem aktuálních spojení (například jejich třídění do skupin nebo pojmenovávání).

Prostor pro rozšíření má i aplikace `EavesdropService`, jejíž funkcionalita se při tvorbě škodlivých aplikací přenáší do *disassemblovaných* legitimních aplikací. Kromě zvukového odposlechu je do aplikace možné (za předpokladu, že legitimní aplikace získá příslušná oprávnění) přidat i další formy narušení soukromí a serverové aplikaci odesílat například i fotografie, videa, GPS polohu nebo obsah SMS zpráv.

Aplikace `EavesdropService` je vytvořena tak, aby funkcionalitu odposlechu bylo možné přesunout do mnoha různých legitimních aplikací univerzálním způsobem, což ale způsobí přepsání legitimní funkcionality odposlechem a tím pádem i narušení původní činnosti legitimní aplikace. Uživatel toto narušení nemusí ihned zpozorovat a odposlech na jeho zařízení může zůstat spuštěn dlouhou dobu. V rámci dalšího rozvoje útoku je však možné škodlivé aplikace upravit tak, aby kromě odposlouchávání fungovaly stejně jako legitimní aplikace a nemohly tak v uživateli vzbudit podezření, že má ve svém zařízení škodlivý software.

Protože je ve škodlivých aplikacích zapsaná IP adresa útočníka, mohl by útočník být snadno vypátrán. Řešením tohoto problému může být například šifrovaný tunel přes proxy server – škodlivé aplikace by šifrovaly pakety s odposlouchávanými daty a zapouzdřovaly je do paketů, které by odesílaly na proxy server. Proxy server by šifrované pakety odesílal na server útočníka.

Jak bylo řečeno výše, odposlechová služba `EavesdropService` je vytvořena tak, aby mohla být univerzálním způsobem přenesena do různých legitimních aplikací. Rozsah útoku by bylo možné rozšířit využitím dalších aplikací – čím více legitimních aplikací je využito, tím širší skupinu obětí útočník získá.

Nejúčinnější obranou proti tomuto útoku a obecně proti instalaci souborů využívajících zranitelnost Janus je oficiální bezpečnostní záplata systému, která byla vydána 1. prosince 2017. Na zařízeních, která záplatu neobdržela, je nutné mít jinou ochranu – ať už některou z antivirových aplikací třetích stran, jejichž test popsaný ve čtvrté kapitole ukázal, že škodlivý soubor detekují, nebo oficiální antivirus pro OS Android zvaný *Google Play Protect*, který se při testování tohoto útoku též osvědčil.

Literatura

[1] US-CERT: Spyware. Technická zpráva, 2008, [online]. Dostupné z: https://www.us-cert.gov/sites/default/files/publications/spywarehome_0905.pdf

[2] MICHALEVSKY, Y.; BONEH, D.; NAKIBLY, G.: Gyrophone: Recognizing Speech from Gyroscope Signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, srpen 2014, ISBN 978-1-931971-15-7, s. 1053–1067, [online]. Dostupné z: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/michalevsky

[3] GUTZMAN, C.; SWEEP, S.; TAMBO, A.: Differences and Similarities of Spyware and Adware. 600 East Fourth Street, Morris, MN 56267: University of Minnesota, 2005, doi:10.1.1.608.109, [online]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.608.109&rep=rep1&type=pdf

[4] CHATTERJEE, R.; DOERFLER, P.; ORGAD, H.; aj.: The Spyware Used in Intimate Partner Violence. In *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, ISSN 2375-1207, s. 441–458, doi:10.1109/SP.2018.00061, [online]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8418618

[5] AYCOCK, J.: *Spyware and Adware*, kapitola Getting There: Installation. Boston, MA: Springer Science+Business Media, 2011, ISBN 978-0-387-77740-5, s. 9–20, doi:10.1007/978-0-387-77741-2.

[6] AYCOCK, J.: *Spyware and Adware*, kapitola Getting There: Startup. Boston, MA: Springer Science+Business Media, 2011, ISBN 978-0-387-77740-5, s. 20–27, doi:10.1007/978-0-387-77741-2.

[7] AYCOCK, J.: *Spyware and Adware*, kapitola Staying There: Avoiding Detection. Boston, MA: Springer Science+Business Media, 2011, ISBN 978-0-387-77740-5, s. 29–37, doi:10.1007/978-0-387-77741-2.

[8] AYCOCK, J.: *Spyware and Adware*, kapitola Staying There: Avoiding Uninstall. Boston, MA: Springer Science+Business Media, 2011, ISBN 978-0-387-77740-5, s. 37–39, doi:10.1007/978-0-387-77741-2.

[9] COLLIER, N.: Android Trojan xHelper uses persistent re-infection tactics: here's how to remove. *Malwarebytes LABS*, únor 2020, [online], [cit. 2020-03-19]. Dostupné z: https://blog.malwarebytes.com/android/2020/02/new-variant-of-android-trojan-xhelper-reinfects-with-help-from-google-play/

[10] COLLIER, N.: Mobile Menace Monday: Android Trojan raises xHelper. *Malwarebytes Labs*, 2019, [obrázek], [cit. 2020-04-14]. Dostupné z: https://blog.malwarebytes.com/android/2019/08/mobile-menace-monday-android-trojan-raises-xhelper/

[11] HOLST, A.: Mobile operating systems' market share worldwide from January 2012 to December 2019. *Statista*, leden 2020, [online], [cit. 2020-02-06]. Dostupné z: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[12] DRAKE, J. J.; FORA, P. O.; LANIER, Z.; aj.: *Android™ Hacker's Handbook*, kapitola Looking at the Ecosystem. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2014, ISBN 978-1-118-60864-7, s. 1–2.

[13] ANDROID DEVELOPERS: Platform Architecture. *Android Developers*, prosinec 2019, [online], [cit. 2020-02-06]. Dostupné z: https://developer.android.com/guide/platform

[14] VITAS, M.: ART vs Dalvik: Introducing the New Android Runtime in KitKat. *Infimum*, prosinec 2013, [online], [cit. 2020-02-06]. Dostupné z: https://infinum.com/the-capsized-eight/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat

[15] CHELL, D.; ERASMUS, T.; COLLEY, S.; aj.: *Mobile Application Hacker's Handbook*, kapitola Analyzing Android Applications: Observing the Structure of a Package. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2015, ISBN 978-1-118-95850-6, s. 182–183.

[16] APVRILLE, A.: An Android Package is no Longer a ZIP. *Fortinet*, srpen 2018, [online], [cit. 2020-02-08]. Dostupné z: https://www.fortinet.com/blog/threat-research/an-android-package-is-no-longer-a-zip.html

[17] ALBERTINI, A.: ZIP[101] an archive walkthrough. *Corkami.com*, prosinec 2013, [online], [cit. 2020-02-07]. Dostupné z: https://github.com/corkami/pics/blob/master/binary/zip101/zip101.pdf

[18] CHELL, D.; ERASMUS, T.; COLLEY, S.; aj.: *Mobile Application Hacker's Handbook*, kapitola Analyzing Android Applications: Looking Under the Hood. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2015, ISBN 978-1-118-95850-6, s. 201–204.

[19] CHELL, D.; ERASMUS, T.; COLLEY, S.; aj.: *Mobile Application Hacker's Handbook*, kapitola Analyzing Android Applications: Running an Application. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2015, ISBN 978-1-118-95850-6, str. 204.

[20] CHELL, D.; ERASMUS, T.; COLLEY, S.; aj.: *Mobile Application Hacker's Handbook*, kapitola Analyzing Android Applications: Understanding Permissions. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2015, ISBN 978-1-118-95850-6, s. 212–219.

[21] DRAKE, J. J.; FORA, P. O.; LANIER, Z.; aj.: *Android^{TM} Hacker's Handbook*, kapitola Android Security Design and Architecture: Android Permissions. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2014, ISBN 978-1-118-60864-7, s. 1–2.

[22] ANDROID DEVELOPERS: Permissions overview. *Android Developers*, prosinec 2019, [online], [cit. 2020-02-13]. Dostupné z: https://developer.android.com/guide/topics/permissions/

[23] CYBERSECURITY AND INFRASTRUCTURE SECURITY AGENCY (CISA): Security Tips (ST04-018): Understanding Digital Signatures. *National Cyber Awareness System*, listopad 2019, [online], [cit. 2020-02-09]. Dostupné z: https://www.us-cert.gov/ncas/tips/ST04-018

[24] CZAGAN, D.: Non-Repudiation and Digital Signature. *Infosec Resources*, září 2019, [online], [cit. 2020-02-09]. Dostupné z: https://resources.infosecinstitute.com/non-repudiation-digital-signature

[25] CHELL, D.; ERASMUS, T.; COLLEY, S.; aj.: *Mobile Application Hacker's Handbook*, kapitola Analyzing Android Applications: Code Signing. Hacker's Handbook Series, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2015, ISBN 978-1-118-95850-6, s. 206–210.

[26] WANG, H.; LIU, H.; XIAO, X.; aj.: Characterizing Android App Signing Issues. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, 11.–15. listopadu 2019*, IEEE, 2019, ISBN 978-1-7281-2508-4, s. 280–292, doi:10.1109/ASE.2019.00035, [online]. Dostupné z: https://doi.org/10.1109/ASE.2019.00035

[27] GUARDSQUARE: Android APK signature scheme v3: context and new opportunities. *Guardsquare Blog*, říjen 2018, [online], [cit. 2020-02-08]. Dostupné z: https://www.guardsquare.com/en/blog/android-apk-signature-scheme-v3-context-and-new-opportunities

[28] ORACLE: Digests and the Signature File. *Oracle Java Documentation*. [zdrojový kód], 2019, [cit. 2020-02-08]. Dostupné z: https://docs.oracle.com/javase/tutorial/deployment/jar/intro.html

[29] ANDROID OPEN SOURCE PROJECT: APK Signature Scheme v2. *Android Source*, leden 2020, [online], [cit. 2020-02-08]. Dostupné z: https://source.android.com/security/apksigning/v2

[30] KIM, S.: AIR GO and APK Signing. *LINE Engineering*, leden 2019, [online], [cit. 2020-02-08]. Dostupné z: https://engineering.linecorp.com/en/blog/air-go-apk-signing/

[31] COUCHBASE: Certificate Rotation. *Couchbase NoEQUAL*, 2019, [online], [cit. 2020-02-09]. Dostupné z: https://docs.couchbase.com/server/current/manage/manage-security/rotate-server-certificates.html

[32] ANDROID OPEN SOURCE PROJECT: APK Signature Scheme v3. *Android Source*, leden 2020, [online], [cit. 2020-02-09]. Dostupné z: https://source.android.com/security/apksigning/v3

[33] WASSON, D. L.: Janus. *Ancient^{TM} History Encyclopedia*, únor 2015, [online], [cit. 2020-02-18]. Dostupné z: https://www.ancient.eu/Janus/

[34] GUARDSQUARE: New Android vulnerability allows attackers to modify apps without affecting their signatures. *Guardsquare Blog*, listopad 2017, [online], [cit. 2020-02-18]. Dostupné z: https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-attackers-modify-apps-without-affecting-their-signatures

[35] MITRE: CVE-2017-13156. *Common Vulnerabilities and Exposures*, srpen 2017, [online], [cit. 2020-02-18]. Dostupné z: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13156

[36] OSMOS: Janus The God of Transitions. *Osmos*, 2020, [obrázek], [cit. 2020-02-18]. Dostupné z: https://i2.wp.com/osmosnetwork.com/wp-content/uploads/2016/01/janus.jpg?resize=1170%2C752

[37] LYNCH, J.: The DEX File Format. *Bugsnag Blog*, leden 2018, [online], [cit. 2020-02-18]. Dostupné z: https://www.bugsnag.com/blog/dex-and-d8

[38] SEHR, D.; CHARTIER, M.; MARKO, V.; aj.: Funkce `DexFile::Open`, `dex_file.cc`, *Google Git*. [zdrojový kód], říjen 2017, [cit. 2020-02-19]. Dostupné z: https://android.googlesource.com/platform/art/+/7b074bf1ce559541d0c19ef793d9702a415ff74d/runtime/dex_file.cc

[39] SEHR, D.; CHARTIER, M.; HUGHES, E.; aj.: Soubor `dex_file_loader.cc`, *Google Git*. [zdrojový kód], červenec 2019, [cit. 2020-02-19]. Dostupné z: https://android.googlesource.com/platform/art/+/master/libdexfile/dex/dex_file_loader.cc

[40] ZHANG, V.: Janus Android App Signature Bypass Allows Attackers to Modify Legitimate Apps. *Trend Micro Security Intelligence Blog*, prosinec 2017, [online], [cit. 2020-02-18]. Dostupné z: https://blog.trendmicro.com/trendlabs-security-intelligence/janus-android-app-signature-bypass-allows-attackers-modify-legitimate-apps/

[41] ANDROID OPEN SOURCE PROJECT: Android Security Bulletin – December 2017. *Android Source*, prosinec 2017, [online], [cit. 2020-02-20]. Dostupné z: https://source.android.com/security/bulletin/2017-12-01

[42] KAMATH, N.: Commit `9dced16`. *Google Git*, srpen 2017, [online], [cit. 2020-02-20]. Dostupné z: https://android.googlesource.com/platform/system/core/+/9dced1626219d47c75a9d37156ed7baeef8f6403%5E%21/#F0

[43] KAMATH, N.: Soubor `zip_archive.cc`, *Google Git*. [zdrojový kód], srpen 2017, [cit. 2020-02-20]. Dostupné z: https://android.googlesource.com/platform/system/core/+/9dced1626219d47c75a9d37156ed7baeef8f6403/libziparchive/zip_archive.cc

[44] LESINSKI, A.; HUGHES, E.: Soubor `zip_archive_common.h`, *Google Git*. [zdrojový kód], říjen 2015, [cit. 2020-02-20]. Dostupné z: `https://android.googlesource.com/platform/system/core/+/9dced1626219d47c75a9d37156ed7baeef8f6403/libziparchive/zip_archive_common.h`

[45] BURKE, D.: Introducing Android 8.0 Oreo. *Android Developers Blog*, srpen 2017, [online], [cit. 2020-02-20]. Dostupné z: `https://android-developers.googleblog.com/2017/08/introducing-android-8-oreo.html`

[46] KASTRENAKES, J.; BRANDOM, R.: Google mandates two years of security updates for popular phones in new Android contract. *The Verge*, říjen 2018, [online], [cit. 2020-02-20]. Dostupné z: `https://www.theverge.com/2018/10/24/18019356/android-security-update-mandate-google-contract`

[47] ANDROID DEVELOPERS: Distribution Dashboard. *Android Developers*, květen 2019, [online], [cit. 2020-02-20]. Dostupné z: `https://developer.android.com/about/dashboards/index.html`

[48] HAZUM, A.; HE, F.; MAROM, I.; aj.: "Agent Smith": The New Virus to Hit Mobile Devices. *Check Point Blog*, červenec 2019, [online], [cit. 2020-02-20]. Dostupné z: `https://blog.checkpoint.com/2019/07/10/agent-smith-android-malware-mobile-phone-hack-virus-google/`

[49] HAZUM, A.; HE, F.; MAROM, I.; aj.: Agent Smith: A New Species of Mobile Malware. *Check Point Research*, červenec 2019, [online], [cit. 2020-02-20]. Dostupné z: `https://research.checkpoint.com/2019/agent-smith-a-new-species-of-mobile-malware/`

[50] ANDROID DEVELOPERS: Manifest.permission: `RECORD_AUDIO`. *Android Developers*, únor 2020, [online], [cit. 2020-02-26]. Dostupné z: `https://developer.android.com/reference/android/Manifest.permission`

[51] ANDROID DEVELOPERS: Services overview. *Android Developers*, únor 2020, [online], [cit. 2020-02-26]. Dostupné z: `https://developer.android.com/guide/components/services`

[52] NÁLEVKA, P.; ROHR, L.; MATULA, M.; aj.: Smartphones are turning back into dumbphones. *Don't kill my app!*, leden 2020, [online], [cit. 2020-02-26]. Dostupné z: `https://dontkillmyapp.com/`

[53] ANDROID DEVELOPERS: Broadcasts overview. *Android Developers*, prosinec 2019, [online], [cit. 2020-02-26]. Dostupné z: `https://developer.android.com/guide/components/broadcasts`

[54] GRUVER, B.: Smali Wiki. *GitHub*, únor 2020, [online], [cit. 2020-02-27]. Dostupné z: https://github.com/JesusFreke/smali/wiki

[55] ANDROID DEVELOPERS: Sharing audio input. *Android Developers*, únor 2020, [online], [cit. 2020-03-06]. Dostupné z: https://developer.android.com/guide/topics/media/sharing-audio-input

[56] ANDROID DEVELOPERS: Request App Permissions. *Android Developers*, prosinec 2019, [online], [cit. 2020-03-06]. Dostupné z: https://developer.android.com/training/permissions/requesting

[57] BURG, J.; ROMNEY, J.; SCHWARTZ, E.: *Digital Sound & Music: Concepts, Applications, and Science*, kapitola 5.2.3 Latency and Buffers. Franklin, Beedle & Associates Inc, 2016, ISBN 978-1590282748, [online]. Dostupné z: http://digitalsoundandmusic.com/5-2-3-latency-and-buffers/

[58] ANDROID DEVELOPERS: AudioRecord. *Android Developers*, únor 2020, [online], [cit. 2020-03-26]. Dostupné z: https://developer.android.com/reference/android/media/AudioRecord

[59] SAMPSON, P.: Sample rates. *Audacity Manual*, červen 2018, [online], [cit. 2020-03-06]. Dostupné z: https://manual.audacityteam.org/man/sample_rates.html

[60] BUCHHOLZ, F.: The structure of a PKZip file. *Userspace of James Madison University*, červen 2015, [online], [cit. 2020-03-11]. Dostupné z: https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html

[61] van EMDEN, R.: BabyApp. *GitHub*, červenec 2018, [online], [cit. 2020-03-14]. Dostupné z: https://github.com/robinvanemden/BabyApp

[62] TUMBLESON, C.: A tool for reverse engineering Android apk files. *Apktool*, únor 2020, [online], [cit. 2020-03-14]. Dostupné z: https://ibotpeaches.github.io/Apktool/

[63] FRIESEN, J.: Study guide: Classes within classes. *JavaWorld*, únor 2002, [online], [cit. 2020-03-16]. Dostupné z: https://www.javaworld.com/article/2074001/study-guide--classes-within-classes.html

[64] HUA, G.: Sound Recorder. *GitHub*, prosinec 2012, [online], [cit. 2020-03-12]. Dostupné z: https://github.com/MiCode/SoundRecorder

[65] HUA, G.: Sound Recorder. *F-Droid*, srpen 2012, [online], [cit. 2020-03-12]. Dostupné z: https://f-droid.org/en/packages/net.micode.soundrecorder/

[66] ANDROID DEVELOPERS: Apksigner. *Android Developers*, březen 2020, [online], [cit. 2020-03-25]. Dostupné z: https://developer.android.com/studio/command-line/apksigner

[67] ORACLE: Jarsigner. *Java SE Documentation*, 2018, [online], [cit. 2020-03-25]. Dostupné z: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

[68] ANDROID DEVELOPERS: Android Debug Bridge (adb). *Android Developers*, březen 2020, [online], [cit. 2020-03-26]. Dostupné z: https://developer.android.com/studio/command-line/adb

[69] MUHAMAD, A.: Recording in Android giving exceptions. *StackOverflow*, srpen 2011, [online], [cit. 2020-03-26]. Dostupné z: https://stackoverflow.com/a/7169939/6136143

[70] ATERLUX: Android emulator audio input capturing. *StackOverflow*, červenec 2015, [online], [cit. 2020-03-26]. Dostupné z: https://stackoverflow.com/a/31214558/6136143

[71] DRAKE, N.; TURNER, B.: Best Android antivirus app of 2020. *Techradar Pro*, prosinec 2019, [online], [cit. 2020-04-02]. Dostupné z: https://www.techradar.com/best/best-android-antivirus-app

[72] BITDEFENDER: Bitdefender Mobile Security & Antivirus. *Bitdefender*, březen 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.bitdefender.com/solutions/mobile-security-android.html

[73] McAFEE: McAfee® Mobile Security pro Android. *McAfee*, únor 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.mcafee.com/consumer/cs-cz/store/m0/catalog/mmsa_459/mcafee-mobile-security-android-free.html

[74] KASPERSKY LAB: Kaspersky Mobile Antivirus: AppLock & Web Security. *Google Play*, březen 2020, [online], [cit. 2020-04-02]. Dostupné z: https://play.google.com/store/apps/details?id=com.kms.free

[75] AVIRA: Avira Antivirus Security for Android, now with Free VPN. *Avira*, březen 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.avira.com/en/free-antivirus-android

[76] ESET: ESET Mobile Security & Antivirus. *ESET*, duben 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.eset.com/cz/domacnosti/android-antivir/

[77] MALWAREBYTES: Malwarebytes Security: Virus Cleaner, Anti-Malware. *Malwarebytes*, únor 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.malwarebytes.com/android/

[78] AVAST SOFTWARE: Avast Antivirus – Mobile Security & Virus Cleaner. *Avast*, březen 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.avast.com/cs-cz/free-mobile-security

[79] AVG NOBILE: AVG AntiVirus 2020 for Android Security Free. *AVG*, březen 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.avg.com/cs-cz/antivirus-for-android

[80] TAPI SECURITY LABS: Antivirus & Virus Cleaner, Applock, Clean, Booster. *Google Play*, březen 2020, [online], [cit. 2020-04-02]. Dostupné z: https://play.google.com/store/apps/details?id=com.antivirus.mobilesecurity.viruscleaner.applock

[81] TREND MICRO: Mobile Security Solutions. *Trend Micro*, únor 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.trendmicro.com/en_us/forHome/products/mobile-security.html?cm_mmc=VURL:USA-_-Mobile-_-Android-_-Android+Security

[82] VIRUSTOTAL: Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community. *VirusTotal*, duben 2020, [online], [cit. 2020-04-02]. Dostupné z: https://www.virustotal.com/gui/home/upload

Příloha A

Seznam použitých zkratek

aapt Android Asset Packaging Tool

AIDL Android Interface Definition Language

AOSP Android Open Source Project

AOT Ahead-of-time

API Application programming interface

APK Android Package

ART Android Runtime

CRC Cyclic redundancy check

DEX Dalvik executable

GUI Graphical user interface

HAL Hardware abstraction layer

JAR Java archive

JIT Just-in-time

NDK Native development kit

ODEX Optimized DEX

PKI Public key infrastructure

SF Signature file

UID User identifier

XML Extensible markup language

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou aplikací
  - `server.jar` ..... JAR balíček serverové aplikace
  - `exploited_emu-BA.apk` .... škodlivé APK pro emulátor (*BabyApp*)
  - `exploited_emu-SR.apk` ..... škodlivé APK pro emulátor (*Sound Recorder*)
  - `janus_exploit` .... Python balíček aplikace pro spojení DEX a APK
- `src` ..... zdrojové kódy aplikací
  - `android` ..... zdrojový kód služby pro odposlech
  - `server` ..... zdrojový kód serverové aplikace
  - `janus_exploit` ..... zdrojový kód aplikace pro spojení DEX a APK
  - `disassembly` ..... disassemblované aplikace pro OS Android
    - `neupravene` ..... neupravené legitimní aplikace
    - `upravene` ..... legitimní aplikace s přidaným odposlechem
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `video` ..... video ukázky útoku

Příloha C

Instalace a spuštění aplikace pro spojení DEX a APK

Aplikace `janus_exploit`, která za DEX soubor připojí APK archiv tak, aby výsledný soubor využíval zranitelnost Janus, je napsaná v jazyce Python. Pro její snadnou instalaci byl vytvořen zdrojový balíček typu `sdist`, který je uložen na přiloženém CD – konkrétně jde o soubor `exe/janus_exploit/janus_exploit-1.0.tar.gz`.

Pro spuštění této aplikace je potřeba mít v systému nainstalovaný Python verze 3.6 nebo vyšší. Před instalací je vhodné vytvořit a aktivovat virtuální prostředí, do kterého se aplikace a všechny potřebné balíčky nainstalují. Virtuální prostředí se vytváří příkazem `python -m venv prostredi`. Poté je nutné ho aktivovat pomocí jednoho z následujících příkazů (první je pro Linux, druhý pro Windows):

```
. prostredi/bin/activate
prostredi/Scripts/activate
```

Instalace aplikace `janus_exploit` do virtuálního prostředí nainstaluje i všechny balíčky, na kterých je aplikace závislá (například *Click*). Aplikace `janus_exploit` se nainstaluje následujícím příkazem:

```
python -m pip install janus_exploit-1.0.tar.gz
```

Po instalaci lze balíček spustit příkazem `python -m janus_exploit`, kterému se zadají žádané argumenty – DEX soubor, APK archiv a název výstupního souboru.

Přiložené CD obsahuje i adresář se zdrojovými kódy této aplikace – `src/janus_exploit`. V adresáři je projekt pro vývojové prostředí *PyCharm*, ale soubory lze upravovat libovolným textovým editorem a spouštět z příkazové řádky. Z tohoto adresáře je možné aplikaci `janus_exploit` spustit i bez instalace, konkrétně příkazem `python -m janus_exploit`. Nevýhodou tohoto způsobu spouštění je, že všechny balíčky, které aplikace potřebuje, je nutné nainstalovat ručně.

Příloha D

Tvorba škodlivého instalačního balíčku

Na CD se již nachází škodlivý instalační balíček (spojený DEX upravené legitimní aplikace a neupravený APK archiv této aplikace) pro použití na emulátoru vývojového prostředí *Android Studio*. Pokud má být škodlivý balíček použit na jiném zařízení, je nutné uvnitř přepsat IP adresu na tu, na které je spuštěna serverová aplikace (emulátor má pro komunikaci s hostitelským systémem pevnou IP adresu 10.0.2.2). V této příloze je popsáno nejen jak adresu změnit, ale i jak sestavit jednotlivé části škodlivého balíčku pouze ze zdrojového kódu, například po úpravě odposlechové služby.

D.1 Úprava IP adresy

Pokud je potřeba pouze změnit IP adresu (případně i port), na kterou má škodlivá aplikace odesílat data, stačí ji přepsat v jednom souboru.

Adresář `src/disassembly/upravene` obsahuje adresáře s rozbalenými APK archivy legitimních aplikací. V těchto adresářích se nachází již upravený *Smali* kód – je do něj přesunuta funkcionalita služby pro odposlech. Ve službě `EavesdropService` je IP adresa uložena jako textový řetězec v proměnné `address`.

Inicializace členské proměnné `address` se v případě aplikace *BabyApp* nachází v souboru `BabyService.smali` uloženém v adresáři `babyapp/smali/com/hollandhaptics/babyapp`. V případě aplikace *Sound Recorder* je hodnota IP adresy (proměnné `address`) inicializována v souboru `RecorderService.smali`, který je uložen v adresáři `soundrecorder/smali/net/micode/soundrecorder`.

Přiřazení hodnoty (řetězce `"10.0.2.2"`) členské proměnné `address` se v obou případech nachází v konstruktoru, tedy v metodě `public constructor <init>()V`. V případě *BabyApp* je řetězec na řádku 80 (viz

výpis D.1, v případě *Sound Recorder* na řádku 134. Pro změnu IP adresy stačí přepsat řetězec na požadovanou adresu a uložit změny v souboru.

```
61 # direct methods
62 .method public constructor <init>()V
63     .locals 3
       # ...
80     const-string v0, "10.0.2.2"
82     iput-object v0, p0, Lcom/hollandhaptics/babyapp/
          BabyService;->address:Ljava/lang/String;

84     .line 29
85     const v0, 0xc355
87     iput v0, p0, Lcom/hollandhaptics/babyapp/
          BabyService;->port:I
```

Výpis D.1: Úryvek z konstruktoru služby `BabyService` s přidanou funkcionalitou odposlechové služby `EavesdropService`. Na řádku 80 je do registru `v0` vložena řetězcová hodnota, kterou je na řádku 82 inicializována IP adresa, na kterou škodlivá aplikace odesílá data. Pro změnu IP adresy stačí přepsat hodnotu na řádku 80. Stejným způsobem lze přepsat i hodnotu portu, která je inicializována hned po IP adrese. Na řádku 85 je do registru `v0` přiřazena celočíselná konstanta 0xC355, (50005 v dekadické soustavě). Tato hodnota je na řádku 87 přiřazena proměnné `port`. Pro změnu čísla portu stačí přepsat hexadecimální hodnotu na řádku 85.

Přímo pod přiřazením IP adresy se stejným způsobem přiřazuje i hodnota portu, kterou je též možné změnit (stejnou hodnotu je potom nutné nastavit i v serverové aplikaci). Číslo portu je celočíselná proměnná a ve *Smali* je zapsaná v hexadecimální formě.

Po změně IP adresy je nutné sestavit upravený *Smali* kód zpět do DEX bytekódu. To lze provést příkazem `apktool b ADRESAR`, kde `ADRESAR` je adresář s rozbaleným APK archivem aplikace, například `src/disassembly/upravene/babyapp`. Po sestavení se v adresáři `ADRESAR` vytvoří podadresář `build/apk`, který obsahuje soubor `classes.dex`. Tento soubor obsahuje bytekód upravené aplikace a poslouží při tvorbě škodlivého instalačního balíčku jako DEX soubor, za který se připojí APK archiv legitimní aplikace. Při instalaci tohoto balíčku do zařízení s operačním systémem Android se tento DEX soubor nahraje do úložiště zařízení místo souboru `classes.dex`, který je obsažen uvnitř APK archivu legitimní aplikace.

Nyní stačí soubor `classes.dex` spojit s neupraveným APK archivem legitimní aplikace. APK archivy obou legitimních aplikací se nachází v adresáři `src/disassembly/neupravene`). Spojení lze provést pomocí apli-

kace `janus_exploit`, konkrétně příkazem `python -m janus_exploit classes.dex legitimni.apk skodlive.apk`. Nově vytvořený balíček `skodlive.apk` využívá zranitelnost Janus a je možné ho nainstalovat do zranitelného zařízení s operačním systémem Android. Nainstalována bude upravená aplikace, i když kontrola podpisu úspěšně ověří podpis certifikátem uloženým v APK archivu legitimní aplikace.

D.2 Sestavení balíčku ze zdrojového kódu

V této sekci je popsáno sestavení celého škodlivého instalačního balíčku ze zdrojového kódu. Na přiloženém CD se nachází adresář `src/android`, ve kterém je projekt pro vývojové prostředí *Android Studio*, který obsahuje zdrojový kód dvou tříd – `MainActivity` a `EavesdropService`. Tyto třídy obsahují funkcionalitu odposlechu, která je vložena do legitimní aplikace.

Kód těchto tříd je možné upravit a následně sestavit v prostředí *Android Studio*. Sestavení se spustí v nabídce *Build – Build Bundle(s) / APK(s) – Build APK*. Vytvořený instalační balíček lze použít pro odladění odposlechu. Nachází se v adresáři `android/app/build/outputs/apk/debug`.

Pro tvorbu škodlivého instalačního balíčku je nutné výše vytvořený instalační balíček *disassemblovat*, což lze provést příkazem `apktool d app-debug.apk`.

Nyní je nutné přesunout funkcionalitu odposlechu do *disassemblovaných* legitimních aplikací. Adresáře s jejich rozbalenými APK archivy a *disassemblovaným* kódem jsou uloženy v adresáři `src/disassembly/neupravene`. Postup pro tento krok (jaké části je nutné přesunout a kam) je podrobně popsán v sekci 3.4.

Po přesunu funkcionality odposlechu do *Smali* souborů je postup vytvoření škodlivého instalačního balíčku totožný, jako po úpravě IP adresy ve *Smali* kódu, jak je popsáno v předchozí sekci.

D.3 Instalace škodlivého balíčku

Do zařízení s operačním systémem Android lze balíček nainstalovat několika způsoby. Balíček je možné přesunout do úložiště zařízení (např. přes USB, Bluetooth apod.) a nainstalovat z něj. Pro tuto instalaci nejspíše bude nutné povolit instalaci aplikací třetích stran v nastavení zařízení. Dále může být potřeba deaktivovat antivirové kontroly (*Google Play Protect*, případně i dodatečně nainstalované antivirové aplikace). Balíček se z úložiště nainstaluje přes prohlížeč souborů. V prohlížeči stačí nalézt adresář, kam byl balíček zkopírován, a na balíček jednou klepnout.

Druhá možnost je balíček nainstalovat přes nástroj *Android Debug Bridge*. Na zařízení je nutné mít povolený vývojářský režim. Ten se povolí tak, že se sedmkrát zaklepe na položku *Číslo sestavení* v nabídce *Nastavení – Info*

o telefonu. Instalace balíčku po připojení zařízení k PC přes USB se poté provede příkazem `adb install balicek.apk`. Po dokončení tohoto příkazu je škodlivá aplikace nainstalována a je možné ji na zařízení spustit.

D.4 Spuštění škodlivé aplikace

Při prvním spuštění nainstalované škodlivé aplikace je nutné přidělit jí oprávnění, o která požádá – v závislosti na verzi operačního systému buď nepožádá o žádná (seznam požadovaných oprávnění mohl být vypsán a odsouhlasen již při instalaci z úložiště), nebo požádá o přístup k mikrofonu.

Pokud je škodlivá aplikace spouštěna na zařízení, které podle iniciativy *Don't kill my app!* [52] potřebuje zvláštní opatření ke korektnímu chování služeb (*Huawei, Samsung, Xiaomi, OnePlus* atd.), je po každém spuštění aplikace v dolní části obrazovky vypsána hláška s instrukcemi, jaká nastavení je potřeba změnit, aby aplikace mohla fungovat na pozadí. Pokud se tato nastavení nezmění, je pravděpodobné, že po ukončení škodlivé aplikace přestane odposlech fungovat. Ve vypsané hlášce jsou obecné instrukce, které se vážou k danému výrobci zařízení. Může se tedy stát, že se bude postup zakazování úsporného režimu na konkrétním zařízení lišit.

Po spuštění škodlivé aplikace založené na aplikaci *BabyApp* (klepnutí na její ikonu v nabídce aplikací) je odposlechová služba spuštěna ihned. Legitimní aplikace ani nemá žádnou další možnost interakce s uživatelem – pouze vypíše text o spuštěné službě.

V případě škodlivé aplikace založené na aplikaci *Sound Recorder* je nutné pro spuštění odposlechu stisknout tlačítko s červeným kruhem. Po tom, co se spustí odposlechová služba, již serverová aplikace automaticky začne přijímat data. Další stisknutí libovolného tlačítka aplikace nemá žádný efekt.

Upozornění: Pokud se zařízení se škodlivou aplikací nachází v blízkosti zařízení se serverovou aplikací, je doporučeno k serverovému zařízení připojit sluchátka. V opačném případě dochází k nepříjemné silné zpětné vazbě.

D.5 Ukončení odposlechu

Škodlivá aplikace neimplementuje žádnou obranu pro zachování odposlechu po odinstalování aplikace. Odinstalováním škodlivé aplikace tedy odposlech končí a obnovit ho je možné pouze opětovnou instalací a spuštěním některé ze škodlivých aplikací.

Ukončit odposlech je možné též přes vývojářská nastavení. Pokud je na zařízení povolen vývojářský režim, nachází se v nastavení zařízení položka *Pro vývojáře* (*Developer Options*). V této nabídce je dále položka *Spuštěné služby*, případně *Spuštěné procesy*, ve kterých lze škodlivou aplikaci nalézt a ukončit odposlech zde – viz obrázek 4.8 na straně 92.

Příloha E

Spuštění serverové aplikace

Na přiloženém CD je v adresáři `exe` uložený soubor `server.jar`. Tento soubor je archivem jazyka Java – zkompilovaným spustitelným balíčkem aplikace, která na útočníkově serveru přijímá a zpracovává odposlechnutá data od škodlivých aplikací na zařízeních s operačním systémem Android.

Ke spuštění tohoto balíčku je nutné na zařízení, které má fungovat jako server, mít nainstalované prostředí jazyka Java verze alespoň 11 (*Java major version 55.0*). Balíček se potom spustí následujícím příkazem:

```
java -jar server.jar
```

Při spouštění je možné serveru předat argumenty pro jeho konfiguraci či pro nápovědu. V případě nepředání některého z konfiguračních argumentů je použita jeho výchozí hodnota. Argumenty jsou následující:

- `-t` T: Změní počet vláken zpracovávajících data na T. Výsledný počet využitých vláken je $T + 2$ (T vláken zpracovává data, jedno vlákno čte a zpracovává vstup uživatele a jedno vlákno přijímá data ze sítě a předává je ke zpracování). Výchozí hodnota je $T = 2$.

- `-m` M: Změní velikost paměti vyhrazené pro předpřipravený záznam každého spojení na M megabytů. Protože zvuk je zakódován do dvoubytových vzorků s frekvencí 44 100 vzorků za sekundu, M megabytů odpovídá $(M \cdot 1000000) \,/\, (2 \cdot 44100) \approx M \cdot 11,3$ sekundám přednahraného záznamu. Výchozí hodnota je $M = 10$.

- `-c` C: Změní maximální počet spojení, od kterých jsou přijímána data (a kterým jsou alokovány prostředky – kruhové pole, výstupní proud pro zápis do souboru) na C. Pokud $C = -1$ (výchozí hodnota), není počet nijak omezen.

- `-h`: Vypíše nápovědu k předchozím třem argumentům.

Na CD je dále přiložen adresář `src/server`, který obsahuje zdrojový kód této aplikace. V tomto adresáři je uložen projekt vývojového prostředí *IntelliJ IDEA*. Samotné zdrojové soubory jsou potom uloženy v adresáři `src/server/src/com/company`.

Aplikace poslouchá na portu 50005 (tento port je nastavený i ve škodlivé aplikaci). Pokud by bylo nutné ho změnit, lze změnu provést v souboru `ServerThread.java`. Hodnota portu je přiřazena na začátku metody `run` (řádek 68). V prostředí *Intellij IDEA* se nový spustitelný JAR archiv sestaví v nabídce *Build – Build Artifacts... – JanusServer.jar*. Nový JAR archiv je potom uložen v adresáři `src/server/out/artifacts/JanusServer_jar`.

E.1 Ovládání aplikace

Po spuštění aplikace jsou nová spojení přidávána automaticky po přijetí prvních dat od daného zdroje (kombinace IP adresy a portu). Pokud od spojení po dobu dvou sekund nejsou přijata žádná data, spojení je ze seznamu známých spojení automaticky odstraněno.

V serverové aplikaci lze zadávat příkazy, které umožňují přepínat mezi spojeními nebo odposlech daného spojení nahrávat do souboru. Jde o jednoznakové příkazy, z nichž některé vyžadují identifikátor spojení. Identifikátor každého spojení (dále uváděn jako `SPOJ`) má vždy tvar `IP:port`. Kromě příkazu „`s`“ lze v ostatních příkazech identifikátor `SPOJ` nahradit znakem „`a`“, což značí spojení, které je právě aktivní (přehrává se v reproduktorech).

V serverové aplikaci lze zadávat následující příkazy:

- `h`: Zobrazení nápovědy k ostatním příkazům.
- `l`: Výpis všech aktuálních spojení.
- `s SPOJ`: Spojení `SPOJ` se bude přehrávat v reproduktorech (bude aplikací označeno jako *aktivní*).
- `d SPOJ`: Odstraní spojení `SPOJ` ze seznamu. Případné nahrávání spojení je ukončeno. Po dobu 1 minuty jsou data od spojení ignorována a pokud jsou od něj po této době přijata nová data, je znovu přidáno do seznamu (pokud není naplněna kapacita daná argumentem `-c`).
- `r SPOJ`: Začne nahrávat spojení `SPOJ` do souboru.
- `c SPOJ`: Zastaví nahrávání spojení `SPOJ`.

Záznam spojení se ukládá do souboru ve formátu WAV. Soubor se vytvoří automaticky v adresáři, odkud je JAR spuštěn. Název každého souboru má formát `IP.port_čas.wav`, kde `čas` je časová známka počátku nahrávání. Například takto vygenerovaný název souboru pro připojení ze zařízení s IP adresou 192.168.0.102 a portem 36774, jehož nahrávání bylo spuštěno 5. 4. 2020 v 17:41:20, bude `192.168.0.102.36774_2020.04.05.17.41.20.wav`.