Sigurnosne prijetnje mobilnih aplikacija i tehnike zaštite

Škunca, Karolina

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij računarstva

Diplomski rad

SIGURNOSNE PRIJETNJE MOBILNIH APLIKACIJA I TEHNIKE ZAŠTITE

Rijeka, srpanj 2022.

Karolina Škunca

0069069663

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij računarstva

Diplomski rad

SIGURNOSNE PRIJETNJE MOBILNIH APLIKACIJA I TEHNIKE ZAŠTITE

Mentor: Prof. dr. sc. Kristijan Lenac

Rijeka, srpanj 2022.

Karolina Škunca

0069069663

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 12. ožujka 2021.

Zavod: **Zavod za računarstvo**
Predmet: **Napredni operacijski sustavi**
Polje: **2.09 Računarstvo**

ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Karolina Škunca (0069069663)**
Studij: Diplomski sveučilišni studij računarstva
Modul: Programsko inženjerstvo

Zadatak: **Sigurnosne prijetnje mobilnih aplikacija i tehnike zaštite / Mobile application security threats and mitigation measures**

Opis zadatka:

U radu je potrebno istražiti, analizirati i kategorizirati moguće prijetnje sigurnosti mobilnih aplikacija. Opisati tehnike zaštite za odabrane važnije sigurnosne prijetnje koje se koriste prilikom razvoja i testiranja mobilnih aplikacija na iOS operacijskom sustavu.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 15. ožujka 2021.

Mentor:

Izv. prof. dr. sc. Kristijan Lenac

Predsjednik povjerenstva za diplomski ispit:

Izv. prof. dr. sc. Kristijan Lenac

IZJAVA O SAMOSTALNOJ IZRADI RADA

Izjavljujem da sam ovaj rad napravila samostalno uz znanja stečena tijekom studija te korištenjem navedene literature.

Karolina Škunca

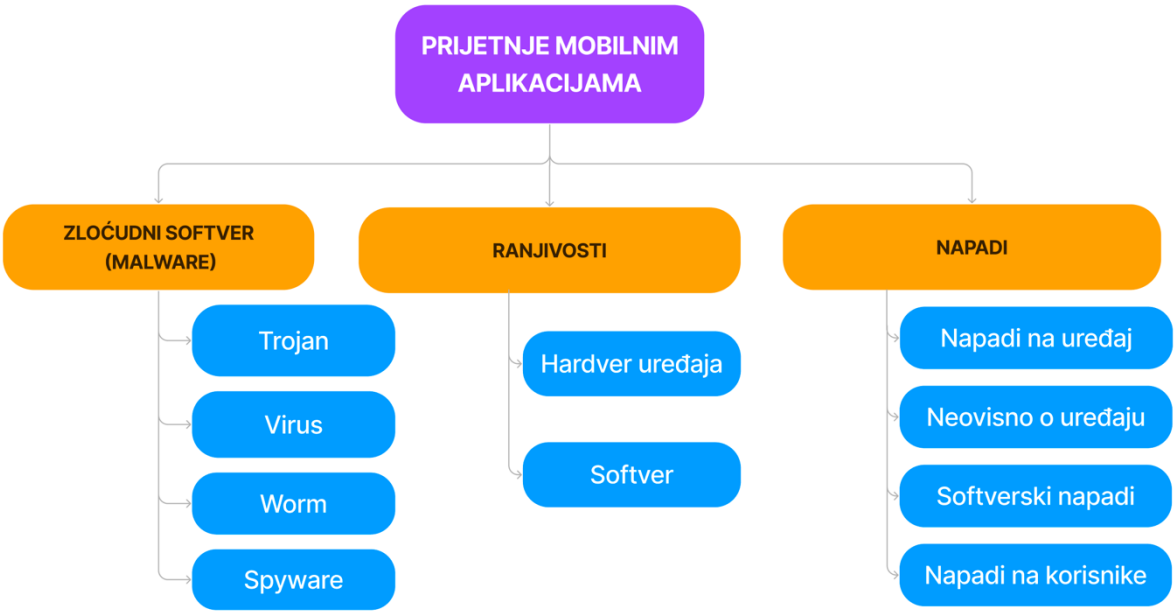
ZAHVALA

Zahvaljujem se svom mentoru prof. dr. sc. Kristijanu Lencu, tvrtki Asseco i njenim zaposlenicima na pruženoj pomoći i znanju i svim bližnjima koji su mi bila podrška.

Sadržaj

1. Uvod

Korisnici se svakodnevno na svojim mobilnim uređajima bave raznim aktivnostima – razmjenjuju poruke, osobne i financijske podatke i još mnogo toga. Povećanom popularnosti mobilnih uređaja, dolaze i nove prijetnje. Ubrzanim razvojem hardverskih i softverskih komponenata mobilnih uređaja može doći do propusta u sigurnosnim procedurama sustava, internim kontrolama, dizajnu i aplikacijama. Prijetnje sustavima također predstavljaju zlonamjerni softveri koji su usmjereni na krajnjeg korisnika kako bi dohvatili korisnikove podatke ili kako bi sustav postao neuporabljiv. Kombinacija zlonamjernih programa i slabosti sustava mogu se predstaviti kao napadi na sustav koje haker izvršava u cilju dobivanja korisnikovih podataka bez njihova znanja [1].



Slika 1.1. Prikaz prijetnji mobilnim sustavima [1].

U ovom radu se primarno fokusira na prijetnje mobilnim aplikacijama i njihovoj sigurnosti.

Korisnikovo očekivanje od programera aplikacija je da testiraju svoje aplikacije, spriječe napade i osiguraju sigurnost podataka krajnjih korisnika. Sigurnost mobilne aplikacije jedan je od najvažnijih aspekata i značajki koje jedna aplikacija mora imati. Aplikacija bi trebala imati mjere zaštite od vanjskih prijetnji kao što su zlonamjerni napadači ili od zlonamjernog softvera. U digitalnom svijetu u kojem živimo postaje neophodno biti svjestan uobičajenih sigurnosnih problema aplikacija i načina kako zaštititi svoju aplikaciju.

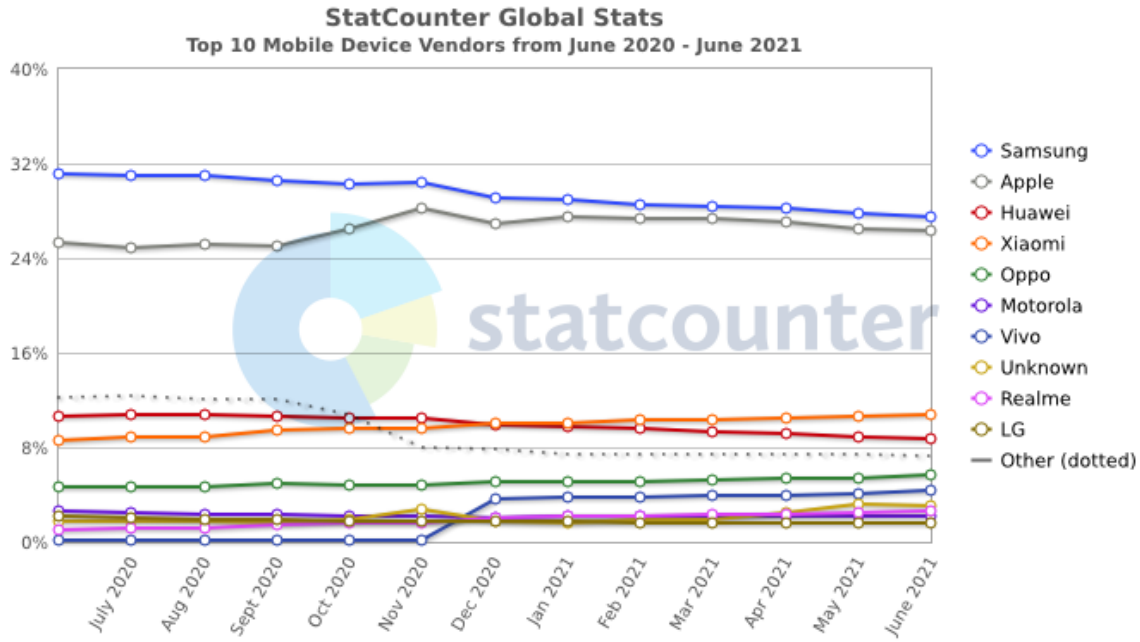
Zbog brzog rasta tržišta i konstantnih promjena u ponudi i potražnji, kompanije i pojedinci stavljaju aplikacije na tržište zanemarujući neke ključne aspekte sigurnosnih implementacija aplikacije. Nakon što je aplikacija dostupna za preuzimanje nakon objavljivanja, svatko, uključujući hakere, ima pristup aplikaciji i njezinu kodu. Ako aplikacija nije dovoljno sigurna, zlonamjerna osoba ili organizacija mogu skenirati aplikaciju i vidjeti ranjivosti unutar koda aplikacije. Dodatno, aplikacija može biti podložna rastavljanju aplikacije, što omogućuje zlonamjernim osobama ili grupama da zavire u pozadinu aplikacije ili se umetnu u komunikaciju između aplikacije i poslužitelja organizacije radi prikupljanja vrijednih informacija. Tako dolazi do presretanja i modifikacije podataka, što je napad na povjerljivost i integritet korisničkih podataka.

Kroz rad će se opisati i objasniti arhitektura najkorištenijih mobilnih operacijskih sustava, Android i iOS, kako bi razumjeli pristup testiranju sigurnosti aplikacija, a potom će se obraditi i moguće prijetnje aplikacijama i njihove slabosti te načini na koje je moguće obraniti se od napada na aplikaciji prilikom razvoja iste, ali i kasnije kada aplikacija nije više u rukama programera.

2. Mobilni sustavi

Mobilnim uređajima se smatraju uređaji koji su lako prenosivi, manjih dimenzija i mogu se koristiti u rukama. To su najčešće uređaji koji imaju LCD ili OLED ekran na kojem se odvija interakcija korisnika s uređajem. Njihova korisnost je u tome što imaju mogućnost spajanja na Internet, pregledavanja sadržaja, mogućnost snimanja videa i fotografija, mogućnost obavljanja sastanka, korištenje društvenih mreža te time postaju dio ljudske svakodnevice.

Zbog popularnosti mobilnih uređaja, dolazi do velikog razvoja i natjecanja među proizvođačima. Najpoznatiji sustavi koji su se profilirali za mobilne sustave su iOS na Apple uređajima i *open-source* sustav Android čije inačice pronalazimo na proizvođačima poput Samsung, Huawei, Meizu, Zte, Xiaomi, Sony, Google, HTC, LG, TCL, Motorola Mobility i Nokia [2].



Slika 2.1. Tržišni udio dobavljača mobilnih uređaja u cijelom svijetu u razdoblju od srpnja 2020. do lipnja 2021 [2].

Prema statistici, danas u svijetu ima 2 milijarde korisnika na Android sustavu, dok na iOS sustavu ima 1.8 milijarde aktivnih korisnika [3].

Globalni market danas drže zajedno Google-ov Android i Apple-ov iOS, a na Android otpada skoro 70% udjela, dok 25% otpada na Apple-ov dio [4].

3. Android

3.1. Općenito

Android je mobilni operacijski sustav razvijan od strane konzorcija zvan Open Handset Alliance i sponzoriran od strane Google-a. Predstavljen je u studenom 2007, a prvi Android uređaj, HTC Dream, je bio predstavljen u rujnu 2008. godine.

Android sustav je besplatan softver otvorenog koda poznat pod Android Open Source Project, licenciran pod Apache licencom. Doduše, većina uređaja na bazi Android sustava dolaze s već unaprijed instaliranim dodatnim vlasničkim softverom poput Google Mobile Service koji uključuje osnovne aplikacije poput Google Chrome internetskog preglednika i Google Play platformu za digitalnu distribuciju i pridruženu razvojnu platformu usluga [6].

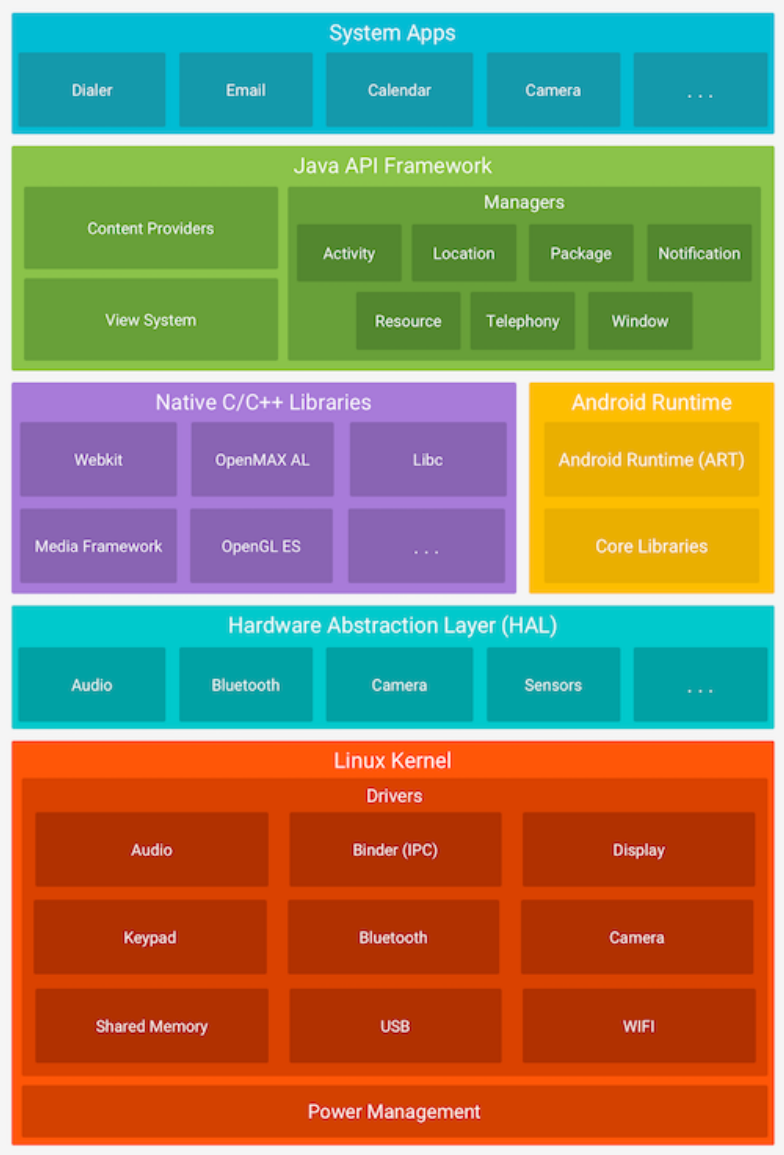
3.2. Povijest

Android je započeo kao projekt za razvoj operacijskog sustava za digitalne fotoaparate 2003. godine pod američkom tehnološkom tvrtkom Android Inc. Godine 2004., projekt se mijenja u projekt za operacijski sustav za mobilne telefone te 2005. godine, Android Inc. je kupljen od strane Google Inc. U Google-u je Android tim odlučio svoj projekt bazirati na Linux-u, operacijskom sustavu otvorenog koda za osobna računala.

5. studenog 2007. godine, Google najavljuje osnivanje Open Handset Alliance, konzorcija tehnoloških i mobilnih telefonskih tvrtki, uključujući Intel Corporation, Motorola, Inc., NVIDIA Corporation, Texas Instruments Incorporated, LG Electronics, Inc., Samsung Electronics, Sprint Nextel Corporation i T-Mobile (Deutsche Telekom). Cilj je bio razviti i promovirati Android kao besplatni operacijski sustav otvorenog koda s podrškom aplikacije trećih strana [7].

3.3. Arhitektura

Prikaz arhitekture Android sustava vidljiv je na Slici 3.1.:



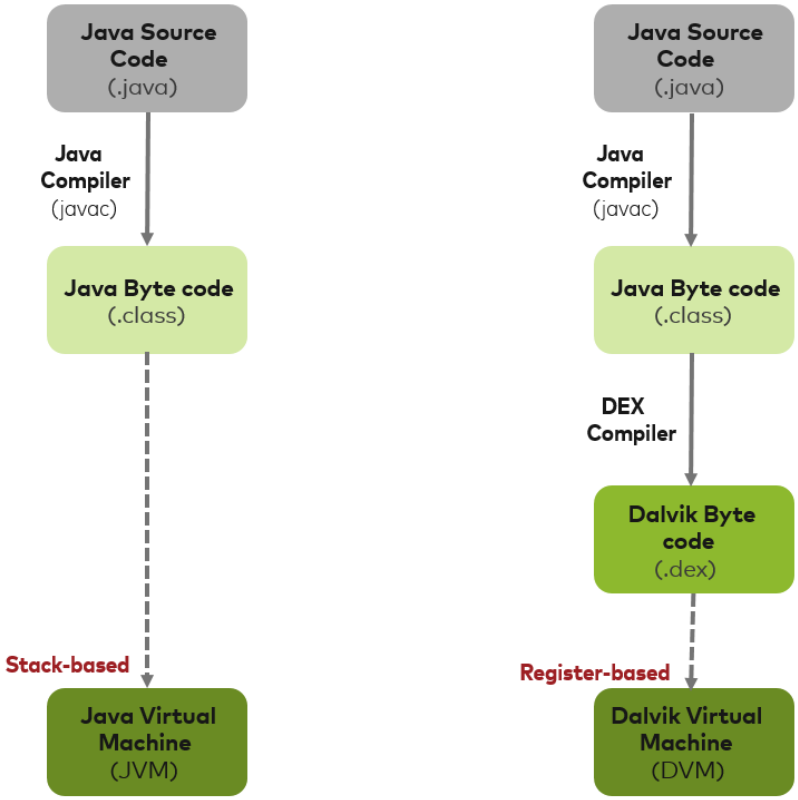
Slika 3.1. Vizualni prikaz arhitekture Android operacijskog sustava [8].

- Linux jezgra – Na najnižoj razini nalazi se varijacija Linux jezgre. To nije Linux u tradicionalnom smislu jer Android sustav ne uključuje GNU C biblioteku (koristi Bionic kao alternativnu C biblioteku) i neke druge komponente koje se obično nalaze u distribucijama Linuxa. Android verzija Linuxa sastoji se od posebnih dodataka kao što su: Low Memory Killer sustav za upravljanje memorijom koji služi boljem očuvanju memorije, *wake locks* kao usluga sustava PowerManager-a, upravljački program Binder IPC i druge.
- Hardware Abstraction Layer – Sloj iznad, nalazi se HAL ili Hardware Abstraction Layer gdje je definirano sučelje za komunikaciju s hardverskim komponentama uređaja kako ne bi bilo potrebno modificirati funkcije višeg nivoa. HAL komponente su

pakirane u module i Android ih poziva po potrebi. npr. Koristi se kod standardne telefonske aplikacije da se omogući korištenje mikrofona i slušalica.

- Sistemski servisi – Sistemski servisi su modularne komponente podijeljene u servise za na primjer pretraživanje, notifikacije i window manager. Aplikacijski okviri komuniciraju preko sistemskih servisa s hardverom.
- Binder IPC – Binder Inter-Process Communication služi kao spona između aplikacijskog okvira i sistemskih servisa kako bi high-level API-ji mogli koristiti sistemske pozive.
- Application framework ili aplikacijski okvir – Koriste se od strane programera te je poželjno biti svjestan razvojnih API-ja koji se mapiraju na temeljna HAL sučelja [8].

Android aplikacije su obično napisane u Javi i kompilirane u Dalvik bajt kod. Dalvik bajt kod se stvara tako da se prvo prevede Java kod u *.class* datoteke, a zatim se JVM bajt kod pretvori u Dalvik *.dex* format pomoću alata d8.



Slika 3.2. Prikaz razlika između Dalvik virtualnog stroja i Java virtualnog stroja [9].

Zadnja verzija Androida izvršava bajt kod u Android Runtime-u (ART). Prethodnik ART-a je bio Dalvik Virtual Machine za izvršavanje bajt koda. Ključna razlika između Dalvika i ART-a je način na koji se izvršava bajt kod.

U DVM-u se bajt-kod prevodi u strojni kod u vrijeme izvršenja, a taj proces se proces naziva kompilacija upravo na vrijeme (JIT). Zatim je Android predstavio ART za poboljšanje performansi. ART koristi hibridnu kombinaciju prijevremene (AOT), JIT-a i kompilacije vođene profilom. Aplikacije se ponovno kompiliraju na uređaju kada se instaliraju ili ako se OS podvrgne ažuriranju. Konačni ponovno kompilirani kod se tada koristi za sva naredna izvršenja.

3.4. Sigurnosni alati

Android ima ugrađene sigurnosne značajke kako bi se osiguralo kreiranje sigurnijih aplikacija. U narednim sekcijama će biti prikazane sigurnosne komponente Android sustava.

3.4.1 Aplikacijski *Sandbox*

Android sustav štiti aplikacijske podatke i resurse od drugih aplikacija tako da ih izolira od drugih aplikacija, a taj mehanizam se naziva *sandbox* aplikacije.

Android je baziran na Linuxu, ali nije podržano implementacija korisničkih računa kao i na ostalim Unix sistemima. U Androidu se taj sustav multi-korisnika koristi kao mehanizam za *sandbox* aplikacija; svaka aplikacija se pokreće kao zasebni Linux korisnik te se time izolira od drugih aplikacija i ostatka operacijskog sustava. To se postiže tako da se kreira novi proces i dodijeli jedinstveni UID te se to postavlja na razini jezgre. To znači da zlonamjerna aplikacija treće strane, s niskim privilegijama, ne bi trebala moći pobjeći iz vlastitog vremena izvođenja i čitati memoriju drugih aplikacija na istom uređaju.

3.4.2 Potpisivanje aplikacije

Kako bi se mogao identificirati autor aplikacije, potrebno je potpisati aplikaciju od strane developera. Time se može potvrditi autentičnost i sigurnost aplikacije. Kriptografski potpis služi kao provjerljiva oznaka koju postavlja programer aplikacije. Identificira autora aplikacije i osigurava da aplikacija nije promijenjena od početne distribucije.

3.4.3 Autentikacija

Kako bi aplikacija bila sigurna, preporučuje se korištenje kriptografije. Android koristi *user-authentication-gated* kriptografske ključeve koji zahtijevaju podržavanje spremanja kriptografskih ključeva, davatelja usluga i *user authenticator*.

Također, uređaji koji podržavaju prepoznavanje otiska prsta ili lica, pružaju dodatan sloj zaštite prilikom autentikacije. Podsustav koji izvodi provjeru autentičnosti zove se Gatekeeper te se izvršavanje izvodi u okruženju pouzdanog izvršavanja - Trusted Execution Environment ili TEE. Trusty je siguran operacijski sustav koji pruža pouzdano okruženje za izvršavanje. Trusty OS radi na istom procesoru kao i Android OS, ali je Trusty izoliran od ostatka sustava i hardverom i softverom.

Dodatno, od Androida 9 pa nadalje podržano je korištenje Protected Confirmation servisa. To omogućuje korisnicima siguran način za službenu potvrdu kritičnih transakcija kao što su plaćanja.

3.4.4 Biometrija

Android 9 i novije verzije uključuju *BiometricPrompt* API koji služi za integraciju biometrijske provjere autentičnosti u aplikaciji na način koji je neovisan o uređaju i modalitetu.

3.4.5 Enkripcija

Nakon što je uređaj šifriran, svi podaci koje je kreirao korisnik automatski se šifriraju prije predaje na disk, a sva čitanja automatski dešifriraju podatke prije nego što ih vrate u proces pozivanja. Šifriranje osigurava da čak i ako neovlaštena strana pokuša pristupiti podacima, neće ih moći pročitati.

3.4.6 Keystore

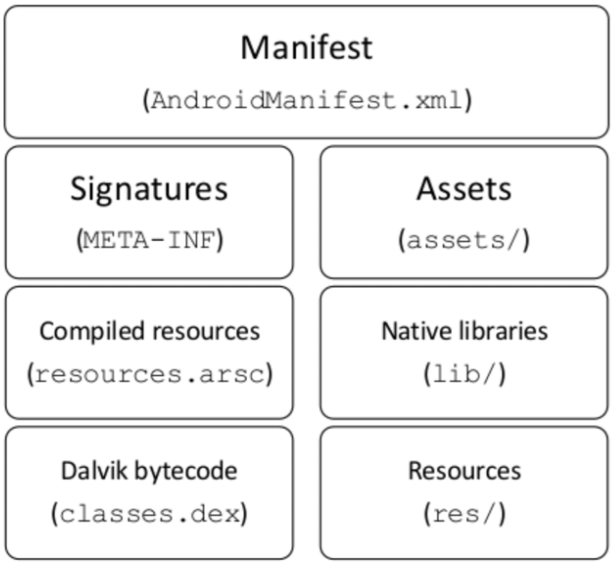
Keystore klasa predstavlja spremište za kriptografske ključeve i certifikate. To je hardverski poduprta pohrana ključeva koja pruža mogućnosti generiranje ključeva, uvoz i izvoz asimetričnih ključeva, uvoz sirovih simetričnih ključeva, asimetrično šifriranje i dešifriranje s odgovarajućim načinima dodavanja i još mnogo toga.

3.4.7 Verified Boot

Verified Boot nastoji osigurati da sav izvršeni kod dolazi iz pouzdanog izvora (obično od originalnog proizvođača opreme), a ne od napadača ili oštećenja. Uspostavlja puni lanac povjerenja, počevši od hardverski zaštićenog korijena povjerenja od *bootloader*-a, do *boot* particije i drugih provjerenih particija. Tijekom podizanja uređaja, svaka faza provjerava integritet i autentičnost sljedeće faze prije predaje procesa izvršenja [8] [10].

3.5. APK

APK datoteka je arhiva koja obično sadrži sljedeće datoteke i direktorije:



Slika 3.3. Struktura .apk datoteke.

- Manifest datoteka – Dodatna datoteka manifesta Androida, koja opisuje naziv, verziju, prava pristupa i referencirane datoteke biblioteke za aplikaciju.
- Potpisi ili *signatures* (META-INF) – Sadrži potvrde i potpise.
- Sredstva ili *assets* – Direktorij koji sadrži sredstva aplikacije koju AssetManager može dohvatiti.
- Prevedeni resursi ili *compiled resources* (resources.arsc) – datoteka koja sadrži unaprijed kompilirane resurse, kao što je binarni XML.
- Nativne knjižice ili *Native Libraries* (lib) – Direktorij koji sadrži prevedeni kod koji ovisi o platformi te može biti podijeljen u više direktorija unutar njega:
    - armeabi-v7a – Kompilirani kod samo za sve ARMv7 i novije procesore
    - arm64-v8a – Kompilirani kod za sve ARMv8 arm64 i novije bazirane procesore

    - x86 – Kompilirani kod samo za x86 procesore
    - x86_64 – Kompilirani kod samo za x86 64 procesore.
- *Dalvik bytecode* (class.dex) – Klase sastavljene u formatu .dex datoteke koje su razumljive Dalvik virtualnom stroju i Android Runtime-u.
- *Resources* (res) ili resursi – Direktorij koji sadrži resurse koji nisu prevedeni u resources.arsc [11].

4. iOS

4.1. Općenito

iOS je vlasnički mobilni operacijski sustav koji je razvijen od strane Apple-a i nalazi se isključivo u njihovim mobilnim uređajima iPod i iPhone. Također je osnova za ostale operacijske sustave koji proizvodi Apple a to su: iPadOS, watchOS i tvOS [12].

4.2. Povijest

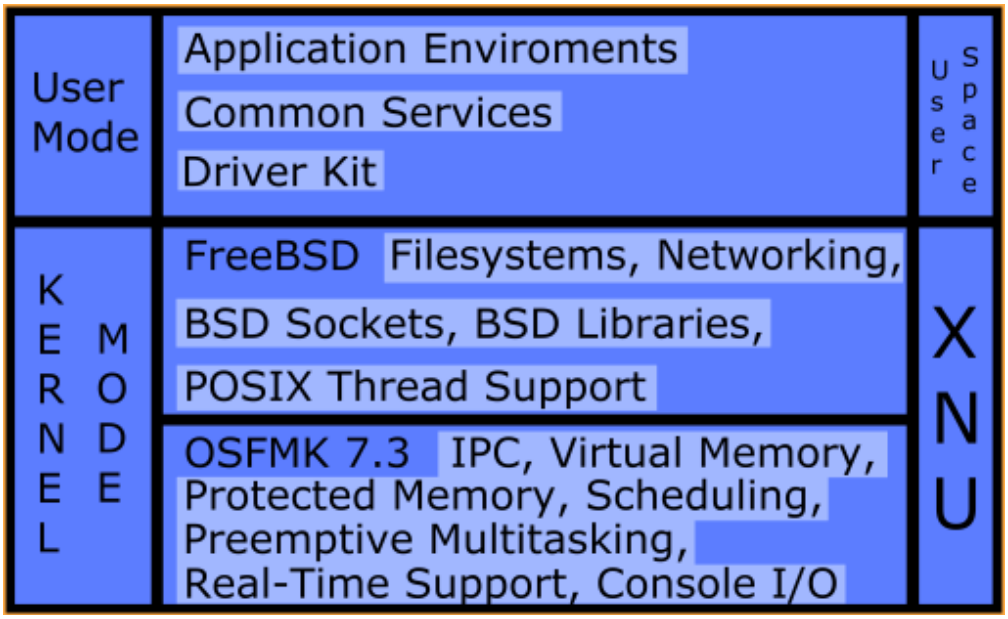
iOS je prvi put predstavljen svijetu 9. siječnja 2007 na „*Macworld conference and expo*“, a objavljen je u lipnju te godine. Prvotno nije bilo zamišljeno da mobilni sustav podržava *third-party* aplikacije već da se preko Safari – Appleovog mobilnog preglednika – koriste web aplikacije prilagođene za mobitele. Doduše, u listopadu iste godine najavljuje se razvoj iPhone SDK-a koji će omogućiti razvoj nativnih aplikacija.

U lipnju 2008. objavljen je App Store gdje se mogu preuzeti nativne aplikacije prilagođene za iOS sustav te ih je inicijalno izašlo 500. Aplikacije su se brzo razvijale, te je već do rujna bilo oko 3000 aplikacija, a već na početku 2009. u siječnju je bilo objavljeno 15 000 aplikacija. Taj trend se nastavio i do danas pa se trenutno na Apple App Store-u može naći oko 2.22 milijuna aplikacija [13].

4.3. Arhitektura

Osnovna arhitektura za iOS je ARM arhitektura. ARM arhitektura je akronim za Advanced RISC Machines što predstavlja obitelj reduciranih instrukcijskih naputaka za računanje. Zahvaljujući malom trošku, minimalnoj potrošnji energije i niskoj proizvodnjom topline, poželjni su kao arhitektura za lagane, portabilne uređaje koji rade na bateriju [14].

Prvotno su na iOS-u bili 32-bitni ARM procesori, a s iOS 7 se uvodi podrška za 64-bitne procesore. Od iOS 11, više se ne podržava 32-bitna arhitektura kao ni 32-bitne aplikacije te iOS postaje samo 64-bitni.



Slika 4.1. Vizualni prikaz XNU jezgre [15].

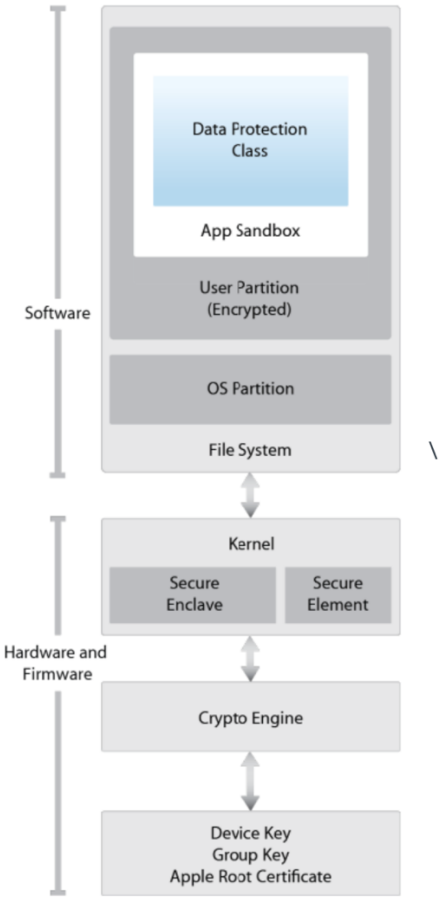
Jezgra koju koristi iOS zove se XNU Darwin od skraćenice X is Not Unix. XNU Darwin je hibridna jezgra koja kombinira komponente Mach i FreeBSD jezgara. Apple je razvija kao besplatan program otvorenoga koda koji je dio Darwin OS-a za macOS. Koristi se kao baza za ostale Apple operacijske sustave [16].

4.4. Sigurnosni alati

Apple ima rigorozna pravila za objavljivanje aplikacija te svaka aplikacija mora proći detaljan pregled s Appleove strane. Njihovi zaposlenici ručno pregledavaju svaki novi zahtjev za objavom na App Store. Svrha toga je da App Store bude siguran način za preuzimanje verificiranih aplikacija, dok *sideloading* aplikacije nije lako omogućen. U kontekstu iOS-a, *sideloading* je proces instaliranja aplikacija u IPA formatu koje nisu dostupne na App Store-u preko računala, internetskog preglednika ili korištenjem uređaja koji ima ovlasti instaliranja aplikacija iz nepouzdanih izvora. IPA format predstavlja arhivu iOS aplikacije, a format je detaljnije opisan u poglavlju 4.5. IPA – iOS App Store Package. Na novijim verzijama iOS-a, izvori aplikacija moraju vjerovati i Appleu i korisniku u "upravljanju profilima i uređajima" u postavkama. Apple ne dopušta bočno učitavanje osim za interno testiranje i razvoj aplikacija pomoću službenih SDK-ova. Osim toga, iOS koristi mnoge alate kako u hardverskom tako i u softverskom dijelu kako bi se zaštitio krajnji korisnik.

4.4.1 Secure Boot

Prvi korak hardverske sigurnosti je sigurno podizanje sustava ili *secure boot*, koji započinje u Boot ROM-u. To je dio *low-level* koda koji potvrđuje da je *bootloader* potpisan od strane Apple Root CA javnog ključa prije pokretanja iOS operacijskog sustava. Ovaj proces osigurava autentičnost *bootloader*-a koji se pokreće na uređaju. Ako je uspješno izvršeno, zadatke preuzima glavni *bootloader* zvan iBoot koji zatim pokreće iOS sustav. Ako proces nije uspješno prošao, uređaj ide u DFU ili *Device Firmware Upgrade* način rada koji ponovno učitava softver i *firmware* uređaja. Ukoliko se uređaju aktivan DFU način rada, potrebno je tvornički povratiti uređaj pošto postoji mogućnost od neispravnog/nesigurnog rada uređaja [17].

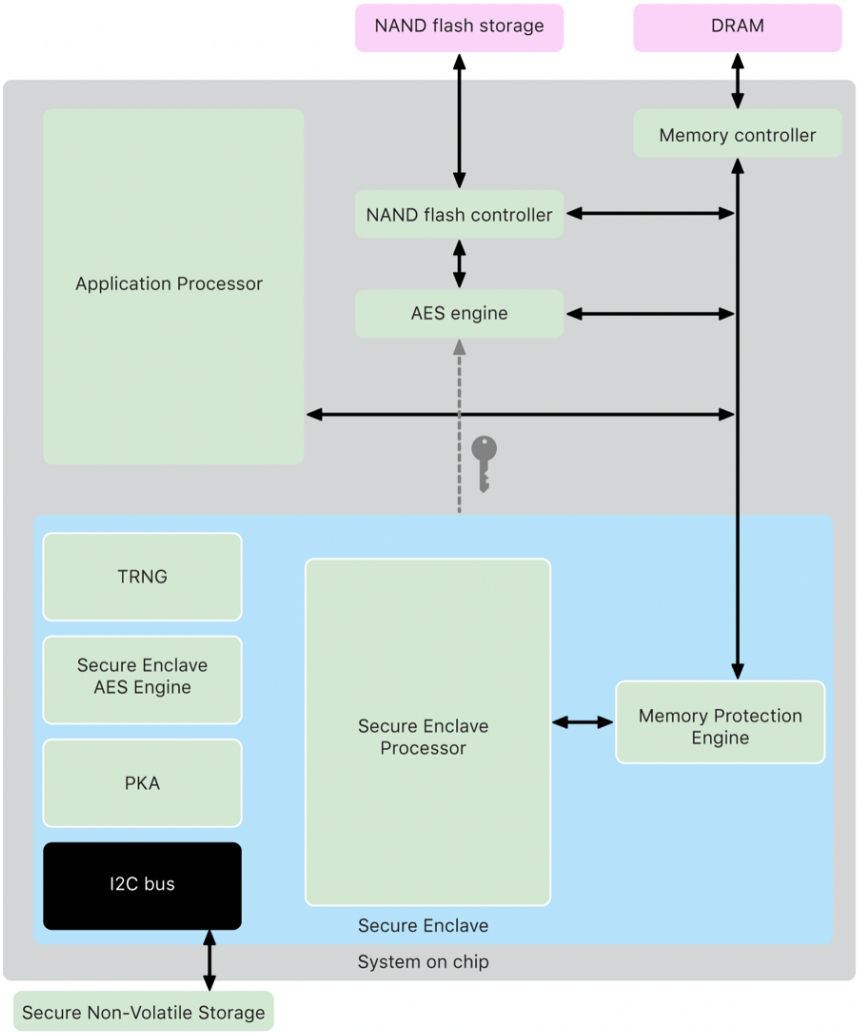


Slika 4.2. Vizualni prikaz softverskih i hardverskih sigurnosnih komponenti iOS operacijskog sustava [18].

4.4.2 Secure Enclave

Secure Enclave je zasebni procesor koji se nalazi u iOS uređajima koji omogućuje FaceID i TouchID funkcionalnosti. Sama svrha Secure Enclave-a je rukovanje ključevima i

drugim osjetljivim informacijama poput biometrije kako ih ne bi mogao obrađivati aplikacijski procesor, čak i u slučaju da aplikacijski procesor postane ugrožen. Koristi iste principe dizajna kao i SoC - ROM za pokretanje za uspostavljanje hardverskog „root of trust“, AES *engine* za učinkovite i sigurne kriptografske operacije i zaštićenu memoriju. Secure Enclave ne uključuje pohranu podataka već ima mehanizam za sigurno pohranjivanje informacija odvojenu od NAND *flash* memorije koju koriste procesor aplikacija i operacijski sustav [19].



Slika 4.3. Komponente Secure Enclave-a [19].

4.4.3 Sigurnost aplikacije

Aplikacije koje dolaze od trećih strana moraju biti potpisane od strane Apple certifikata. Time se dobiva lanac povjerenja sve od Secure Boot-a pa sve do aplikacija koje

korisnik ima instalirano na uređaju. Također, aplikacije su *sandboxed* što znači da nemaju pristup vanjskim direktorijima već samo vlastitima. Moguće je jedino pristupati podacima drugih aplikaciji uz eksplicitno dopuštenje. Daljnje mjere sigurnosti se primjenjuju i na pristup kameri, kontaktima, osvježavanje aplikacije u pozadini i ostalim podacima koje korisnik može prekinuti u svakom trenu u postavkama.

Većinom se aplikacija odvija u *mobile user* načinu rada koji nema *root* privilegije te se time osigurava da sistemske datoteke i drugi resursi sustava ostanu sakriveni i zaštićeni.

4.4.4 Mrežna sigurnost

Za sigurnu komunikaciju, iOS podržava TLS s API-ima niske i visoke razine za programere.

Koristi se TLS 1.2 koji zamjenjuje SSL protokol, ali programerima je omogućeno da implantiraju vlastite metode za komunikaciju preko mreže.

Dodatna zaštita koju iOS pruža je kada je korisnik spojen na Wi-Fi mrežu. Ako je Wi-Fi omogućen, iOS koristi nasumičnu MAC adresu tako da nitko tko „njuši“ bežični promet ne može pratiti korisnikov uređaj.

4.5. IPA – iOS App Store Package

Ekstenzija datoteke .ipa predstavlja arhivsku datoteku iOS aplikacije te se njome pohranjuje iOS aplikacija. To je format koji koristi isključivo Apple.

Svaka .ipa datoteka sadrži binarnu datoteku koja se može instalirati samo na iOS ili ARM MacOS uređajima. Datoteke s tim nastavkom se mogu dekomprimirati promjenom ekstenzije u .zip i raspakiranjem te datoteke.

```
/Payload/
/Payload/Application.app/
/iTunesArtwork
/iTunesArtwork@2x
/iTunesMetadata.plist
/WatchKitSupport/WK
/META-INF
```

Slika 4.4. Struktura .ipa datoteke.

Struktura .ipa datoteke ima ugrađenu strukturu za prepoznavanje na iTunes-u i App Store-u.

U Payload mapi se nalaze svi podaci. Datoteka iTunes Artwork je PNG slika veličine 512×512 piksela, koja sadrži ikonu aplikacije za prikazivanje u iTunesu i App Store-u. iTunesMetadata.plist sadrži različite bitove informacija, u rasponu od imena i ID-a programera, identifikatora paketa, informacija o autorskim pravima, žanra, naziva aplikacije, datuma izdanja, datuma kupnje itd.

Potpisivanje koda se nalazi u .app direktoriju.

Mapa META-INF sadrži samo metapodatke o tome koji je program korišten za stvaranje .ipa datoteke.

Kako bi napadači mogli instalirati aplikaciju s izmijenjenim potpisom koda, potrebno je izmijeniti .ipa datoteku tako da se kopiranjem mape s ekstenzijom .app iz mape Products aplikacije u Xcode-u u mapu pod nazivom Payload i komprimiranjem aplikacije pomoću naredbe *zip -0 -y -r myAppName.ipa Payload/*.

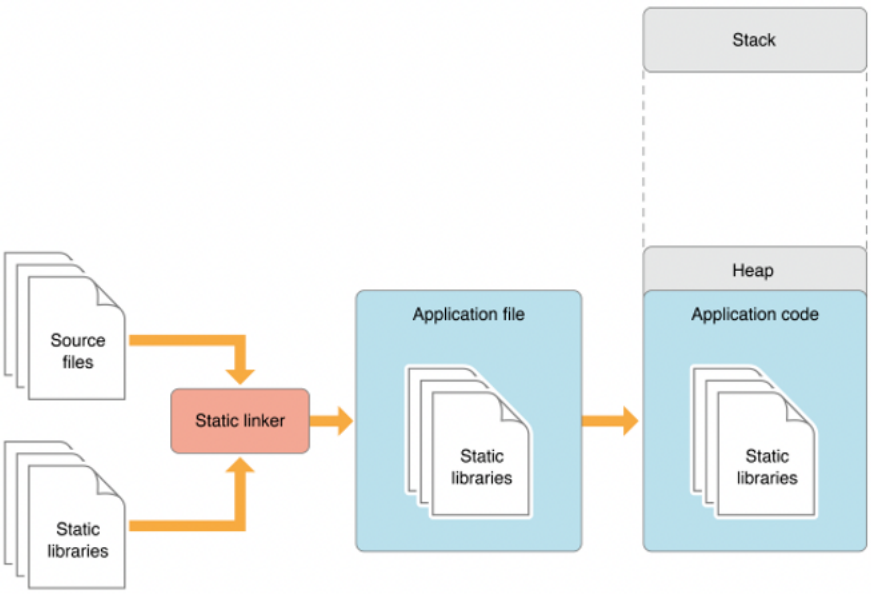
Postoje i razni drugi alati koji omogućuje ponovno potpisivanje aplikacije te s tim alatima izvršavati izmijenjenu aplikaciju. Na primjer, projekt *resign* koji je dostupan na GitHub-u omogućava ponovno potpisivanje aplikacije s dodavanjem novih dinamičkih knjižica unutar aplikacije koje omogućuju dinamičko instrumentiranje aplikacije [20].

4.6. Dinamičke i statičke knjižice

Aplikacije za iOS sustave mogu biti kompilirane sa statičkim ili dinamičkim knjižicama.

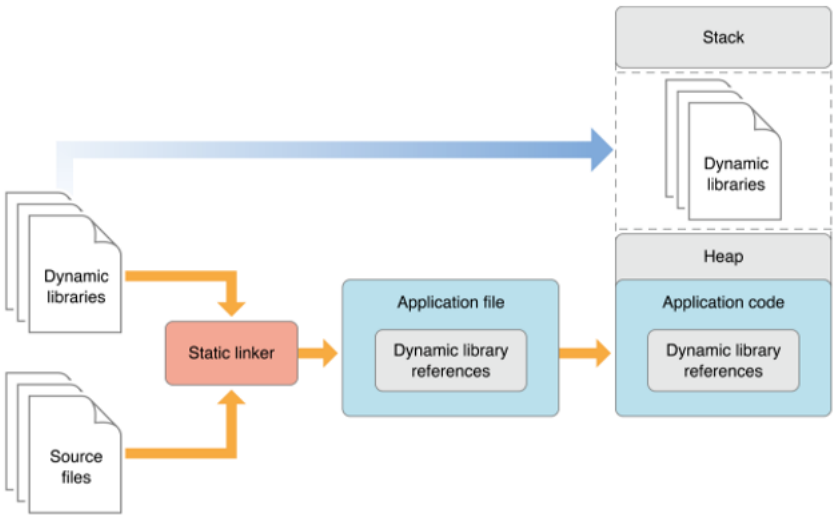
Kod statičkih knjižica, kod se učitava u aplikacijski adresni prostor prilikom kompiliranja. Rezultat toga je veća veličina aplikacije na disku i sporije učitavanje prilikom pokretanja.

Pošto je kod povezan direktno u izvrsnu binarnu datoteku, ako se želi promijeniti neki dio koda, potrebno je ponovno kompilirati aplikaciju.



Slika 4.5. Vizualni prikaz korištenja statičke knjižice u aplikaciji [21].

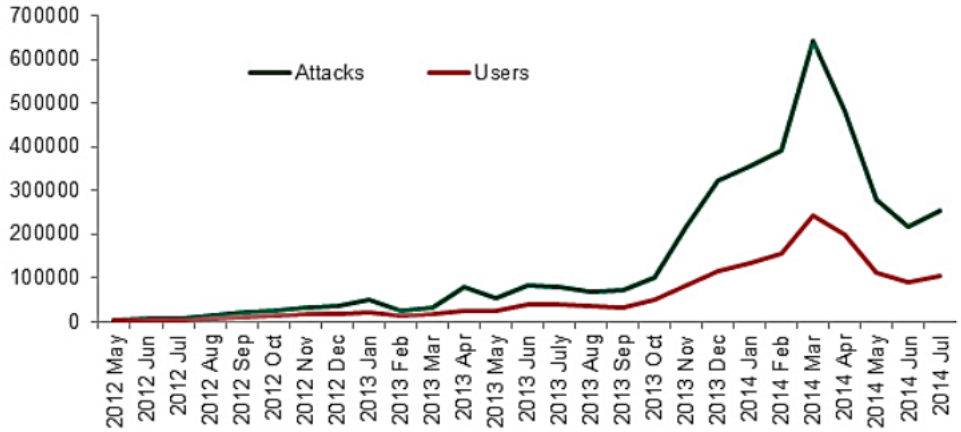
Kod dinamičkih knjižica, omogućeno je aplikaciji učitavanje koda u adresni prostor aplikacije kada je to potrebno prilikom izvođenja. Pošto kod nije statički povezan s izvršnom binarnom datotekom, knjižice se mogu ažurirati novi značajkama ili ispravcima bez potrebe za ponovnim kompiliranjem aplikacije [21].



Slika 4.6. Vizualni prikaz korištenja dinamičke knjižice u aplikaciji [21].

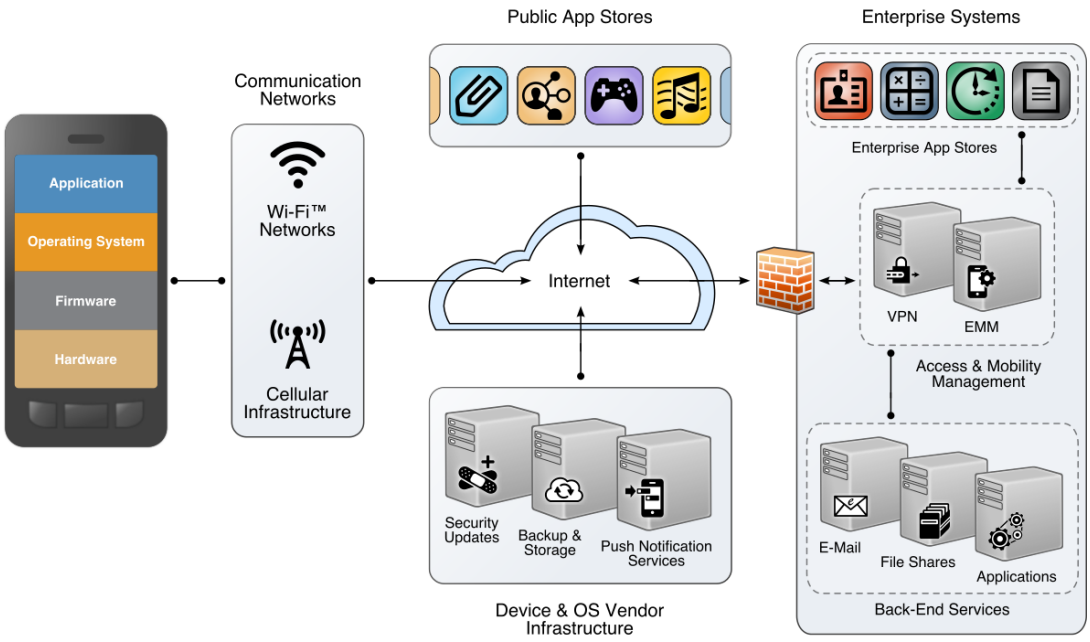
5. Pregled napada i mehanizama zaštite

Razvoj mobilnih sustava je iznimno napredovao zadnjih desetljeća te se time uvodi nova razina rizika. Kao što je već bilo navedeno u uvodu, ubrzanim razvojem hardverskih i softverskih komponenata mobilnih uređaja može doći do propusta u sigurnosnim procedurama sustava, internim kontrolama, dizajnu i aplikacijama. Kumulativno se ti propusti manifestiraju kroz napade na mobilne uređaje gdje napadači žele iskoristiti ranjivosti sustava [1]. Na Slici 5.1. prikazano je kako povećani broj korisnika utječe i na porast broja napada:



Slika 5.1. Prikaz grafa koji opisuje povezanost broja korisnika i napada na mobilne uređaje [1].

Mobilni uređaji imaju pristup mnogim vrijednim i osjetljivim osobnim podacima kao što su financijski podaci, vjerodajnice koje se koriste za pristup drugim resursima, kao i usluge u oblaku koje zaposlenici koriste za pristup podacima tvrtki kako bi mogli obavljati svoj posao [22].



Slika 5.2. Prikaz mobilnog ekosustava i razmjene podataka [23].

Gledajući cijeli ekosustav, potrebno je obratiti sve veću pozornost na sigurnost mobilnih uređaja s obzirom na sve veći broj mobilnih uređaja i aplikacija. Mobilni sustavi se smatraju sigurnijima od stolnih računala jer je otežano instaliranje nesigurnih programa. Unatoč tome, često zna doći do problema i kod razvoja mobilnih aplikacija. Prilikom razvoja mobilnih aplikacija potrebno je razmotriti na koji se način spremaju podaci, kako se razmjenjuju podaci preko mreže, ispravno korištenje kriptografskih API-ja i komunikacija s ostalim aplikacijama kako bi se smanjila mogućnost eksploatacije. Potrebno je predstaviti detaljan uvid u moguće napade kako bi se programerima olakšalo implementiranje sigurnosnih mjera i time zaštitilo krajnjeg korisnika.

5.1. Lokalno spremanje podataka

Jedno od ključnih elemenata na koje je potrebno obratiti pažnju je spremanje osjetljivih podataka na uređaju. Potrebno je obratiti pažnju na pravilno korištenje odgovarajućih API-ja za pohranu ključeva kako ne bi došlo do „curenja“ podataka ili korištenja podataka od strane drugih aplikacija. Također je potrebno upozoriti da ako dođe uređaj u tuđe ruke, ako podaci nisu kriptirani, lako je iščitati podatke iz memorije [24].

Određeni modeli uređaja podržavaju hardversko spremanje osjetljivih podataka poput Secure Enclave na iOS-u, dok na Androidu također postoji mehanizam hardverske zaštite koji se odnosi na hardverski podržan KeyStore.

Postoji više vrsta API-ja na svakoj platformi za spremanje podataka lokalno. Najčešće se za spremanje neosjetljivih podataka može koristiti SharedPreferences na Androidu ili NSUserDefaults na iOS-u. Omogućavaju spremanje *key-value* parova podataka i dohvaćanje bez potrebe za autentikacijom korisnika te se podaci spremaju u *sandbox* prostor aplikacije. Ti podaci nisu kriptirani te se upravo iz tog razloga treba izbjegavati pisati ikakve sigurnosne informacije korisnika.

5.1.1 Lokalno spremanje podataka na Androidu

Za pohranu podataka na Androidu može se koristiti unutarnja i vanjska pohrana. Ako je potrebno sigurno pohraniti podatke, ne preporučuje se korištenje vanjske pohrane jer tim podacima mogu pristupiti sve aplikacije. Sve što se nalazi u internoj pohrani u suštini je zatvoreno zbog aplikacijskog *sandbox*-a te se time ograničava među-aplikacijski pristup bez jasno definiranog dopuštenja.

Za sigurnu pohranu vjerodajnica koristi se Keystore. Keystore API je dostupan od Android 4.3 verzije te pruža API-je za pohranu i korištenje privatnih ključeva.

Noviji uređaji podržavaju implementaciju KeyStore-a koji je podržan hardverom, tj. izvršava se u sigurnom okruženju TEE ili sigurnom elementu SE, a operacijski sustav im ne može izravno pristupiti.

Ključevi koji su bazirani samo na softverskoj implementaciji, ako nije podržan hardverski KeyStore, su šifrirani glavnim ključem za šifriranje koji se odnosi na pojedinačnog korisnika. Za generiranje ključeva koristi se korisnikov pin ili lozinka za zaključavanje zaslona te KeyStore nije dostupan ako je uređaj zaključan.

Ako dođe do napada na uređaju koji je *root*-an (detaljnije o uređajima koji su *root*-ani u poglavlju 7.1.3), napadač može pristupiti svim ključevima koji su pohranjeni u mapi /data/misc/keystore/. Zbog toga je uvedeno od Android 9 (razina API-ja 28) oznaka *unlockedDeviceRequied* što onemogućuje dešifriranje pohranjenih ključeva ako se ne otključa zaslon.

Kako bi se osigurala dodatna razina zaštite za poslovno kritične operacije, uvodi se Key Attestation. Od Androida 8.0 (razina API-ja 26), potvrda ključa je obvezna za sve uređaje od Android 7 verzije te moraju imati certifikat uređaja za Google aplikacije.

Tijekom atestiranja ključa, određuje se alias par ključeva i povratno se dobiva lanac certifikata koji se koristi za provjeru svojstava tog para ključeva. Ako je korijenski certifikat

lanca korijenski certifikat Google Hardware Attestation i izvršene su provjere vezane za pohranu para ključeva u hardveru, to daje jamstvo da je uređaj siguran.

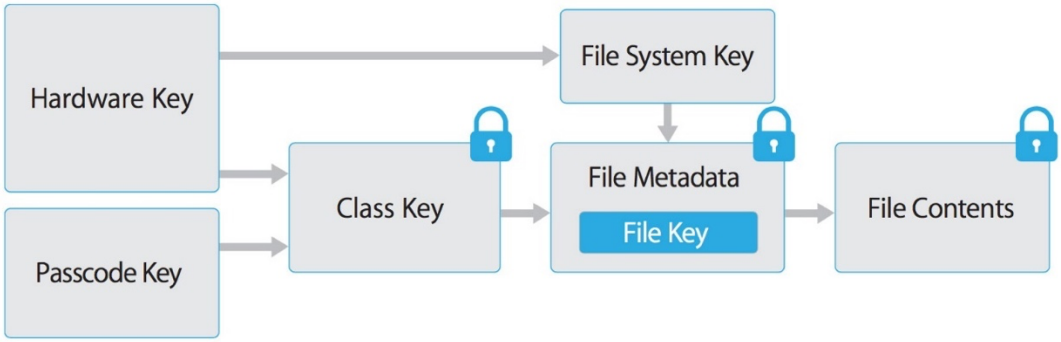
Preporučuje se da se iz sigurnosnih razloga atestiranje implementira na strani poslužitelja, iako se može implementirati aplikativno.

Za spremanje neosjetljivih *key-value* parova informacija koristi se SharedPreferences. S obzirom na to da ovi podaci nisu kriptirani, preporuča se izbjegavanje upisivanje osjetljivih podataka jer ih je lako iščitati [24].

5.1.2 Lokalno spremanje podataka na iOS-u

Kako bi se omogućilo sigurno lokalno spremanje podataka na iOS-u, preporučuje se korištenje iOS Data Protection API-ja. Ti API-ji se povezu sa Secure Enclave-om, zasebnim procesorom, koji omogućuje kriptografske operacije i zaštitu podataka. Unutar Secure Enclave-a nalazi se zapisan UUID – jedinstven ID uređaja, čime se osigurava integritet podataka iako operacijski sustav može biti ugrožen.

Prema Slici 5.3. može se vidjeti da je arhitektura zaštite podataka bazirana na hijerarhiji ključeva.



Slika 5.3. Arhitektura zaštite podataka na iOS-u.

Na vrhu arhitekture se nalaze UUID ili *Hardware Key* te *Passcode Key* koji je izveden iz korisničke šifre koristeći PBKDF2 algoritam. Pomoću njih se može pristupiti *Class Key*-evima koji su povezani sa stanjem uređaja, npr. definiraju stanje ako je uređaj otključan ili zaključan.

Datoteke se mogu dodijeliti jednoj od četiri različite klase zaštite:

- Potpuna zaštita (NSFileProtectionComplete) – Ključ je izveden iz korisničke šifre i UUID uređaja. Izvedeni ključ briše se iz memorije ubrzo nakon zaključavanja uređaja, čineći podacima nedostupnima sve dok korisnik ponovno ne otključa uređaj.

- Zaštićeno osim ako nije otvoreno (NSFileProtectionCompleteUnlessOpen) – Ova klasa zaštite slična je potpunoj zaštiti, ali ako se datoteka otvori kada je otključana, aplikacija može nastaviti pristupati datoteci čak i ako korisnik zaključa uređaj. Ova se klasa zaštite koristi kada se, na primjer, privitak pošte preuzima u pozadini.
- Zaštićeno do provjere autentičnosti prvog korisnika (NSFileProtectionCompleteUntilFirstUserAuthentication) – Datoteci se može pristupiti čim korisnik otključa uređaj prvi put nakon pokretanja. Može mu se pristupiti čak i ako korisnik naknadno zaključa uređaj i ključ klase nije uklonjen iz memorije.
- Bez zaštite (NSFileProtectionNone) – Ključ za ovu klasu zaštite zaštićen je samo UUID-om. Ključ klase pohranjen je u *flash* memoriji na uređaju. Koristi ga obično sistem za određene radnje pošto se ovoj datoteci uvijek može pristupati.

Za spremanje kratkih, osjetljivih podataka poput lozinka, koristi se Keychain. Keychain je implementiran kao SQLite baza podataka kojoj se može pristupiti samo preko Keychain API-ja.

Za razliku od macOS gdje svaka aplikacija može proizvesti proizvoljan broj „privjesaka“, na iOS-u postoji samo jedan Keychain koji je dostupan svim aplikacijama. Pristup stavkama može se dijeliti između aplikacija potpisanih od strane istog programera putem značajke pristupnih grupa atributa *kSecAttrAccessGroup*.

Pristup Keychain-u upravlja *securityd daemon*, koji odobrava pristup u skladu s ovlastima aplikacije koje su definirane poljima:

- *Keychain-access-groups*
- *application-identifier*
- *application-group*

Glavne operacije koje se omogućene u API-ju su:

- *SecItemAdd* – Dodavanje nove stavke.
- *SecItemUpdate* – Ažuriranje stavke.
- *SecItemCopyMatching* – Dohvaćanje stavke.
- *SecItemDelete* – Brisanje stavke.

Podaci pohranjeni u Keychain-u kodirane su kao binarna lista i šifrirane 128-bitnim AES ključem po stavci u načinu rada Galois/Counter (GCM).

Iako je to najsigurniji način za spremanje podataka, na primjer, ako netko dođe do fizičkog posjeda uređaja, korištenjem *keychain dumper*-a mogu se prikazati sve spremljene lozinke na uređaju.

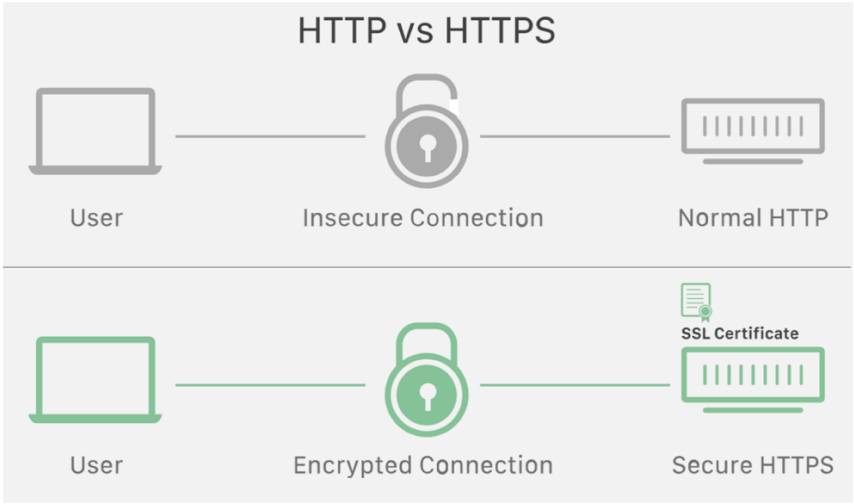
NSUserDefaults je primjer nesigurnog lokalnog spremanja podataka te se većinom koristi za spremanje korisnikovih referenci te prema tome mijenjati izgled ili ponašanje aplikacije. Podaci koje spremaju NSUserDefaults mogu se vidjeti u aplikacijskom paketu. Ova klasa pohranjuje podatke u .plist datoteku, te je namijenjena za korištenje s malim količinama podataka [24].

5.2. Komunikacija s pouzdanim krajnjim točkama

Komunikacija mobilnih uređaja s web servisima i raznim mrežama je neizbježna u današnje vrijeme. To može predstavljati prijetnju mobilnim uređajima od mrežnih napada ako nisu uspostavljene mjere zaštite. Jedan od ključnih mehanizama zaštite je uspostavljanje sigurnog, šifriranog kanala za komunikaciju koristeći TLS protokol s odgovarajućim postavkama [22].

TLS ili *Transport Layer* protokol je sigurnosni mrežni protokol, a uveden je kao poboljšanje SSL protokola.

SSL je skraćenica od *Secure Socket Layer*. To je standardna sigurnosna tehnologija za uspostavljanje šifrirane veze između klijenta i poslužitelja te ova veza osigurava privatnost poruka koje se izmjenjuju. SSL je ujedno najrašireniji kriptografski protokol za pružanje sigurnog komunikacijskog kanala. Slika 5.4. prikazuje dvije vrste konekcije, sigurnu i nesigurnu ako veza nema SSL certifikat.



Slika 5.4. Prikaz sigurne i nesigurne konekcije [25].

Prilikom stvaranja konekcije, postoji opasnost od presretanja veze ili *Man-in-the-Middle* napada, gdje postoji „netko“ tko prisluškuje i pregledava poruke koje se razmjenjuju između klijenta i poslužitelja.

SSL protokol osigurava sigurnost podataka tako da šifrira podatke koji se prenose. Ova tehnika proizlazi iz koncepta SSL certifikata i infrastrukture tijela za izdavanje certifikata. SSL certifikati imaju par ključeva: javni i privatni ključ. Ovi ključevi rade zajedno kako bi uspostavili šifriranu vezu. Certifikat također sadrži ono što se naziva "predmet", a to je identitet vlasnika certifikata/web stranice. Time se pokreće postupak provjere autentičnosti koji se naziva rukovanje između dva komunikacijska uređaja kako bi se osiguralo da su oba uređaja stvarno onakva za koju tvrde da jesu. SSL također digitalno potpisuje podatke kako bi osigurao integritet podataka, provjeravajući da podaci nisu neovlašteni prije nego što stignu do željenog poslužitelja ili primatelja.

SSL pričvršćivanje ili SSL Pinning je proces povezivanja hosta s njihovim očekivanim X509 certifikatom ili javnim ključem. Nakon što je certifikat ili javni ključ poznat ili vidljiv za hosta, certifikat ili javni ključ se povezuje ili 'prikvači' na hosta. To omogućuje aplikaciji da vjeruje samo valjanim ili unaprijed definiranim certifikatima ili javnim ključevima.

Ako osoba ima *jailbreak*-an uređaj može vrlo jednostavno zaobići SSL Pinning. *Jailbreak*-an uređaj omogućuje *root* pristup korisniku koji zatim dobiva potpune ovlasti nad operacijskim sustavom i programima. Instaliranjem *jailbreak*-a na uređaj, instalira se i Cydia aplikacija koja omogućuje preuzimanje i instaliranje neovlaštenih aplikacija, *tweak*-ova, mobilnih tema i slično.

Na iOS-u se koriste API pozivi iz knjižice Secure Transport. Jedan od načina kako zaobići je preuzimanje SSL Kill Switch paketa s Cydia aplikacije ili instaliranje tog *tweak*-a. *Tweak*-ovi su programski kodovi koji nakon instalacije mijenjaju ponašanje sistemskih metoda ili metoda određenih programa. SSL Kill Switch knjižica omogućuje prikačivanje na *low-level* metode koje definiraju koji će se *Certificate Pinning* koristiti. To znači da svaki pokušaj sigurnog spajanja na server će biti presretan i uspostavit će se nesigurna veza. Primjer koda koji mijenja originalnu funkciju je prikazan sljedećim kodom [26]:

```
static void replaced_SSL_CTX_set_custom_verify(void *ctx, int mode, ssl_verify_result_t
(*callback)(void *ssl, uint8_t *out_alert))
{
```

```
    original_SSL_CTX_set_custom_verify(ctx, SSL_VERIFY_NONE
verify_callback_that_does_not_validate);
    return;
}
void* boringssl_handle = dlopen("/usr/lib/libboringssl.dylib", RTLD_NOW);
void *SSL_CTX_set_custom_verify = dlsym(boringssl_handle,
"SSL_CTX_set_custom_verify");
if (SSL_CTX_set_custom_verify)
{
    MSHookFunction((void *) SSL_CTX_set_custom_verify, (void *)
replaced_SSL_CTX_set_custom_verify,  NULL);
}
char *replaced_SSL_get_psk_identity(void *ssl)
{
    return "notarealPSKidentity";
}
MSHookFunction((void *) SSL_get_psk_identity, (void *) replaced_SSL_get_psk_identity,
(void **) NULL);
```

MSHookFunction je ključna metoda iz CydiaSubstrate knjižice koja se prikvači na originalnu metodu te se umjesto pozivanja originalne metode, pozove nova, izmijenjena funkcija koja vraća vrijednost koju napadač želi.

Ova metoda se testirala i uspješno se svaki put zaobiđe SSL Pinning koristeći ovaj *tweak* iz Cydia aplikacije. U sekciji 7 je predložen način ublažavanja ove prijetnje tako da se MobileSubstrate.dylib stavi u popis malicioznih knjižica te detekcijom i reakcijom na pronalazak te knjižice može se zaustaviti i rad ovog *tweak*-a.

Korištenje tehnike pričvršćivanja SSL-a služi kao dodatni sigurnosni sloj za promet aplikacije i za provjeru identiteta udaljenog hosta. Ako SSL Pinning nije implementiran ili se *hook*-anjem ukloni, aplikacija vjeruje prilagođenom certifikatu i dopušta *proxy* alatima da presretnu promet.

5.3. Autentifikacija i autorizacija

Autentikacija i autorizacija korisnika na udaljenu uslugu sastavni je dio cjelokupne arhitekture mobilne aplikacije. Iako se većinom logika provjere autentičnosti odvija na krajnjoj točki, može se i odvijati na mobilnoj strani. Za bolje korisničko iskustvo, često se pohranjuju dugotrajne token sesije koje otključavaju s biometrijskim podacima.

Developer bi trebao biti svjestan da lokalnu provjeru autentičnosti uvijek treba provoditi na udaljenoj krajnjoj točki ili na temelju kriptografskog primitiva. Napadači mogu lako zaobići lokalnu autentifikaciju ako se dodatno ne provjerava autentičnost na nekoj krajnjoj točki [24].

5.4. Interakcija s mobilnom platformom

Arhitekture mobilnih operacijskih sustava razlikuju se od klasičnih *desktop* arhitektura tako što svi mobilni operacijski sustavi implementiraju sustave dopuštenja aplikacija koji reguliraju pristup određenim API-ima. Za razmjenu signala i podataka između aplikacije koristi se među-procesna komunikacija ili IPC. Problem može nastati ako se IPC API-i zlouporabe tako da osjetljivi podaci ili funkcionalnost mogu biti nenamjerno izloženi drugim aplikacijama koje se pokreću na uređaju.

Primjer toga je korištenje među-spremnika za kopiranje podataka. Među-spremnik je dostupan na cijelom sustavu i stoga ga dijele aplikacije. Ovo dijeljenje mogu iskoristiti zlonamjerne aplikacije za dobivanje osjetljivih podataka koji su pohranjeni u među-spremniku.

Primjer takvog napada je na iOS sustavima gdje je zlonamjerna aplikacija mogla nadzirati *pasteboard* u pozadini dok povremeno dohvaća tekst iz [UIPasteboard generalPasteboard].string. Od iOS-a 9, sadržaj *pasteboard*-a ploče dostupan je samo aplikacijama koje se izvršavaju u prvom planu, što smanjuje mogućnost napada njuškanja lozinke iz među-spremnika [22].

5.5. Kvaliteta koda i ublažavanje iskorištavanja

Prilikom razvoja aplikacija često se koriste javno dostupne knjižice koje omogućuju brži razvoj. Ako dođe do određenog *bug*-a u knjižici, može doći do eksploatacije aplikacije. Također se mogu pojaviti ostale ranjivosti poput SQL injekcije, prekoračenja među-spremnika i skriptiranja na više mjesta (XSS) te je potrebno obratiti pažnju na prakse

sigurnog programiranja i ako je moguće, razviti vlastite knjižice koje se ne oslanjaju na ostale, vanjske knjižice [24].

5.6. Obrnuti inženjering

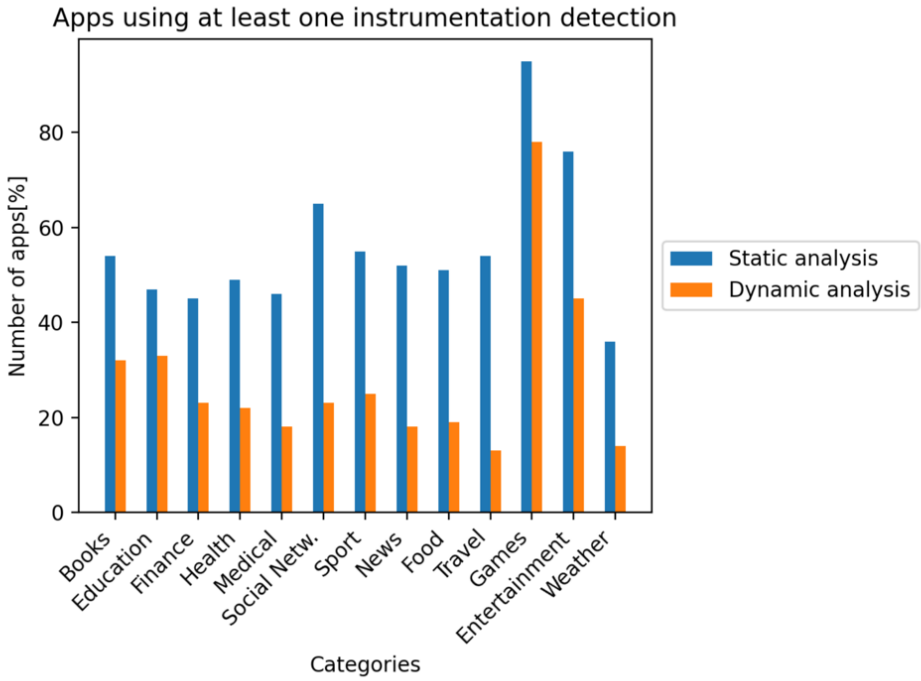
Obrnuti inženjering (eng. Reverse engineering) mobilne aplikacije je proces analize kompilirane aplikacije kako bi se izdvojile informacije o njezinom izvornom kodu. Cilj obrnutog inženjeringa je razumijevanje koda i što se događa u pozadini aplikacije.

Obrnuti inženjering se često povezuju s pojmom *application tampering*. Pojam *application tampering* predstavlja izmjenjivanje ponašanja aplikacije. Obrnutim inženjeringom mogu se saznaju pozadinski procesi aplikacije i zatim izmijeniti originalno ponašanje aplikacije kako bi ponašanje prilagodili svojim potrebama. Na primjer, ukloniti iz aplikacije ugrađene procese za zaštitu od *root*-anih ili *jailbreak*-anih uređaja kako bi napadač olakšano mogao analizirati aplikaciju.

Najpopularniji alat danas koji se koristi u tu svrhu je Frida. Frida alati omogućuju pregledavanje funkcija koje se izvršavaju u aplikaciji u realnom vremenu, ispis svih metoda, pozivanje skripti koje izmjenjuju ponašanje funkcije te praćenje određenih metoda da se sazna u kojim se sve trenucima izvode.

U današnje vrijeme, jedna od najvećih prijetnji je izmjenjivanje aplikacije te je potrebno implementirati zaštitu od alata koji pomažu u dinamičkom instrumentiranju aplikacije.

Pregledom 1300 aplikacija iz AppStore-a, dobilo se prikaz koliko zapravo aplikacija sadrži detekciju instrumentacije aplikacije. Na Slici 5.5. vidi se da većinom oko 30% aplikacija po sektoru sadrži takvu detekciju, dok aplikacije iz područja igara i zabave češće detektiraju takve prijetnje [27].



Slika 5.5. Grafički prikaz postotak aplikacija koje sadrže detekciju instrumentiranja [27].

6. Testiranje sigurnosti aplikacije

Testiranje aplikacije se može koristiti prilikom razvoja aplikacije kako bi se otkrilo gdje je potrebno pojačati sigurnost aplikacije. Ovim pregledom se može dobiti uvid kako bi maliciozna osoba mogla analizirati aplikaciju.

6.1. Analiza ranjivosti aplikacije

Analiza ranjivosti aplikacije se može proučavati statičkom i dinamičkom analizom.

Statička analiza uključuje analizu aplikacijskih komponenata bez njihovog izvršavanja, poput manualne ili automatske analize izvršnog koda.

Dinamička analiza uključuje analizu aplikacije prilikom njenog izvršavanja u realnom vremenu. Ova analiza također može biti manualna ili automatska.

Primarna prednost statičke analize je u tome što se ispituju svi mogući putevi izvršenja i vrijednosti varijabli, a ne samo one koje se pozivaju tijekom izvršavanja. Stoga, statička analiza može otkriti pogreške koje se možda neće prikazati do tek nakon objavljivanja aplikacije i njenog opsežnog korištenja.

6.2. Statička analiza

Statička manualna analiza koda uključuje analizu aplikacijskog izvršnog koda. Postoje različite metode poput pretraživanja ključne rijeci koristeći metodu *grep* u terminalu kako bi dohvatili svako pojavljivanje te riječi ili pregledanje liniju po liniju koda.

Česti pristup u takvoj analizi je pronalaženje ključnih indikatora ranjivosti poput određenih API poziva ili ključnih riječi, poput sistemskih poziva za detekciju *debugger*-a.

Manualna analiza može biti dugačka i oduzimati puno vremena, ali je vrlo dobra za nalaženje pogrešaka u poslovnoj logici aplikacije, kršenja standarda i nedostatka u dizajnu.

Kako bi se ubrzao proces statičke manualne analize, može se koristiti automatizirana analiza koda. Provjerava se izvorni kod ako se slaže s redefiniranim setom pravila ili najboljih praksa.

6.3. Dinamička analiza

Glavni fokus dinamičke analize je evaluacija aplikacije u njenom izvođenju u realnom vremenu kako bi se pronašle ranjivosti prilikom izvođenja.

Prilikom analize, testira se i sloj mobilne platforme te API pozivi.

Svrha testiranja je da se otkriju najčešće pogreške u prijenosu podataka, problemi s autentifikacijom i autorizacijom ili pogreške u konfiguraciji poslužitelja.

6.4. Penetration testing ili Pentest

Pentest je proces koji se radi u završnoj fazi aplikacije kako bi iskusni testeri mogli u vremenu izvođenja aplikacije pronaći slabosti u aplikaciji. Firme najčešće daju aplikacije na *pentesting* kako bi dobile povratne informacije kako unaprijediti neke funkcionalnosti prije objavljivanja aplikacije.

Tipična struktura testiranja odvija se u sljedećim fazama:

- Priprema – Potrebno je definirati opseg testiranja što uključuje određivanje koje će se sigurnosne kontrole provjeravati, koji su ciljevi testiranja te na koji način se razmjenjuju podaci. Također je potrebna pravna zaštita testera jer je nezakonito na ovaj način testirati aplikaciju bez pismenog odobrenja od strane klijenta.
- Prikupljanje obavještajnih podataka – Potrebno je analizirati okruženje arhitekture aplikacije kako bi se steklo generalno razumijevanje okoline koja će se testirati.
- Mapiranje aplikacije – Ovisno o zahtjevima, može se mapirati aplikacija automatskim skeniranjem ili ručnim istraživanjem aplikacije. Mapiranje pruža detaljno razumijevanje aplikacijskih ulazni točaka, podataka koje posjeduje i pregled potencijalnih ranjivosti. Zatim se ti podaci mogu rangirati prema šteti koju bi mogle prouzročiti ako dođe do eksploatacije aplikacije te rangirati po prioritetu kako bi se znalo što treba napraviti za poboljšanje sigurnosti aplikacije.
- Eksploatacija – Faza u kojoj tester pokušava iskoristiti prijašnje utemeljene ranjivosti. U ovoj fazi se otkriva ako su ranjivosti stvarne i kojoj mjeri stvaraju štetu.
- Izvještavanje – Proces u kojem se prijavljuju ranjivosti koje su detaljno opisane u dokumentu koji se zatim dostavlja klijentu.

6.4.1 Priprema

Sigurnosna razina i zahtjevi koje je potrebno testirati potrebno je definirati na početku projekta. Različite organizacije imaju različite sigurnosne potrebe i raspoložive resurse za ulaganje u testne aktivnosti. Ovisno o regulatornim i zakonskim obavezama na određenim teritorijima, organizacije mogu imati različite zahtjeve. Na primjer, provjera autentičnosti s dva faktora (2FA) može biti obvezna za financijsku aplikaciju i provodi ju središnja banka zemlje i/ili financijska regulatorna tijela.

6.4.2 Identificiranje osjetljivih podataka

Klasifikacija podataka se može razlikovati po industriji ili državi. Ovisno o tome ako firma radi s banka, potrebna je dodatna razina sigurnosti aplikacija za razliku od drugih aplikacija gdje se ne spremaju osjetljivi podaci poput kreditnih kartica.

Postoje tri generalna stanja u kojima podaci mogu biti dostupni:

- U mirovanju – Podatak je spremljen u datoteci ili spremniku podataka.
- U korištenju – Podaci su učitani u adresni prostor aplikacije.
- U prijenosu – Podaci se razmjenjuju između aplikacije i određene udaljene krajnje točke ili u nekom procesu na uređaju.

Podaci koji se drže u memoriji aplikacije mogu biti ranjiviji od podataka na web poslužiteljima jer može biti vjerojatnije da će napadači dobiti fizički pristup mobilnim uređajima nego web poslužiteljima.

Ako nema klasifikacije važnosti određenih podataka, generalno važni podaci na koje se tester može osvrnuti su:

- Podaci za autentifikaciju korisnika kao što je na primjer PIN
- Podaci koji otkrivaju identitet te se mogu zloupotrijebiti za krađu identiteta: OIB, brojevi kreditnih kartica, brojevi bankovnih računa, zdravstveni podaci
- Identifikatori uređaja koji mogu identificirati osobu
- Vrlo osjetljivi podaci čiji bi kompromis doveo do povrede reputacije i/ili financijskih troškova
- Sve podatke čija je zaštita zakonska obveza
- Sve tehničke podatke koje generira aplikacija (ili srodni sustavi) i koji se koriste za zaštitu drugih podataka ili samog sustava (npr. ključevi za šifriranje)

6.4.3 Prikupljanje obavještajnih podataka

Prikupljanje obavještajnih podataka uključuje prikupljanje informacija o arhitekturi aplikacije, *use-cases* koje aplikacija služi i kontekstu u kojem aplikacija djeluje. Takve informacije se mogu klasificirati ako podaci o okolini te podaci o arhitekturi aplikacije.

Potrebno je prikupiti i informacije o arhitekturi, što uključuje:

- Mobilna aplikacija – Potrebno je definirati kako aplikacija pristupa i upravlja podacima, kako komunicira s različitim resursima te ako aplikacija detektira da se izvodi na *jailbreak*-nim ili *root*-anim uređajima i kako reagira na takve situacije.

- Operacijski sustav – Definira se na kojim verzijama operacijskog sustava aplikacija radi i koje su relevantne ranjivosti sustava.
- Mreža – Definira se ako se upotrebljava sigurni prijenos podataka poput TLS-a, koriste li se algoritmi i ključevi za sigurnosno šifriranje mrežnog prometa ili *certificate pinning.*
- Udaljeni servisi - Potrebno je istražiti koje udaljene usluge aplikacija koristi i ako mogu kompromitirati klijenta.

6.4.4 Mapiranje aplikacije

Nakon prikupljanja podataka o aplikaciji i njenom kontekstu, sljedeći je korak identificiranje njenih ulaznih točaka, značajki i podataka.

Ako se provodi penetracijsko testiranje, korištenje dijagrama ili specifikacije koje je firma ustupila na korištenje, može biti od velike pomoći za mapiranje aplikacije. Time se ubrzava proces analize te će testeru biti potrebno manje vremena.

Ako maliciozna osoba želi mapirati aplikaciju, alat kojim se može poslužiti je Hopper za iOS ili Apktool za Android.

Hopper je alat koji omogućuje rastavljanje, dekompiliranje i debuggiranje aplikacije gdje se korištenjem .ipa datoteke može saznati koje procedure aplikacija ima, koji je logički tok izvršavanja funkcija ili saznati gdje se pojavljuju ključne riječi i slično.

Apktool je alat za obrnuti inženjering, zatvorene, binarne Android aplikacije treće strane. S ovim alatom se mogu dekodirati resursi u gotovo originalni oblik i ponovno ih izgraditi nakon nekih izmjena.

6.4.5 Eksploatacija

Relevantnost ranjivosti i njihova mogućnost eksploatacije je potrebno rangirati s obzirom na:

- Potencijal štete - Šteta koja može proizaći iz iskorištavanja ranjivosti.
- Reprodukcija – Lakoća reprodukcije napada.
- Iskorištavanje – Jednostavnost izvođenja napada.
- Pogođeni korisnici - Broj korisnika pogođenih napadom.
- Uočljivost – Lakoća otkrivanja ranjivosti.

6.4.6 Izvještavanje

Nalazi testera sigurnosti potrebno je jasno dokumentirani kako bi klijent imao koristi od takvog dokumenta. Klijenta se izvještava o pronađenim ranjivostima, metodama koje su bile korištene te se definiraju načini na koje bi se mogle ublažiti ranjivosti [24].

7. Primjeri napada i njihovih ublažavanja

Kako bi se aplikacija zaštitila od mogućih napada, potrebno je unutar aplikacije implementirati metode koje detektiraju moguće prijetnje. Te metode se razlikuju između platforma, ali postoje načini kako ih detektirati. Potrebno je uzeti u obzir da napadač zna da je poznato koji su načini detekcije te je potrebno na što više različitih način prepoznati prijetnju kako bi se otežalo napadaču.

Najčešći oblici napada su:

- Preko debuggera
- *Hook* metoda
- *Jailbreak*/*root*
- Simulator

Mogući načini prevencije:

- Obfuskacija koda
- Provjera integriteta aplikacije
- Povezivanje uređaja
- Zaštita u vremenu izvođenja aplikacije gdje se periodično provjera ako je aplikacija ugrožena.

U nastavku će se detaljno objasniti svaki aspekt napada, način implementacije te njegove prednosti i mane.

7.1. Moguće detekcije i načini zaštite na Android platformi

7.1.1 Debuggiranje

Kao i na iOS-u, postoji više načina kako detektirati ili onemogućiti spajanje debuggera na aplikaciju.

Jedan od načina kako onesposobiti da se dopusti spajanje debuggera je deklarirati atribut `android:debuggable="false"` u AndroidManifest.xml datoteci. Iako je ovo dobra praksa zabrane debuggera, vrlo je jednostavno napadaču izmijeniti ovaj atribut u navedenoj datoteci. Zato je poželjno dodatno detektirati u procesu izvođenja ako je spojen debugger na proces.

Sljedeći kod se može koristiti unutar aplikacije kako bi se detektiralo proces debuggiranja:

```
ApplicationInfo appInfo =
context.getPackageManager().getApplicationInfo("com.nettitude.labs.app", 0)
if((appInfo.flags && appInfo.FLAG_DEBUGGABLE) != 0)
// aplikaciju se može debuggirati
```

Također se uz ovu metodu može dodatno periodično provjeravati prisutnost debuggera koristeći metodu i*sDebuggerConnected()* iz klase *android.os.Debug* te također *System.Diagnostics.Debugger.IsAttached* metodu [29].

Dodatno se može provjeravati koliko dugo se trenutni *thread* izvršava posto debugger produljuje vrijeme izvođenja procesa. Koristeći *Debug.threadCpuTimeNanos* može se izračunati razlika u izvršavanju:

```
static boolean detect_threadCpuTimeNanos(){
  long start = Debug.threadCpuTimeNanos();
  for(int i=0; i<1000000; ++i)
    continue;
  long stop = Debug.threadCpuTimeNanos();
  if(stop - start < 10000000) {
    return false;
  }
  else {
    return true;
  }
}
```

Problem kod određenih knjižica je u tome da se može prikvačiti na njih te vratiti željenu vrijednost. Stoga treba obratiti pažnju da se periodički provjerava prisutnost debuggera i da se detekcija nalazi na više mjesta kroz kod te se s time povećava uspješnost detekcije.

7.1.2 Hooking

Izmijene ponašanja metoda u vremenu izvođenja ili *hooking* najčešće se odvija koristeći Cydia Substrate i Xposed framework-e te frida-alate.

Jedna od detekcija je pronalazak malicioznih paketa unutar aplikacije. Koristeći *PackageManager*, može se dohvatiti lista instaliranih paketa te ako se naiđe na paket s liste nepoželjnih paketa, aplikacija može reagirati na detekciju.

```
PackageManager pm = context.getPackageManager();
List appInfoList = pm.getInstalledApplications(PackageManager.GET_META_DATA);
for(ApplicationInfo appInfo : appInfoList) {
   if(appInfo.packageName.equals("de.robv.android.xposed.installer"))
      // Xposed framework found!
   if(appInfo.packageName.equals("com.saurik.substrate"))
      // Cydia Substrate found!;
if(appInfo.packageName.equals("frida "))
      // Cydia Substrate found!;
}
```

Sljedeća detekcija sastoji se od nalaženja sumnjivih putanja ili artefakta koji su povezani sa sumnjivim knjižicama. Unutar Linux-a, */proc/{pid}/maps* datoteka predstavlja trenutno mapirana memorijska područja i njihove dozvole pristupa [37].

Pregledavanjem datoteka mogu se pronaći sumnjive putanje:

- Cydia Substrate:
  - */data/app-lib/com.saurik.substrate-1/libAndroidBootstrap0.so*
  - */data/app-lib/com.saurik.substrate-1/libAndroidCydia.cy.so*
  - */data/app-lib/com.saurik.substrate-1/libDalvikLoader.cy.so*
  - */data/app-lib/com.saurik.substrate-1/libsubstrate.so*
  - */data/app-lib/com.saurik.substrate-1/libsubstrate-dvm.so*
  - */data/app-lib/com.saurik.substrate-1/libAndroidLoader.so*
- Xposed

  /data/data/de.robv.android.xposed.installer/bin/XposedBridge.jar

Sljedeći je isječak koda za dokaz koncepta koji koristi ovu tehniku [30]:

```
Set libs = new HashSet();
   String mapFile = "/proc/" + android.os.Process.myPid() + "/maps";
   BufferedReader reader = new BufferedReader(new FileReader(mapFile));
   String line;
   while((line = reader.readLine()) != null) {
      if (line.endsWith(".so") || line.endsWith(".jar")) {
         int n = line.lastIndexOf(" ");
         libs.add(line.substring(n + 1));
      }
   }
   for (String library : libraries) {
      if(library.contains("com.saurik.substrate"))
         // Cydia Substrate library found!
      if(library.contains("XposedBridge.jar"))
         //"Xposed framework's JAR found!
   }
   reader.close();
```

Daljnja detekcija Xpose i Cydia frameworka može biti u analizi *stack trace*-a. Ako je Xposed framework aktivan, *metoda robv.android.xposed.XposedBridge.main* će se prikazati u *stack trace*-u nakon *dalvik.system.NativeStart.main* metode te ako se Xposed prikvači na određenu metodu, u *stack trace*-u će se prikazati poziv prema *robv.android.xposed.XposedBridge.handleHookedMethod* i *de.robv.android.xposed.XposedBridge.invokeOriginalMethodNative* metodama.

Slično ponašanje se događa i kod Cydia Subtrate knjižice. Ako je aktivna, nakon *dalvik.system.NativeStart.main* metode bit će prisutna i *android.internal.os.ZygoteInit.main* metoda, a ako se prikvači na metodu, bit će prisutna metoda *saurik.substrate.MS$2.invoked*, *com.saurik.substrate.MS$MethodPointer.invoke* and i treća metoda koja je Substrate ekstenzija koja može varirati ovisno o vrsti *hook*-a.

Sljedeći kod je *proof-of-concept* metoda za detekciju knjižica u *stack-trace*-u:

```
try {
   possiblyHookedMethod("Nettitude Labs");
}
```

```
catch(Exception e) {
   int zygoteInitCallCount = 0;
   for(StackTraceElement stackTraceElement : e.getStackTrace()) {
      if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit")) {
         zygoteInitCallCount++;
         if(zygoteInitCallCount == 2)
            // Cydia Substrate in use
      }
      if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
       stackTraceElement.getMethodName().equals("invoked"))
        // some method has been hooked using Cydia Substrate
      if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge")
&&
       stackTraceElement.getMethodName().equals("main"))
        // Xposed framework in use

      if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge")
&&
       stackTraceElement.getMethodName().equals("handleHookedMethod"))
        // some method has been hooked using Xposed
   }
}
```

Dodatno je potrebno detektirati Frida alate. Frida-gadget i frida-agent ostavljaju artefakt za sobom u obliku stringa „LIBFRIDA“ te se iteracijom kroz memorijska područja može pretražiti ako postoji taj artefakt. Artefakti mogu biti datoteke paketa, binarne datoteke, biblioteke, procesi i privremene datoteke.

Ugrađena frida-gadget knjižica u aplikaciju može se detektirati tako da se provjeri APK *signing signature* s obzirom na to da je potrebno prepakirati APK. Ovo se može lako zaobići te se ne preporučuje kao detekcija.

Dodatna detekcija frida-server-a je kao i na iOS-u je da se pretraže otvoreni TCP portovi ako na nekom od njih sluša Frida. Prednost u ovoj detekciji naspram iste na iOS-u je u tome što se Android dopusta dohvaćanje svih otvorenih TCP konekcija. Ovo je jedna od

robusnijih metoda za detekciju frida-server-a, ali se može zaobići koristeći druge frida alate. Stoga je važno da postoji više različitih metoda detekcije Fride kako bi se uspješno detektirala [24, 30].

7.1.3 Root

Root-anjem uređaja korisnik dobiva *root* prava te se time uklanjaju brojne restrikcije koje omogućuju sigurnost sistema poput *sandbox*-a, mijenjanje sistemskih datoteka, aplikacija ili postavki i slično.

Android API SafetyNet nudi mogućnosti detekcije *root*-anog uređaja. SafetyNet pruža različite usluge te ujedno stvara profile uređaja prema informacijama o softveru i hardveru koji se zatim uspoređuje s popisom prihvaćenih modela uređaja koji su prošli testiranje kompatibilnosti s Androidom. Za korištenje API-ja koristi se *SafetyNet.attest* metoda koja vraća JWS poruku s rezultatom testiranja [39].

```
{
  "timestampMs": 9860437986543,
   "nonce": "R2Rra24fVm5xa2Mg",
  "apkPackageName": "com.package.name.of.requesting.app",
  "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
                                  certificate used to sign requesting app"],
  "ctsProfileMatch": true,
  "basicIntegrity": true,
  "evaluationType": "BASIC",
}
```

Slika 7.1. JWS poruka koju vraća SafetyNet.attest metoda.

Rezultat potvrde obično sadrži sljedeća polja:

- *timestampMS* – Vrijeme u kojem je generirana poruka na Google serverima. Računa se u milisekundama nakon Unix epohe.
- *nonce* – Token za jednokratnu upotrebu koji aplikacija prosljeđuje API-ju.
- *apkPackageName* – Naziv paketa aplikacije koja poziva API.
- *apkCertificateDigestSha256* - Base-64 kodirani prikaz(i) SHA-256 hash certifikata(a) za potpisivanje aplikacije.
- *ctsProfileMatch* – Stroža presuda o integritetu uređaja. Ako je vrijednost *ctsProfileMatch* postavljena na *true*, profil uređaja na kojem se izvodi aplikacija podudara se s profilom uređaja koji je prošao testiranje kompatibilnosti s Androidom i koji je odobren kao Android uređaj s Googleovim certifikatom.

- *basicIntegrity* – Blaža presuda o integritetu uređaja. Ako je samo vrijednost *basicIntegrity* postavljena na *true*, tada uređaj na kojem je pokrenuta vaša aplikacija vjerojatno nije neovlašteno mijenjan. Međutim, uređaj nije nužno prošao testiranje kompatibilnosti s Androidom.
- Opcionalna polja:
    - *error* – Kodirane informacije o pogrešci relevantne za trenutni API zahtjev.
    - *advice* – Prijedlog kako uređaj vratiti u dobro stanje.
    - *evaluationType* – Vrste mjerenja koje su doprinijele trenutnom API odgovoru.

SafetyNet nije dovoljno dokumentiran te nije jasno na koji način se detektira *root*-an uređaj. Unatoč tome, Google preporučuje korištenje ovog API-ja kao dodatan mehanizam u obrani aplikacije [39].

Sljedeća detekcija bazirana je provjeri postojanju određenih datoteka koje se mogu na *root*-anim uređajima poput:

- */system/app/Superuser.apk*
- */system/etc/init.d/99SuperSUDaemon*
- */dev/com.koushikdutta.superuser.daemon/*
- */system/xbin/daemonsu*
- */sbin/su*
- */system/bin/su*
- */system/bin/failsafe/su*
- */system/xbin/su*
- */system/xbin/busybox*
- */system/sd/xbin/su*
- */data/local/su*
- */data/local/xbin/su*
- */data/local/bin/su*

Također se može provjeriti ako se *su* nalazi kao Environment varijabla u PATH-u:

```
public static boolean checkRoot(){
    for(String pathDir : System.getenv("PATH").split(":")){
        if(new File(pathDir, "su").exists()) {
            return true;
        }
```

```
    }
    return false;
  }
```

Drugi način provjere prisutnosti *su* naredbe je pokušaj izvršavanja naredbe koristeći *Runtime.getRuntime.exec* metode. Ako *su* komanda nije prisutna u PATH-u, dogodit će se *IOException exception*.

Dodatno se mogu provjeriti procesi koji se izvode. Jedan od popularnih *rooting* alata je *daemonsu* koji se može provjeriti sljedećim kodom koji provjerava pokrenute procese na uređaju:

```
public boolean checkRunningProcesses() {
    boolean returnValue = false;
    List<RunningServiceInfo> list = manager.getRunningServices(300);
    if(list != null){
      String tempName;
      for(int i=0;i<list.size();++i){
        tempName = list.get(i).process;
        if(tempName.contains("supersu") || tempName.contains("superuser")){
          returnValue = true;
        }
      }
    }
    return returnValue;
  }
```

Detekcija instaliranih paketa unutar aplikacije također se koristi kao mehanizam obrane.

Dohvaćanjem liste instaliranih paketa, provjera se postojanje sljedećih malicioznih paketa koja često pripadaju *rooting* alatima:

- *com.thirdparty.superuser*
- *eu.chainfire.supersu*
- *com.noshufou.android.su*
- *com.koushikdutta.superuser*

- *com.zachspong.temprootremovejb*
- *com.ramdroid.appquarantine*
- *com.topjohnwu.magisk*

Ako je uređaj *root*-an može se provjeriti dopuštenja pisanja u sistemskim direktorijima, što nije dopušteno u uređajima koji nisu *root*-ani.

Provjera znakova testnih verzija i prilagođenih ROM-ova također može biti način detekcije. Provjerava se način da se provjeri oznaka BUILD za testne ključeve, koji obično označavaju prilagođenu Android sliku:

```
private boolean isTestKeyBuild()
{
String str = Build.TAGS;
if ((str != null) && (str.contains("test-keys")));
for (int i = 1; ; i = 0)
  return i;
}
```

Također, nedostatak Google Over-The-Air (OTA) certifikata još je jedan znak prilagođenog ROM-a, s obzirom na to da OTA ažurira Googleove javne certifikate.

Ove metode se mogu zaobići na sljedeće načine:

- Preimenovanjem binarnih datoteka.
- Demontiranje /proc kako bi se spriječilo čitanje popisa procesa.
- Korištenjem Frida ili Xposed. Korištenjem ovih alata mogu se presretati metode koje provjeravaju ako je uređaj *root*-an i vratiti lažne vrijednosti.
- Zakrpa aplikacije za uklanjanje provjera.

Iako se mogu zaobići metode, poželjno ih je implementirati jer nije nužno svaki korisnik koji ima *root*-an uređaj maliciozni ali prepoznavanjem *root*-a može se zaustaviti pokretanje aplikacije kako se ne bi ugrozio integritet aplikacije [24].

7.1.4 Provjere integriteta aplikacije

Provjera integriteta aplikacije jedna je od ključnih detekcija kako bi se znalo da se nije „petljalo“ s aplikacijom. Kod integriteta datoteke, često se koristi:

- Uspoređivanje sadržaja memorije ili kontrolnog zbroja nad sadržajem s dobrim vrijednostima.
- Traženje u memoriji potpisa neželjenih izmjena – Može se pretraživati memorija i uspoređivati s listom neželjenih stringova ukoliko se pojavljuju u memoriji.

Također je već bio predstavljen SafetyNet Attestation API. Ovaj API se također može koristiti u svrhu provjere integriteta aplikacije tako da se procjeni uređaj na kojem se aplikacija izvodi [39].

7.1.5 Obfuskacija

Obfuskacija je proces transformacije koda i podataka kako bi ih bilo teže razumjeti.

Zamagljivanje često nosi cijenu u izvedbi tijekom izvođenja, stoga može biti primijenjeno samo na određene vrlo specifične dijelove koda, tipično na one koji se bave sigurnošću i zaštitom tijekom izvođenja.

Dodatne tehnike zamagljivanja kao što su "Sravnavanje kontrolnog toka" ili "Lažni kontrolni tijek", koje će biti detaljnije objašnjene u iOS dijelu, mogu su se primijeniti pomoću npr. Obfuscator-LLVM [24].

7.1.6 Detekcija emulatora

Otkrivanje emulatora osnovni je sigurnosni mehanizam koji otežava obrnuti inženjering softvera. Budući da je prevladavanje provjera emulatora ili korištenje fizičkog uređaja puno teže, obrnuti inženjerima je otežano izvesti veliku analizu uređaja na mobilnim uređajima.

Nekoliko je znakova da se aplikacija odvija u emulatoru. Unatoč činjenici da se svi API pozivi mogu *hook*-ati i promijeniti povratnu vrijednost, ove detekcije nude skromnu prvu liniju obrane.

U datoteci *build.prop* može se pronaći prvi skup indikatora da je uređaj emulator.

API Method	Value	Meaning
Build.ABI	armeabi	possibly emulator
BUILD.ABI2	unknown	possibly emulator
Build.BOARD	unknown	emulator
Build.Brand	generic	emulator
Build.DEVICE	generic	emulator
Build.FINGERPRINT	generic	emulator
Build.Hardware	goldfish	emulator
Build.Host	android-test	possibly emulator
Build.ID	FRF91	emulator
Build.MANUFACTURER	unknown	emulator
Build.MODEL	sdk	emulator
Build.PRODUCT	sdk	emulator
Build.RADIO	unknown	possibly emulator
Build.SERIAL	null	emulator
Build.USER	android-build	emulator

Slika 7.2. Pregled build.prop datoteke.

S obzirom na to da *root*-ani Android uređaj može uređivati datoteku *build.prop*, može se lako zaobići ova provjera.

Sljedeća provjera je koristeći TelephonyManager API. Svaki Android emulator ima fiksne vrijednosti kojima ovaj API može pristupiti, za razliku od vrijednosti na pravom uređaju.

API	Value	Meaning
TelephonyManager.getDeviceId()	0's	emulator
TelephonyManager.getLine1 Number()	155…..	emulator
TelephonyManager.getNetworkCountryIso()	us	possibly emulator
TelephonyManager.getNetworkType()	3	possibly emulator
TelephonyManager.getNetworkOperator().substring(0,3)	310	possibly emulator
TelephonyManager.getNetworkOperator().substring(3)	260	possibly emulator
TelephonyManager.getPhoneType()	1	possibly emulator
TelephonyManager.getSimCountryIso()	us	possibly emulator
TelephonyManager.getSimSerial Number()	8901…..	emulator
TelephonyManager.getSubscriberId()	3102…..	emulator
TelephonyManager.getVoiceMailNumber()	1555…..	emulator

Slika 7.3. Pregled vrijednosti koje se mogu dohvatiti koristeći TelephonyManager API.

Ove detekcije je relativno lako izmijeniti iskusnom napadaču.

Prvi način je da se neželjeno ponašanje „zakrpa“ tako da se jednostavnim prepisivanjem pridruženog bajt koda ili izvornog koda NOP uputama.

Drugi način je da se prikvači na API-je pomoću Frida ili Xposed API-ja. Umjesto originalnih vrijednosti emulatora, vrate se vrijednosti koje izgledaju nevino poput IMEI vrijednosti s pravog telefona [24].

7.1.7 Povezivanje uređaja

Cilj povezivanja uređaja je spriječiti napadača koji pokušava kopirati aplikaciju i njezino stanje s uređaja A na uređaj B i nastaviti s izvršavanjem aplikacije na uređaju B.

Postoje tri metode koje omogućuju bolje povezivanje uređaja na Androidu:

- Povećanje vjerodajnica koje se koriste identifikatorima uređaja za provjeru autentičnosti. To ima smisla ako aplikacija treba često ponovno autentificirati sebe i/ili korisnika.
- Šifriranje podataka pohranjenih u uređaju pomoću ključnog materijala koji je čvrsto vezan za uređaj može ojačati vezanje uređaja. Android Keystore nudi ne-izvozne privatne ključeve koje možemo koristiti za to. Kada bi zlonamjerni akter izvukao takve podatke s uređaja, ne bi bilo moguće dešifrirati podatke jer ključ nije dostupan.
- Upotrijebiti autentifikaciju uređaja na temelju tokena (ID instance) kako bi se osiguralo da se koristi ista instanca aplikacije.

U prošlosti su se Android programeri često oslanjali na *Settings.Secure.ANDROID_ID* (SSAID) i MAC adrese kako bi se identificirao uređaj. Od Androida 8.0 (razina API-ja 26), MAC adresa je sada često nasumična kada nije povezana s pristupnom točkom, a SSAID više nije ID vezan za uređaj. Umjesto toga, postao je vrijednost vezana za korisnika, uređaj i ključ za potpisivanje aplikacije koji zahtijeva SSAID.

Stoga, postoje nove preporuke za identifikatore u Google-ovoj SDK dokumentaciji:

- Korištenje ID oglašavanja (*AdvertisingIdClient.Info*) kada je u pitanju oglašavanje - tako da korisnik ima mogućnost odbiti.
- Korištenje ID instance (*FirebaseInstanceId*) za identifikaciju uređaja.
- Korištenje SSAID samo za otkrivanje prijevare i za dijeljenje stanja između aplikacija koje je potpisao isti programer.

Potrebno je imati na umu da ID instance i ID oglašavanja nisu stabilni tijekom nadogradnje uređaja i resetiranja uređaja. Međutim, ID instance će barem omogućiti identifikaciju trenutne instalacije softvera na uređaju.

Postoje određeni jedinstveni identifikatori koji više neće raditi te treba pripaziti da se njih ne koristi za provjeru povezanosti uređaja:

- *Build.SERIAL* bez *Build.getSerial*
- *htc.camera.sensor.front_SN* za HTC uređaje
- *persist.service.bdroid.bdadd*
- *Settings.Secure.bluetooth_address* ili *WifiInfo.getMacAddress* iz WifiManagera, osim ako je dopuštenje sustava LOCAL_MAC_ADDRESS omogućeno u manifestu.

Google Instance ID koristi tokene za provjeru autentičnosti pokrenute instance aplikacije. U trenutku kada se aplikacija resetira, deinstalira itd., ID instanca se resetira, što znači da će doći do nove "instance" aplikacije. Na primjer, može se koristiti ID instanca (iid) i token da se proslijede poslužitelju za provjeru valjanosti tokena i iid-a. Kada se iid ili token čine nevažećim, može se pokrenuti zaštitna procedura [21].

7.2. Moguće detekcije i načini zaštite na iOS platformi

7.2.1 Debuggiranje

Za pokretanje *debugging* procesa na iOS-u koristi se sistemski poziv ptrace(). Mogu ga koristiti *third-party* aplikacije te je potrebno onemogućiti pozivanje odmah prilikom inicijalizacije aplikacije. Sljedeći isječak koda prikazuje primjer onemogućavanja spajanje debugger-a [27]:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#import <dlfcn.h>
#define PT_DENY_ATTACH 31
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);
int main(int argc, char *argv[]) {
#ifndef DEBUG
    ptrace_ptr_t ptrace_ptr = dlsym(RTLD_SELF, "ptrace");
    ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
```

```
#endif
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}
```

Koristeći ptrace s PT_DENY_ATTACH, koji je prema dokumentaciji definiran brojem 31 [29], osigurava se da nijedan drugi program za ispravljanje pogrešaka ne može priključiti na proces pozivanja; čak i ako se pokuša priključiti program za ispravljanje pogrešaka, proces izlazi [28].

Važno je znati da ptrace metoda nije dio javnog API-ja na iOS-u, te prema pravilima objavljivanja App Store-a, zabranjena je uporaba nejavnih API-ja i njihova upotreba može dovesti do odbijanja aplikacije iz App Store-a. Zbog toga programeri ne pozivaju ptrace() izravno u kodu, već se pozivaju putem dobivanja pokazivača funkcije ptrace() koristeći dlsym – funkcija za dohvaćanje dinamičkih knjižica.

S obzirom na to da se pozivaju funkcije više razine koje omotavaju stvarni poziv sustava, iskusni napadač bi mogao pronaći u kodu gdje se poziva ptrace() i promijeniti *wrapper* funkciju prije nego se ona pozove.

Prethodni primjer koda je pozvao funkciju ptrace u biblioteci ptrace.o izloženoj preko <sys/ptrace.h> API-ja. Kako bi uspješnije zaštitili od napada, predlaže se da direktno u asemblerskom kodu pozove sistemski poziv.

```
int main(int argc, char * argv[]) {
    asm volatile (
        "mov x0 , #31 \n\t" // 0x1f
        "mov x1 , #0  \n\t"
        "mov x2 , #0  \n\t"
        "mov x3 , #0  \n\t"
        "mov x16, #26 \n\t" // 0x1a
        "svc #0x80"
    );
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
```

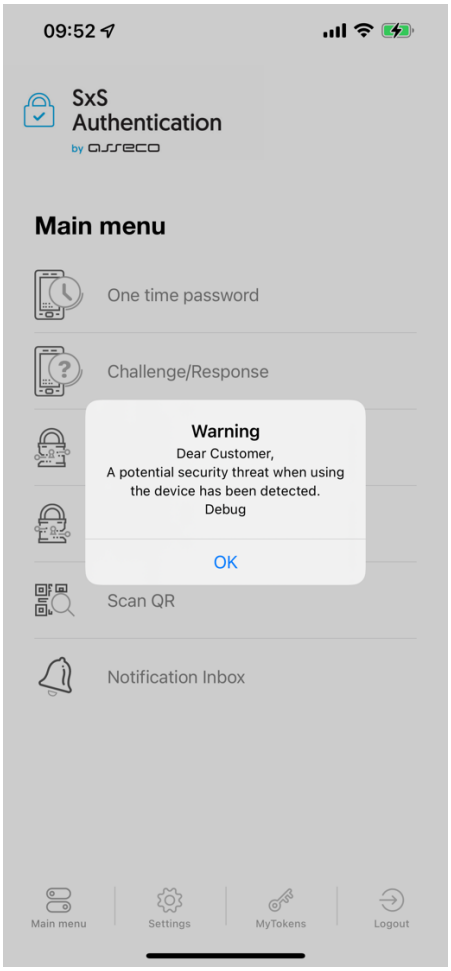
```
    }
}
```

Za referencu, broj koji se odnosi na syscall za ptrace je 26, a vrijednost zastavice PT_DENY_ATTACH je 31. Poziv se upućuje izravno na jezgru iOS-a, što ga čini mnogo težim za instrumentiranje.

Osim onemogućavanja spajanja na debugger, također postoji poziv sysctl() koji vraća informaciju ako se odvija proces debuggiranja. Taj sistemski poziv se može koristiti kao dodatna provjera, pogotovo ako napadač pokuša zaobići ptrace() metodu. Kada se poziva s odgovarajućim argumentima, funkcija sysctl() vraća strukturu s oznakom KP_proc.p_Flag koja označava status procesa te ako je uključeno debuggiranje ili nije [30].

```
inline int isDebugged() attribute((always_inline));
int isDebugged() {
    int name[4];
    struct kinfo_proc info;
    size_t info_size = sizeof(info);
    info.kp_proc.p_flag = 0;
    name[0] = CTL_KERN;
    name[1] = KERN_PROC;
    name[2] = KERN_PROC_PID;
    name[3] = getpid();
    if(sysctl(name, 4, &info, &info_size, NULL, 0) == -1)
        return 1; // bad sign
    return((info.kp_proc.p_flag & P_TRACED) != 0);
    // 0 is good - not being debugged
}
```

Ako nema napada na uređaju osim da je sustav u debug načinu rada, ove metode će pokazati ispravno detektiranje debuggera na uređaju. Ako je uređaj samo spojen preko USB-a neće se ništa detektirati te to možemo smatrati ispravnim načinom rada ove detekcije. Korištenjem ovih metoda u frameworku koji detektira napade u realnom vremenu, dobije se prikaz na Slici 7.4.



Slika 7.4. Prikaz detekcije debug načina rada na aplikaciji s frameworkom za zaštitu u realnom vremenu.

7.2.2 Hooking

Za aplikacije gdje je onemogućeno debuggiranje, zaobilazi se tako da se spoji na aplikaciju u vremenu izvođenja koristeći dinamičke knjižice dylibs. Postoje različite knjižice koje su stekle veliku popularnost i omogućavaju pregledavanje funkcije koje se izvode te njihovo modificiranje. Jedan od načina za detekciju zloćudnih knjižica je analiza učitanih dinamičkih knjižica u aplikacijskom *sandbox*-u te koristeći popis najpopularnijih, pronaći ih u imenu tih učitanih knjižica:

```
inline void checkDylibInjected() attribute((always_inline));
void checkDylibInjected() {
    uint32_t count = _dyld_image_count();
    char* blacklist[] = { "Substrate", "cycript" }; // Primjer maliciozne knjižice
```

```
    for(uint32_t i=0; i<count; i++) {
        const char *dyld = _dyld_get_image_name(i);
        int length = strlen(dyld);
        int j;
        for(j=length-1; j>= 0; --j)
            if(dyld[j] == '/')
                break;
        char *name = strndup(dyld + ++j, length - j);
        for(int x=0; x<sizeof(blacklist)/sizeof(char*); x++)
            if(strstr(name, blacklist[x]) || strstr(dyld, blacklist[x]))
                // Maliciozna knjižica učitana
        free(name);
    }
} [30]
```

Koristeći funkciju *_dydl_image_count()* dohvaća se broj učitanih knjižica, a s metodom *_dydl_get_image_name()* dohvaća se njeno ime.

Nedostatak ove metode je u tome što se prilikom ubacivanja knjižice može promijeniti njeno ime u bilo koji naziv te je ova metoda ne bi uspjela detektirati ako tog imena nema na popisu.

Nadalje, može se provjeriti gdje se nalazi izvorno mjesto metode pomoću funkcije dladdr().

Metode koje dolaze iz Apple SDK-ova obično se nalaze na sljedećim lokacijama:

- */System/Library/TextInput*
- */System/Library/Accessibility*
- */System/Library/PrivateFrameworks/*
- */System/Library/Frameworks/*
- */usr/lib/*

Sljedeća implementacija ponavlja metode dane klase i provjerava izvornu lokaciju slike u odnosu na skup poznatih mogućih lokacija slike te provjerava nalazi li se funkcija unutar puta u odnosu na samu aplikaciju [31]:

```
int isClassHooked(char * class_name) {
    char imagepath[512];
    int n;
    Dl_info info;
    id c = objc_lookUpClass(class_name);
    Method * m = class_copyMethodList(c, &n);
    for (int i=0; i<n; i++) {
        char *methodname = sel_getName(method_getName(m[i]));
        void *methodimp = (void *)method_getImplementation(m[i]);
        int d = dladdr((const void*)methodimp, &info);
        if(!d)
            return YES;
        //Provjera naspram poznatih lokacija
        memset(imagepath, 0x00, sizeof(imagepath));
        memcpy(imagepath, info.dli_fname, 9);
        if(strcmp[2](imagepath, "/usr/lib/") == 0)
            continue;
        memset(imagepath, 0x00, sizeof(imagepath));
        memcpy(imagepath, info.dli_fname, 27);
        if(strcmp(imagepath, "/System/Library/Frameworks/") == 0)
            continue;
        memset(imagepath, 0x00, sizeof(imagepath));
        memcpy(imagepath, info.dli_fname, 34);
        if(strcmp(imagepath, "/System/Library/PrivateFrameworks/") == 0)
            continue;
        memset(imagepath, 0x00, sizeof(imagepath));
        memcpy(imagepath, info.dli_fname, 29);
        if(strcmp(imagepath, "/System/Library/Accessibility") == 0)
            continue;
        memset(imagepath, 0x00, sizeof(imagepath));
        memcpy(imagepath, info.dli_fname, 25);
        if(strcmp(imagepath, "/System/Library/TextInput") == 0)
```

```
            continue;
        // provjera naziva slike u odnosu na lokaciju slike aplikacije
        if(strcmp(info.dli_fname, image_name) == 0)
            continue;
        return YES;
    }
    return NO;
}
```

Mogu se provesti daljnje provjere gdje *hook*-anje s dinamičkom knjižicom Cydia Substrate ostavlja trag na metodi tzv. *Trampoline.* Trampolin je skup instrukcija odgovornih za preusmjeravanje toka kontrole na novi fragment koda kako bi se zamijenilo izvorno ponašanje.

Ako je implementacija funkcije poznata unaprijed, prvih nekoliko bajtova pronađene funkcije može se usporediti s poznatim bajtovima. Za Cydia Substrate znamo da je funkcija zakrpljena s bezuvjetnom granom u registar (BR Xn), tako da možemo provjeriti nalazimo li takvu instrukciju u prvih nekoliko bajtova. Ako se pronađe instrukcija grananja, pretpostavljamo da je funkcija zakačena, u suprotnom pretpostavljamo da je metoda originalna. Ova implementacija vrijedi samo za ARM64 arhitekturu [32] :

```
int isSSLHooked() { void* (*createTrustFunc)() = dlsym(RTLD_DEFAULT,
"tls_helper_create_peer_trust");
        if(createTrustFunc == 0x0){
// Unable to find symbol, assume function is hooked.
return 1;
         }
unsigned int * createTrustFuncAddr = (unsigned int *) createTrustFunc;
// Verify if one of first three instructions is an unconditional branch
// to register (BR Xn), unconditional branch with link to register
// (BLR Xn), return (RET).
for(int i = 0; i < 3; i++){ int opCode = createTrustFuncAddr[i] & 0xfffffc1f; if(opCode ==
0xD61F0000){
// Instruction found, function is hooked.
```

```
return 1; } }
// Function is not hooked through a trampoline.
return 0; }
```

U posljednje vrijeme raste popularnost frida-alata za *hooking* te postoji više načina za njezinu detekciju ali i izbjegavanje. Prijašnji primjeri također mogu detektirati prisutnost Fride, ali zbog njenog specifičnog načina rada, potrebne su dodatne detekcije.

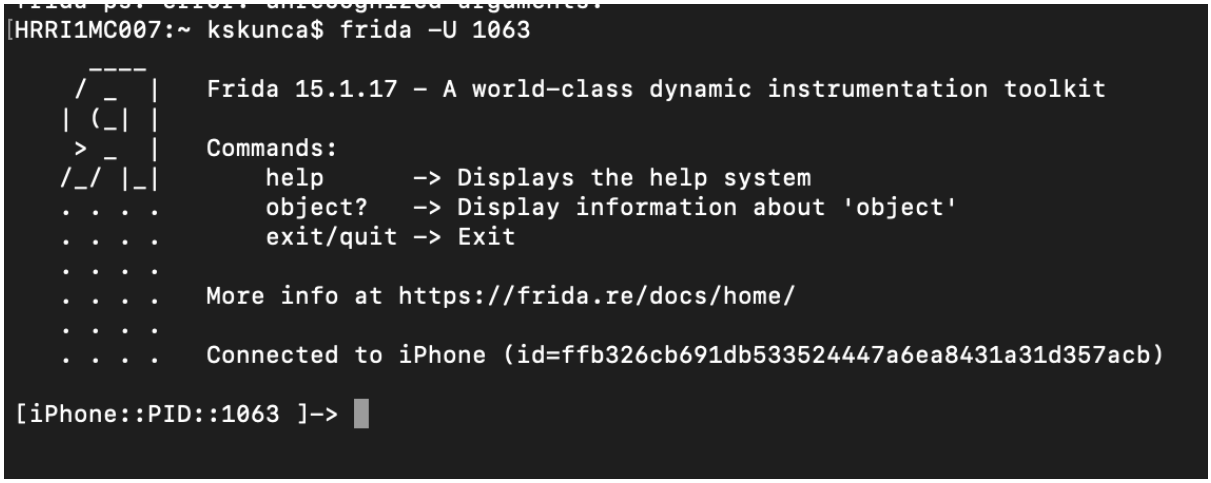
Frida se može ubaciti u aplikaciju tako da se u .ipa datoteku ubaci *frida_gadget.dylib* te se aplikacija ponovno potpiše s developerskim računom. Time se omogućava pokretanje aplikacije s malicioznom knjižicom na *jailbreak*-anom i *ne-jailbreak*-anom uređaju.

Ako korisnik ima *jailbreak*-an uredaj, može preko Cydia aplikacije instalirati frida-server te pomoću njega prisluškivati aplikaciju. Pomoću servera nije potrebno ponovno potpisivati aplikaciju već je moguće *hook*-ati bilo koju aplikaciju koja je pokrenuta na uređaju. Jednom kad je instaliran frida-server na uređaju, možemo se spojiti s računalom tako da spojimo mobitel na računalo s USB-om te u terminalu upišemo *frida-ps -U*.

```
HRRI1MC007:~ kskunca$ frida-ps -U
Waiting for USB device to appear...
 PID  Name
----  ------------------------------------------------------
 192  ACCHWComponentAuthService
8285  AMDEngagementExtension
1426  ANECompilerService
 417  ANEStorageMaintainer
 384  ASPCarryLog
 392  AccountExtension
8073  AppPredictionIntentsHelperService
 348  AppSSODaemon
 331  AppStoreWidgetsExtension
  83  AppleCredentialManagerDaemon
 292  AssetCacheLocatorService
 777  BTLEServer
 270  BiomeLighthousePlugin
 162  BlueTool
 338  CAReportingService
 197  CMFSyncAgent
 298  CacheDeleteAppContainerCaches
```

Slika 7.5. Prikaz izlistaja procesa koristeći frida-alat.

Tom komandom dobijemo listu svih aktivnih procesa na uređaju. Ako se želimo spojiti na neki od procesa, korištenjem komande *frida -U {PID},* gdje PID predstavlja ID procesa.



Slika 7.6. Prikaz instrumentiranja aplikacije koristeći frida-alat.

Jedan od načina kako detektirati frida-server je detekcija alata u zadanoj konfiguraciji koja radi na adresi 127.0.0.1 i portu 27042. Ako port nije slobodan, traži se sljedeći slobodan [31].

```
private static func checkOpenedPorts() -> Bool {
    let ports = [
      27042, // default Frida
      27043,
      27044,
      27045,
    ]
    for port in ports {
      if canOpenLocalConnection(port: port) {
        return true
      }
    }
    return false
  }
  private static func canOpenLocalConnection(port: Int) -> Bool {

    func swapBytesIfNeeded(port: in_port_t) -> in_port_t {
      let littleEndian = Int(OSHostByteOrder()) == OSLittleEndian
      return littleEndian ? _OSSwapInt16(port) : port
```

```
    }

    var serverAddress = sockaddr_in()
    serverAddress.sin_family = sa_family_t(AF_INET)
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1")
    serverAddress.sin_port = swapBytesIfNeeded(port: in_port_t(port))
    let sock = socket(AF_INET, SOCK_STREAM, 0)

    let result = withUnsafePointer(to: &serverAddress) {
        $0.withMemoryRebound(to: sockaddr.self, capacity: 1) {
            connect(sock, $0, socklen_t(MemoryLayout<sockaddr_in>.stride))
        }
    }

    defer {
        close(sock)
    }

    if result != -1 {
        return true // Port is opened
    }
    return false
}
```

Pošto Frida koristi DBus protokol, dodatno se provjera odgovor s porta - ako se pošalje "AUTH" poruka, Frida će vratiti "REJECTED" pošto ona trenutno komunicira na tom portu.

Ova detekcija se da lako zaobići tako što se promjeni adresa i port na kojem komunicira Frida. To otežava detekciju Fride jer na iOS-u nisu dopušteni pozivi za otkrivanje aktivnih TCP konekcija, a preskupo je prolaziti po svakom portu i čekati odgovor [33]. Zbog toga je u ovom primjeru ograničeno pretraživanje na 4 porta.

Također je poznata putanja na kojoj se instalira server, a to je: /usr/sbin/frida-server. To je manje poželjna detekcija jer frida-server ne mora biti upaljen, a detektira se *hook* [31].

```
private static func checkExistenceOfSuspiciousFiles() -> Bool {
    let paths = [
      "/usr/sbin/frida-server"
    ]
    for path in paths {
      if FileManager.default.fileExists(atPath: path) {
        return true
      }
    }
    return false
  }
```

Prilikom testiranja, frida-alati su predstavljali poprilično zahtjevan problem. Detekcija Frida datoteke na uređaju se može koristiti samo na *jailbreak*-nim uređajima jer se ta datoteka preuzima s Cydia aplikacije. Ova metoda se smatra nepoželjnom detekcijom jer se detektira *hook*-anje aplikacije, a frida-server *daemon* može biti nepokrenuti.

Zatim, detekcija malicioznih knjižica se iznimno lako zaobiđe tako da se promjeni naziv dinamičke knjižice i s tim imenom se ubaci u aplikaciju te time detekcija gubi na svojoj vrijednosti. Način kako bi se ovo moglo poboljšati je *hash*-iranje popisa dinamičkih knjižica koje će se sigurno nalaziti u aplikaciji te ako je prilikom izvođenja aplikacije izračunati *hash* trenutno učitanih knjižica te ako je različiti od prvotnog *hash*-a, možemo pretpostaviti da se u aplikacijskom *sandbox*-u nalazi neželjena knjižica.

Zatim je bio problem s detekcijom uključenog frida-servera jer, iako je bio u originalnoj konfiguraciji, port je zapravo bio na portu 27043 te se iz tog razloga dodalo više portova u slučaju da se Frida nalazi na nekom narednom. Metoda koja provjerava portove je iznimno vremenski skupa te je to još jedan razlog zašto se ograničilo na samo par portova. Iako na iOS-u postoji sistemski poziv za dohvaćanje svih aktivnih TCP konekcija, on je zabranjen te njegovim pozivanjem sustav vrati -1. Pokušaj ispitivanja svih portova je trajao u minutama te je jedino preostalo ispitivanje originalne konfiguracije.

Zanimljiv mehanizam zaštite je detekcija trampolina koja je prije navedena. Testiranjem se utvrdilo da se pomoću te metode stvarno detektira *hook* jer je metoda sadržavala *jump* naredbu na četvrtoj liniji. Ovu metodu je potrebno oprezno koristiti jer je

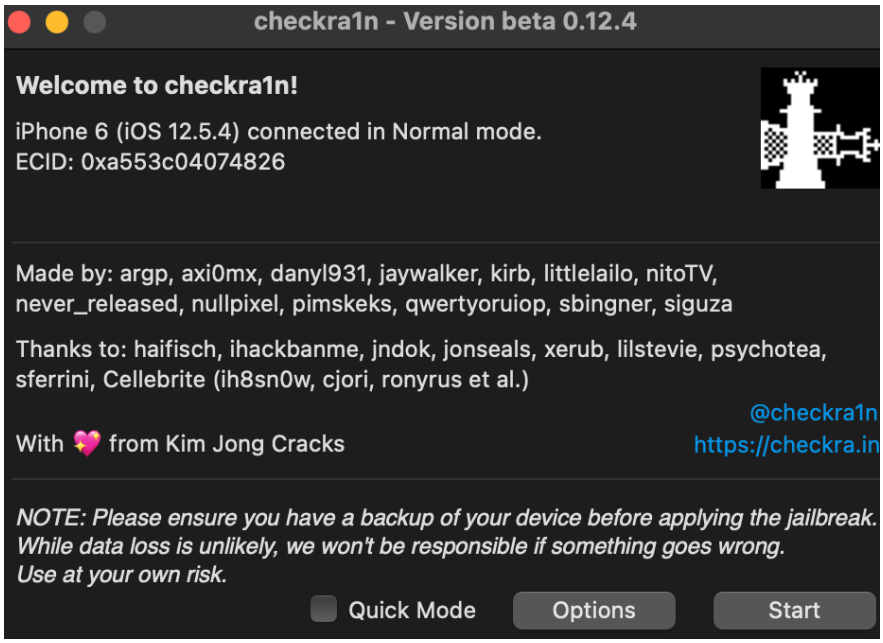
potrebno unaprijed znati koristili li metoda koju se provjera *jump* naredbu. Poželjno bi bilo koristiti kod kritičnih funkcija koje su dobro testirane.

7.2.3 Jailbreak

Detekcija *jailbreak*-a je potrebna jer omogućava lakše obrnuti inženjering. Iako je lako zaobići mjere koje se postavi kao zaštita od *jailbreak*-a, i dalje je poželjno postaviti dok se aplikacija izvršava kako bi se otežao *tampering* aplikacije.

Postoje 4 vrste *jailbreak*-a:

- *Tethered Jailbreak* – Nakon što se uređaj isključi (ili se baterija isprazni), uređaj ne može dovršiti ciklus pokretanja bez pomoći računalne aplikacije za *jailbreak* i fizičke kabelske veze između uređaja i dotičnog računala. Ako se želi ponovno pokrenuti uređaj nakon što je ugašen, telefon se mora ponovno povezati s računalom, staviti u DFU način rada i ponovno napraviti *jailbreak*. Najpopularniji *jailbreak* iz ove kategorije je RedSn0w.
- *Semi-Tethered Jailbreak* – Nakon što se uređaj isključi (ili se baterija isprazni), uređaj se može ponovno pokrenuti bez potrebne za pokretanjem preko računala, ali *jailbreak* više neće biti aktivan. Ako se telefon ponovno pokrene, možete se koristiti normalno, ali aplikacije kao što je Cydia će se srušiti nakon pokretanja i *tweak*-ovi se neće učitati. Kako bi *jailbreak* bio ponovno aktivan, jezgru telefona treba ponovno zakrpiti putem računala. Najpopularniji *jailbreak* iz ove kategorije je checkra1n.



Slika 7.7. Sučelje za checkra1n program za jailbreak.

- *Semi-Untethered Jailbreak* – Ova vrsta *jailbreak*-a omogućuje uređaju da završi ciklus podizanja sustava nakon što je bio *jailbreak*-an, ali *tweak*-ovi za *jailbreak* neće se učitati sve dok se na samom uređaju ne implementira *sideloaded* aplikacija za bijeg iz zatvora. *Semi-Untethered* aplikacija za *jailbreak* može se *sideload*-ati preko fizičke kabelske veze s računala putem programa Cydia Impactor ili preuzeti izravno na sam uređaj uz uslugu potpisivanja kao što je Ignition. Najpopularniji su Chimera i unc0ver.
- *Untethered Jailbreak* – *Untethered jailbreak* je onaj koji dopušta uređaju da završi ciklus pokretanja nakon što je pokrenuti bez ikakvog prekida funkcionalnosti orijentirane na *jailbreak*. Nevezani *jailbreakovi* su češće traženi od drugih, ali ih je i najizazovnije postići zbog snažnih podviga i razvojnih vještina koje zahtijevaju za razvoj. *Untethered jailbreak* se može poslati putem fizičke USB kabelske veze s računalom ili izravno na sam uređaj putem eksploatacije temeljene na aplikaciji, kao što je web stranica u Safariju. Najpopularniji su Pangu i JailbreakMe.

Postoji više načina detekcije *jailbreak*-a. Jedno od tih načina je provjera datoteka prisutnih na uređaju. Implementacija pretraživanja putanja na uređaju s poznatim putanjama dana je sljedećim kodom [31]:

```
private static func checkExistenceOfSuspiciousFiles() -> Bool {
    var paths = [
        "/usr/sbin/frida-server", // frida
        "/etc/apt/sources.list.d/electra.list", // electra
        "/etc/apt/sources.list.d/sileo.sources", // electra
        "/.bootstrapped_electra", // electra
        "/usr/lib/libjailbreak.dylib", // electra
        "/jb/lzma", // electra
        "/.cydia_no_stash", // unc0ver
        "/.installed_unc0ver", // unc0ver
        "/jb/offsets.plist", // unc0ver
        "/usr/share/jailbreak/injectme.plist", // unc0ver
        "/etc/apt/undecimus/undecimus.list", // unc0ver
        "/var/lib/dpkg/info/mobilesubstrate.md5sums", // unc0ver
```

```
"/Library/MobileSubstrate/MobileSubstrate.dylib",
"/jb/jailbreakd.plist", // unc0ver
"/jb/amfid_payload.dylib", // unc0ver
"/jb/libjailbreak.dylib", // unc0ver
"/usr/libexec/cydia/firmware.sh",
"/var/lib/cydia",
"/etc/apt",
"/private/var/lib/apt",
"/private/var/Users/",
"/var/log/apt",
"/Applications/Cydia.app",
"/private/var/stash",
"/private/var/lib/apt/",
"/private/var/lib/cydia",
"/private/var/cache/apt/",
"/private/var/log/syslog",
"/private/var/tmp/cydia.log",
"/Applications/Icy.app",
"/Applications/MxTube.app",
"/Applications/RockApp.app",
"/Applications/blackra1n.app",
"/Applications/SBSettings.app",
"/Applications/FakeCarrier.app",
"/Applications/WinterBoard.app",
"/Applications/IntelliScreen.app",
"/private/var/mobile/Library/SBSettings/Themes",
"/Library/MobileSubstrate/CydiaSubstrate.dylib",
"/System/Library/LaunchDaemons/com.ikey.bbot.plist",
"/Library/MobileSubstrate/DynamicLibraries/Veency.plist",
"/Library/MobileSubstrate/DynamicLibraries/LiveClock.plist",
"/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist",
"/Applications/Sileo.app",
"/var/binpack",
```

```
        "/Library/PreferenceBundles/LibertyPref.bundle",
        "/Library/PreferenceBundles/ShadowPreferences.bundle",
        "/Library/PreferenceBundles/ABypassPrefs.bundle",
        "/Library/PreferenceBundles/FlyJBPrefs.bundle",
        "/usr/lib/libhooker.dylib",
        "/usr/lib/libsubstitute.dylib",
        "/usr/lib/substrate",
        "/usr/lib/TweakInject",
        "/var/binpack/Applications/loader.app", // checkra1n
        "/Applications/FlyJB.app", // Fly JB X
        "/Applications/Zebra.app" // Zebra
    ]
    for path in paths {
        if FileManager.default.fileExists(atPath: path) {
            return false
        }
    }
    return true
}
```

Pregledavaju se putanje te ako se naiđe na neku od datoteka, detektira se *jailbreak*.

Sljedeći način detekcije *jailbreak*-a je testiranje ako korisnik ima prava pisanja/čitanja van aplikacijskog *sandbox*-a. To se može testirati pokušajem kreiranja datoteku u /private direktoriju [31]:

```
    private static func checkRestrictedDirectoriesWriteable() -> Bool {
    let paths = [
        "/",
        "/root/",
        "/private/",
        "/jb/"
    ]

    // If library won't be able to write to any restricted directory the return(false, ...) is never
```

```
reached
        // because of catch{} statement
        for path in paths {
            do {
                let pathWithSomeRandom = path+UUID().uuidString
                try "AmIJailbroken?".write(toFile: pathWithSomeRandom, atomically: true,
encoding: String.Encoding.utf8)
                try FileManager.default.removeItem(atPath: pathWithSomeRandom) // clean if
succesfully written
                return false
            } catch {}
        }
        return true
    }
```

Treći način moguće detekcije je korištenje URL sheme koja otvara Cydia aplikaciju. URL sheme se koriste za otvaranje aplikacije putem linka ako je aplikacija instalirana na uređaju. Ako je Cydia instalirana na uređaju, link koji počinje s cydia:// će uspješno otvoriti Cydia aplikaciju i time dokazati da je uređaj *jailbreak*-an [31].

```
private static func canOpenUrlFromList(urlSchemes: [String]) -> Bool {
        for urlScheme in urlSchemes {
            if let url = URL(string: urlScheme) {
                if UIApplication.shared.canOpenURL(url) {
                    return false
                }
            }
        }
        return true
    }
```

Nadalje se mogu testirati API sistemski pozivi. Pozivanje fork() funkcije na ne *jailbreak*-anom uredaju nije moguće jer *sandbox* odbija stvaranje *fork* procesa. Ako je moguće *fork*-ati, vrlo vjerojatno se radi o *jailbreak*-anom uredaju.

Pozivanje funkcije system() s NULL argumentom na uređaju koji nije *jailbroken* vratit će 0. Ako se pozove system() funkcija na *jailbroken* uređaju vratit će se 1. To je zato što će funkcija provjeriti postoji li /bin/sh, a postoji samo ako je *jailbroken* uređaj [31].

```
private static func checkFork() -> Bool {
    let pointerToFork = UnsafeMutableRawPointer(bitPattern: -2)
    let forkPtr = dlsym(pointerToFork, "fork")
    typealias ForkType = @convention(c) () -> pid_t
    let fork = unsafeBitCast(forkPtr, to: ForkType.self)
    let forkResult = fork()

    if forkResult >= 0 {
      if forkResult > 0 {
        kill(forkResult, SIGTERM)
      }
      return false
    }
    return true
  }
```

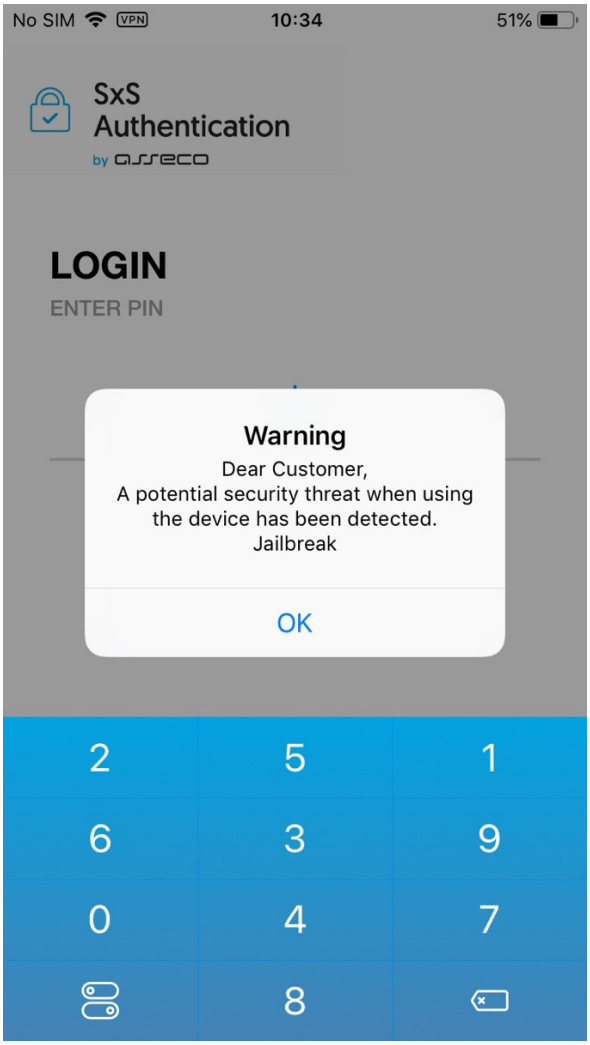
Prilikom *jailbreak*-anja uređaja, određeni se podaci moraju premjestiti na veću particiju. Budući da staro mjesto datoteke mora ostati valjano, stvaraju se simboličke veze. Detekcijom datoteka i direktorija koje mogu biti simboličke veze, može se otkriti *jailbreak*-an uređaj [34].

Sljedećim isječkom koda može se provjeriti postojanje simboličkih veza [31]:

```
private static func checkSymbolicLinks() -> Bool {
    let paths = [
      "/var/lib/undecimus/apt", // unc0ver
      "/Applications",
      "/Library/Ringtones",
      "/Library/Wallpaper",
      "/usr/arm-apple-darwin9",
      "/usr/include",
```

```
        "/usr/libexec",
        "/usr/share"
    ]
    for path in paths {
        do {
            let result = try FileManager.default.destinationOfSymbolicLink(atPath: path)
            if !result.isEmpty {
                return false
            }
        } catch {}
    }
    return true
}
```

Testiranjem *jailbreak*-a sve ove metode su se pokazale korisnima. Na osobnom uređaju, instaliran je bio Semi-Tethered Jailbreak checkra1n te je uvijek bila uspješno prikazana detekcija *jailbreak*-a koristeći navedene metode.



Slika 7.8. Prikaz detekcije jailbreak-a na aplikaciji s frameworkom za zaštitu u realnom vremenu.

7.2.4 Provjere integriteta aplikacije

Za provjeru integriteta aplikacije, od iOS 14, se može koristiti dijeljena instance klase *DCAppAttestService* koja provjerava legitimnost aplikacije prema serveru. Pomoću te klase, u aplikaciji se generira hardverski, kriptografski ključ koji se zatim provjerava koristeći Apple servere kako bi se potvrdilo da ključ pripada valjanoj instanci aplikacije. Zatim se ta usluga može koristiti za kriptografsko potpisivanje zahtjeva poslužitelja pomoću certificiranog ključa [35].

Ako se koristi iOS manji od 14 ili se želi dodati dodatni mehanizam zaštite, moguće je uvesti provjere integriteta na razini izvornog koda aplikacije. Prikaz takve zaštite je dan kodom [36]:

```
int xyz(char *dst) {
    const struct mach_header * header;
    Dl_info dlinfo;
    if (dladdr(xyz, &dlinfo) == 0 || dlinfo.dli_fbase == NULL) {
        NSLog(@" Error: Could not resolve symbol xyz");
        [NSThread exit];
    }

    while(1) {

        header = dlinfo.dli_fbase;  // Pointer on the Mach-O header
        struct load_command * cmd = (struct load_command *)(header + 1); // First load
command
        // Now iterate through load command
        //to find __text section of __TEXT segment
        for (uint32_t i = 0; cmd != NULL && i < header->ncmds; i++) {
            if (cmd->cmd == LC_SEGMENT) {
                // __TEXT load command is a LC_SEGMENT load command
                struct segment_command * segment = (struct segment_command *)cmd;
                if (!strcmp(segment->segname, "__TEXT")) {
                    // Stop on __TEXT segment load command and go through sections
                    // to find __text section
                    struct section * section = (struct section *)(segment + 1);
                    for (uint32_t j = 0; section != NULL && j < segment->nsects; j++) {
                        if (!strcmp(section->sectname, "__text"))
                            break; //Stop on __text section load command
                        section = (struct section *)(section + 1);
                    }
                    // Get here the __text section address, the __text section size
                    // and the virtual memory address so we can calculate
                    // a pointer on the __text section
                    uint32_t * textSectionAddr = (uint32_t *)section->addr;
                    uint32_t textSectionSize = section->size;
```

```
                uint32_t * vmaddr = segment->vmaddr;
                char * textSectionPtr = (char *)((int)header + (int)textSectionAddr -
(int)vmaddr);
                // Calculate the signature of the data,
                // store the result in a string
                // and compare to the original one
                unsigned char digest[CC_MD5_DIGEST_LENGTH];
                CC_MD5(textSectionPtr, textSectionSize, digest);     // calculate the signature
                for (int i = 0; i < sizeof(digest); i++)             // fill signature
                   sprintf(dst + (2 * i), "%02x", digest[i]);
                // return strcmp(originalSignature, signature) == 0;    // verify signatures match
                return 0;
             }
          }
          cmd = (struct load_command *)((uint8_t *)cmd + cmd->cmdsize);
       }
    }
}
```

Mach_header se raščlanjuje kako bi se izračunao početak podataka instrukcije, koji se koriste za generiranje potpisa. Zatim se potpis uspoređuje s danim potpisom. Provjerava se ako je generirani potpis pohranjen ili kodiran negdje drugdje.

Prilikom osiguravanja integriteta same pohrane aplikacije, može se stvoriti HMAC - kod za provjeru autentičnosti poruke zasnovan na *hash*-u - bilo na paru ključ/vrijednost ili na datoteci pohranjenoj na uređaju. Implementacija *CommonCrypto* se preporučuje za stvaranje HMAC-a. *CommonCrypto* knjižnica podržava simetrično šifriranje, šifre za provjeru autentičnosti poruka koje se temelje na *hash*-u i sažetke [24, 37].

Prednost korištenja metoda provjere integriteta je u tome da možda većina napadača nije upoznati s tim mehanizmom zaštite te se time dodaje novi sloj zaštite aplikacije [24].

7.2.5 Obfuskacija

Obfuskacija koda je jedna od ključnih elemenata zaštite. Svrha obfuskacije je transformacija koda kako bi se teže rastavila aplikacije te ujedno razumjela. Cilj je da nakon obfuskacije koda, aplikacija i dalje zadrži svoje funkcionalnosti.

Sljedeće tehnike se mogu koristiti u svrhu obfuskacije:

- Obfuskacija imena – Ova tehnika obuhvaća promjenu naziva imena klasa, varijabli i metoda. Ako se imena ne zamijene obfuciranom inačicom, pregledom binarne datoteke aplikacije, moguće je lagano pretraživanje ključnih riječi poput *jailbreak* ili Frida. Ako se pronađu ključne riječi, napadač zna gdje treba napraviti izmijene kako bi promijenio detekcije tih funkcija.
- Zamjena instrukcija – Ova tehnika zamjenjuje standardne binarne operatore poput zbrajanja i oduzimanja. Na primjer, zbrajanje x = a + b može se predstaviti kao x = -(-a) - (-b). Potrebno je dodati više tehnika za zamjene kako bi osiguralo da je teže obrnuti natrag ovu zamjenu.
- Izravnavanje toka kontrole – Ovom tehnikom se zamjenjuje izvorni kod sa složenijim prikazom. Transformacija koda razbija tijelo funkcije u osnovne blokove i sve ih stavlja unutar jedne beskonačne petlje s naredbom *switch* koja kontrolira tijek programa. To čini tijek programa znatno težim za praćenje jer uklanja prirodne uvjetne konstrukcije koje obično čine kod lakšim za čitanje.
- Ubacivanje mrtvog koda - Ova tehnika čini kontrolni tijek programa složenijim tako da se ubaci „mrtvi“ kod u program. Taj dio koda koji ne utječe na ponašanje izvornog programa, ali znatno otežava proces obrnutog inženjeringa.
- Enkripcija stringova – Ova tehnika se preporuča za kriptiranje tvrdo kodiranih ključeva, licencama, tokena i URL-ova te da se zatim ubace dijelovi koda u program koji će dešifrirati te podatke prije nego što ih program upotrijebi.

Cilj ovih tehnika je da se obeshrabri napadača od daljnje analize koda, ali ne mora nužno značiti da će i zaustaviti.

S obzirom na to da je vremenski skupo vlastito razvijanje obfuskatora, često se preporučuje *SwiftShield* ili *obfuscator-llvm*.

SwiftShield može prikrivati imena, a funkcionira tako da čita izvorni kod *Xcode* projekta i zamjenjuje sva imena klasa, metoda i polja sa slučajnim vrijednostima prije nego što se koristi prevodilac.

obfuscator-llvm radi na *Intermediate Representation* (IR) umjesto na izvornom kodu. Može prikrivati simbole, kriptirati nizove i izravnavati tok kontrole. Budući da se temelji na IR-u, može sakriti znatno više informacija o aplikaciji u odnosu na *SwiftShield* [24].

7.2.6 Simulator

Cilj otkrivanja simulatora je povećati poteškoće pokretanja aplikacije na simuliranom uređaju. Međutim, to nije problem na iOS-u jer je jedini dostupni simulator onaj koji se isporučuje s Xcode-om. Binarne datoteke simulatora se kompiliraju u x86 kod umjesto ARM koda, a aplikacije kompilirane za pravi uređaj (ARM arhitektura) ne rade u simulatoru te se zbog toga simulator ne može koristiti za obrnuti inženjering [24].

7.2.7 Povezivanje uređaja

Kao što je već bilo navedeno u Android sekciji, cilj povezivanja uređaja je spriječiti napadača koji pokušava kopirati aplikaciju i njezino stanje s uređaja A na uređaj B i nastaviti s izvršavanjem aplikacije na uređaju B. Korisnost u povezivanju uređaja u tome sto nakon što je uređaj A utvrđen kao pouzdan, može imati više privilegija od uređaja B. Potrebno je zaštiti uređaj da se te različite privilegije ne mijenjaju kada se aplikacija kopira s uređaja A na uređaj B.

Od iOS-a 7.0, identifikatori poput MAC adresu nisu više dostupni. Za uspješno povezivanje aplikacije s uređajem mogu se koristiti sljedeće opcije:

- Korištenje *UIDevice.current.identifierForVendor?.uuidString* – Alfanumerički niz koji jedinstveno identificira uređaj.
- Spremanje podataka u Keychain – Podaci se spremaju u Secure Enclave-u na uređaju i potrebno je dodati zastavice kako bi podaci bili sigurno spremljeni: *kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly* ili *kSecAttrAccessibleWhenUnlockedThisDeviceOnly*.
- Korištenje Google's InstanceID za iOS – Ova instanca se veže direktno uz instancu aplikacije tako da, ako se izbriše aplikacija ili resetira, stvori će se nova instanca aplikacije [24].

8. Demonstracija napada na iOS aplikaciju

Alati za *tampering* s aplikacijom su svima dostupni i postoji puno resursa koji pomažu u tome iako nisu bili stvoreni sa zlobnom namjerom.

U nastavku će biti prikazan postupak izmijene aplikacije gdje je cilj izmijeniti metodu koja provjerava ako je uređaj *jailbreak*-an ili ne. Svrha ove demonstracije je prikaz koraka koje bi napadač napravio u slučaju napada na aplikaciju te saznati gdje je sve potrebno zaštiti aplikaciju.

Ako nemamo pristup IPA datoteci aplikacije kojoj želimo izmijeniti ponašanje, istu možemo preuzeti s AppStore-a. Postoje dva načina:

1. Koristeći Apple Configurator 2 – Za ovu metodu nije potrebno imati *jailbroken* iPhone.
2. Koristeći iOS executable dumper – Za ovu metodu je potreban *jailbreak*-an uređaj [40].

S obzirom na to da je ilegalno proučavati aplikaciju bez znanja vlasnika te aplikacije [41], koristit će se DViA-v2 aplikacija čija je namjena učenje *pentesting*-a u legalnom okruženju.

Kako bi instrumentirali aplikaciju s frida-alatima, možemo koristiti *jailbreak*-an ili *ne-jailbreak*-an uređaj.

Ako se koristi *ne-jailbreak*-an uređaj, potrebno je dodati dinamičku knjižicu frida-gadget.dylib te ponovno potpisati aplikaciju. Ako se ponovno ne potpiše aplikacija, neće se moći pokrenuti jer nisu sve knjižice potpisane istim developerskim certifikatom. U tu svrhu može se koristiti projekt s GitHub-a *resign.* Kako bi se uspješno pokrenuo projekt, potrebno je unutar projekta smjestiti željenu .ipa datotetku i dinamičku knjižicu. Potrebnu dinamičku knjižicu može se preuzeti s linka: https://github.com/frida/frida/releases.

Na Slici 8.1. možemo vidjeti kako bi trebala izgledati struktura *resign* projekta.

Slika 8.1. Struktura resign projekta.

Kada su obavljeni svi prethodni koraci, može se izgraditi i pokrenuti projekt. S obzirom na to da je ubačena Frida knjižica, aplikacija je zamrznuta te je potrebno pokrenuti frida-server na računalu u terminalu s komandom:

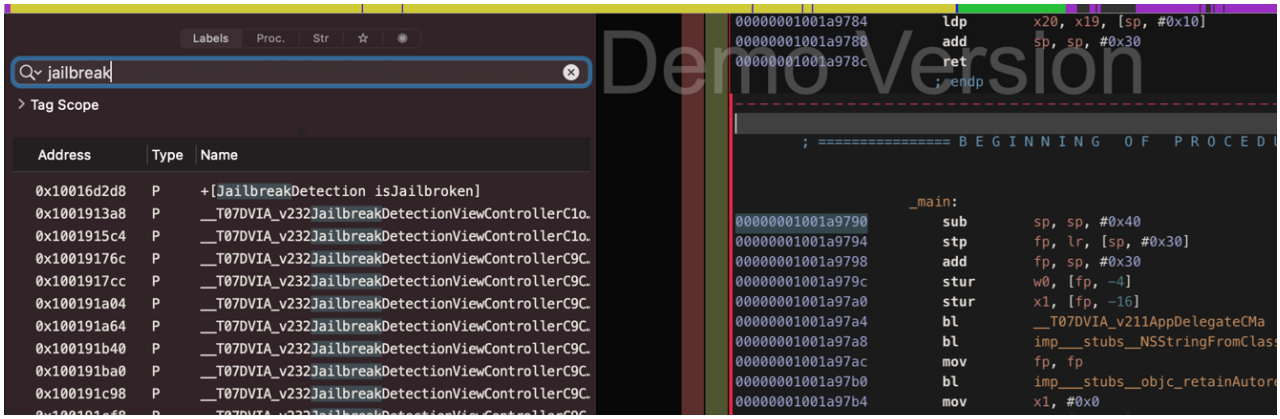
- **frida-ps -U *ImeAplikacije** ili **PID** aplikacije*.

Ako se koristi *jailbroken* uređaj, proces je jednostavniji jer se može instalirati frida-server na uređaju preko Cydia aplikacije te je onda omogućeno instrumentiranje bilo koje pokrenute aplikacije na uređaju.

Nakon što je uspješno pokrenuta Frida, potrebno je definirati na koji dio se želimo prikvačiti.

Kako bi saznali u kojem dijelu aplikacije se nalazi dio na koji se želimo prikvačiti, moramo saznati gdje se u kodu pojavljuje metoda koju želimo. U ovu svrhu možemo koristiti Hopper program definiran u sekciji 6.4.4.

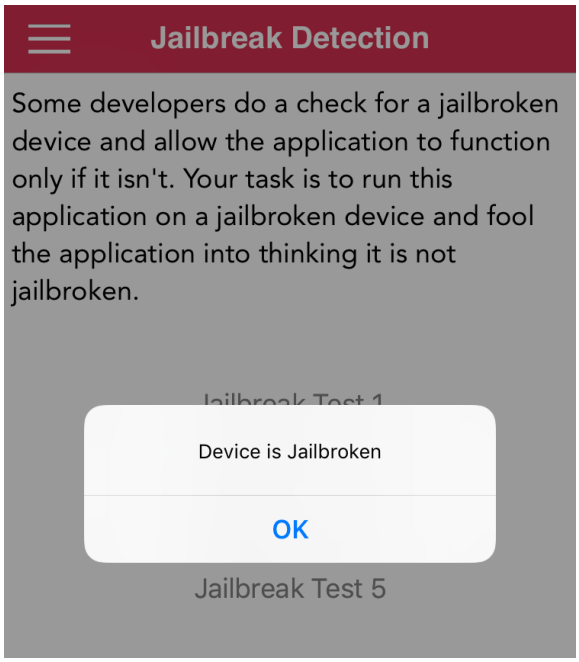
Pokretanjem Hopper programa i ubacivanjem .ipa datoteke, dobivamo aplikaciju rastavljenu na njene osnovne metode, tok programa i korištene varijable. S obzirom na to da se želimo prikvačiti na metodu koja detektira *jailbreak*, pretraživat ćemo ključnu riječ „*jailbreak*“.



Slika 8.2. Prikaz pretraživanja ključne riječi u Hopper programu.

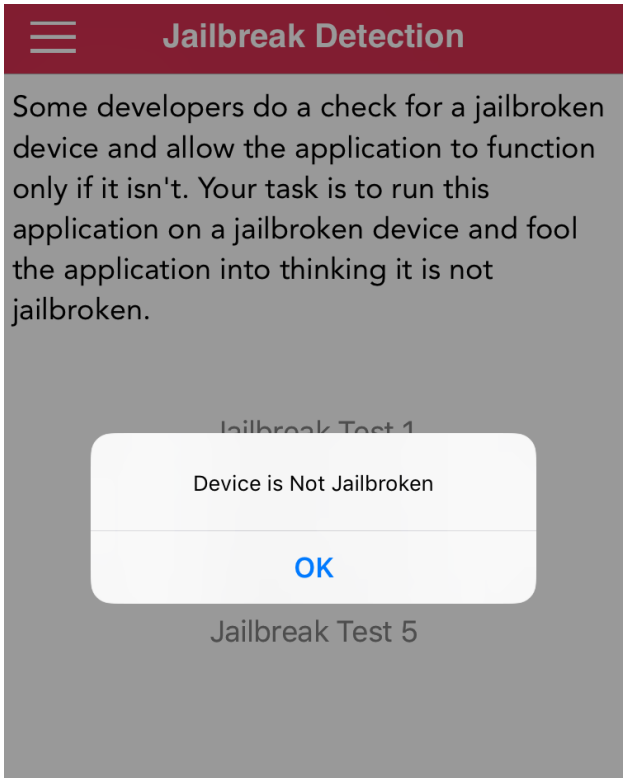
Kada saznamo koju klasu i metodu želimo izmijeniti, koristeći JavaScript skripte i Fridu možemo izmijeniti ponašanje.

S obzirom na to da je uređaj *jailbroken*, stanje koje se prikazuje na Slici 8.3. je ispravno:



Slika 8.3. Prikaz ispravnog ponašanja aplikacije.

Pokretanjem sljedeće skripte s komandom *frida -U --no-pause -l *ime_skripte* -f *app bundle identifier**, dobit će se rezultata na Slici 8.4.:



Slika 8.4. Prikaz izmijenjenog ponašanja aplikacije.

Kod skripte koja je pokrenuta s Fridom je dana u nastavku [42]:

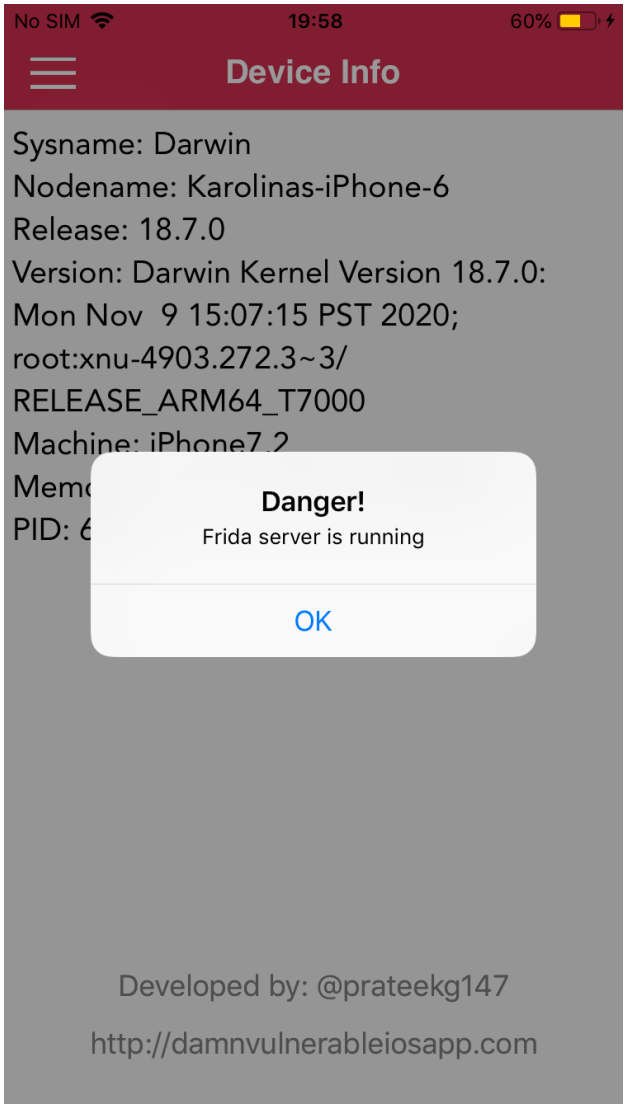
```
function show_modify_function_return_value(className_arg, funcName_arg)
{
   var className = className_arg;
   var funcName = funcName_arg;
   var hook = ObjC.classes[className][funcName];
   Interceptor.attach(hook.implementation, {
     onLeave: function(retval) {
       console.log("\n[*] Class Name: " + className);
       console.log("[*] Method Name: " + funcName);
       console.log("\t[-] Type of return value: " + typeof retval);
       console.log("\t[-] Return Value: " + retval);
       var newretval = ptr("0x0") //nova vrijednost koja se upisuje u registar
       retval.replace(newretval)
       console.log("\t[-] New Return Value: " + newretval)
     }
   });
```

```
}
show_modify_function_return_value("JailbreakDetection" ,"+ isJailbroken")
```

Zatim je u aplikaciju dodan isječak koda za zaštitu od frida-servera koji sluša na zadanom portu definiran u sekciji 7.1.2:

```
private static func checkOpenedPorts() -> Bool {
    let ports = [
        27042, // default Frida
        27043,
        27044,
        27045,
    ]
    for port in ports {
        if canOpenLocalConnection(port: port) {
            return true
        }
  }
    return false
  }
```

Ako je frida-server pokrenut, aplikacija će prikazati u dijalogu da je aplikacija ugrožena kao na Slici 8.5.



Slika 8.5. Dijalog detekcije frida-servera.

Iako je ova detekcija pozitivan korak, programeri trebaju omogućiti raspršenu detekciju i proaktivne akcije uslijed detekcija poput gašenja aplikacije kako bi se napadaču zabranilo daljnje korištenje aplikacije.

9. Zaključak

Cilj ovog diplomskog rada je bilo istražiti moguće napade na mobilne sustave i predložiti načine obrane od njih. Prilikom istraživanja ustvrdile su se najčešće prijetnje poput *hooking*-a, *root*-anih ili *jailbreak*-anih uređaja, izvođenja u debug načinu rada te popularni načini obrane koji se sastoje od provjere integriteta aplikacije, obfuskacije koda i zaštite u realnom vremenu.

Sigurnost aplikacije je iznimno bitan aspekt razvoja mobilnih aplikacija te je potrebno obratiti pažnju na svaki aspekt zaštite aplikacije. Treba obratiti pozornost šifriranje osjetljivih podataka. Ponekad se podaci u aplikaciji mogu spremiti u lokalni datotečni sustav ili u vanjsku pohranu uređaja. Ti podaci nisu kriptirani, pa programeri moraju znati da tamo ne spremaju važne informacije. Preporuka je koristiti na iOS-u *Keychain*, siguran mehanizam za spremanje lozinki i osjetljivih informacija, a na Androidu koristi *KeyStore* za sigurno čuvanje ključeva za dešifriranje osjetljivih podataka.

Također operacijski sustavi imaju svoje ranjivosti, a napadači uvijek traže moguće rupe u operacijskim sustavima kako bi ih mogli iskoristiti. Zato je preporuka krajnjim korisnicima redovito ažurirati operacijski sustav.

Korisnici također mogu ugroziti operacijski sustav ukoliko ih odluče *jailbreak*-ati ili *root*-ati. Iako neki korisnici žele samo imati mogućnost veće slobode prilagođavanja uređaja s temama i drugim funkcionalnostima, kompromitirani telefon izlaže njihov mobilni uređaj velikom riziku jer se sigurnosne mjere iOS-a/Androida mogu lako ukloniti. Također su njihovi podaci i aplikacije koje sadrže važne informacije pod rizikom. Raznim tehnika se može doći do pristupa aplikacijama i informacijama. Također, napadaču su od iznimne koristi kompromitirani uređaji jer onda može lakše koristiti alate za manipulaciju aplikacijom.

Iako postoje propusti prilikom razvoja aplikacije gdje programer nepravilno koristi dostupne API-je koji omogućuju sigurnije ponašanje aplikacije, smatram da veći problem predstavljaju upravo alati za „petljanje“ s aplikacijama koji su dostupnim krajnjim korisnicima. Alati koji omogućuju manipuliranje podataka i tok izvođenja aplikacije postaju sve veća prijetnja te sam se u istraživanju najviše fokusirala na *frida-tools*. To je iznimno moćan alat koji ne ostavlja puno tragova za sobom, pogotovo na iOS-u, te smatram da je potrebno obratiti veću pažnju na tu prijetnju. U ovom radu su predstavljeni različiti načini rada frida alata, njihovi nedostatci i prednosti, te mogu biti od iznimne koristi programerima prilikom razvoja modela zaštite vlastite aplikacije.

Developeri trebaju obratiti veću pažnju na zaštitu aplikacije i probati ublažiti opasnosti i prijetnje aplikacijama. Iako se postave sve moguće zaštite za sigurnost aplikacije, uporni napadač i dalje može ostvariti svoj cilj. Glavni cilj je osigurati da to bude što teže tako da se uspostaviti što više mjera zaštita i rasporediti ih na različitim mjestima tijekom izvođenja programa.

10. Literatura

[1] Murat Yesilyurt, Yildiray Yalman : „Security Threats on Mobile Devices and their Effects: Estimations for the Future“, s Interneta, https://www.researchgate.net/publication/297746368_Security_Threats_on_Mobile_Devices_and_their_Effects_Estimations_for_the_Future, 1. lipnja, 2022.

[2] Wikipedia: „Mobile Device“, s Interneta, https://en.wikipedia.org/wiki/Mobile_device, 15. prosinca 2021.

[3] Santos Das: „Top 10 Mobile Phone Manufacturers in the world“, s Interneta, http://www.mobilecellphonerepairing.com/top-mobile-phone-manufacturers-in-the-world.html, 15. prosinca 2021.

[4] Jul Clover: „Apple Now Has More Than 1.8 Billion Active Devices Worldwide“, s Interneta, https://www.macrumors.com/2022/01/27/apple-1-8-billion-active-devicesworldwide/, 15. prosinca 2021.

[5] Statista Research Department: „Android - Statistics & Facts“, s Interneta, https://www.statista.com/topics/876/android/#dossierSummary__chapter2___, 30. svibnja 2022.

[6] Wikipedia: „Android (operating system)“, s Interneta, https://en.wikipedia.org/wiki/Android_(operating_system) , 7. siječnja 2022.

[7] Encyclopaedia Britannica: „Android“, s Interneta, https://www.britannica.com/technology/Android-operating-system, 7. siječnja 2022.

[8] Android: „Architecture“, s Interneta, https://source.android.com/devices/architecture , 7. siječnja 2022.

[9] Ankit Sinhal: „Closer Look At Android Runtime: DVM vs ART“, s Interneta, https://medium.com/android-news/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924, 6. lipnja 2022.

[10] Android: „Security“, s Interneta, https://source.android.com/security/overview, 7. siječnja 2022.

[11] Kal: „Introduction to Android Security“, s Interneta, https://medium.com/mobis3c/introduction-to-android-security-64609edeb18c__, 7. siječnja 2022.

[12] Wikipedia: „IOS“, s Interneta, https://en.wikipedia.org/wiki/IOS, 15. prosinca 2021.

[13] L. Ceci: „Number of apps available in leading app stores 2022“, s Interneta, https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 15. prosinca 2021.

[14] Wikipedia: „ARM architecture“, s Interneta, https://en.wikipedia.org/wiki/ARM_architecture, 16. prosinca 2021.

[15] Wikipedia: „XNU“, s Interneta, https://en.wikipedia.org/wiki/File:The_XNU_Kernel_Graphic.svg, 16. prosinca 2021.

[16] Wikipedia: „XNU“, s Interneta, https://en.wikipedia.org/wiki/XNU, 16. prosinca 2021.

[17] Wikipedia: „iBoot“, s Interneta, https://en.wikipedia.org/wiki/IBoot, 16. prosinca 2021.

[18] Apple: „iOS Security“, s Interneta, https://css.csail.mit.edu/6.858/2020/readings/ios-security-may19.pdf, 16. prosinca 2021.

[19] Apple: „Secure Enclave“, s Interneta, https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web, 16. prosinca 2021.

[20] vtky: „resign“, s Interneta, https://github.com/vtky/resign, 10. travnja 2022.

[21] Saroj Raut, s Interneta, https://stackoverflow.com/questions/40841670/what-is-the-difference-between-dylib-and-a-lib-in-ios, 18. prosinca 2021.

[22] Teodor MITREA, Monica BORDA: „MOBILE SECURITY THREATS: A SURVEY ON PROTECTION AND MITIGATION STRATEGIES“, s Interneta, https://sciendo.com/it/article/10.2478/kbo-2020-0127, 10. svibnja 2022.

[23] NIST: „Mobile ecosystem“, s Interneta, https://pages.nist.gov/mobile-threat-catalogue/background/mobile-attack-surface/mobile-ecosystem.html, 10. svibnja 2022.

[24] Mobile Security Testing Guide: „Mobile Security Testing Guide“, s Interneta, https://mobile-security.gitbook.io/mobile-security-testing-guide/overview/0x04b-mobile-app-security-testing, 10. travnja 2022.

[25] Apurv Pandey: „SSL Pinning in Android“, s Interneta, https://mailapurvpandey.medium.com/ssl-pinning-in-android-90dddfa3e051, 26. ožujka 2022.

[26] Nabla-c0d3: „How SSL Kill Switch works on iOS 12“, s Interneta, https://nabla-c0d3.github.io/blog/2019/05/18/ssl-kill-switch-for-ios12/ , 10. travnja 2022.

[27] Jan Seredynski: „A security review of 1,300 AppStore applications“, s Interneta, https://seredynski.com/articles/a-security-review-of-1300-appstore-applications.html, 5. ožujka 2022.

[28] Vikas Gupta: „Bypassing anti-debugger check in iOS Applications“, s Interneta, https://serializethoughts.wordpress.com/2018/01/23/bypassing-anti-debugger-check-in-ios-applications/, 12. ožujka 2022.

[29] Apple: „Mac OS X manual page“, s Interneta, https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/ptrace.2.html, 5. ožujka 2022.

[30] Jose Lopes: „Who owns your runtime?“, s Interneta, https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/?fbclid=IwAR2sQXDQ8gmcoYNXugDMuKo7DcaCiYzddgED4rZe6qOh6s43r394ePVRgv0, 10. travnja 2022.

[31] IOSSecuritySuite, s Interneta, https://github.com/securing/IOSSecuritySuite , 10. travnja 2022.

[32] Dennis Frett: „Prevent bypassing of SSL certificate pinning in iOS applications“, s Interneta, https://www.guardsquare.com/blog/ios-ssl-certificate-pinning-bypassing, 9. travnja 2022.

[33] hot3eed: „Reverse Engineering Starling Bank (Part II): Jailbreak & Debugger Detection, Weaknesses & Mitigations“, s Interneta, https://hot3eed.github.io/2020/08/02/starling_p2_detections_mitigations.html , 9. travnja 2022.

[34] Trustwave: „Jailbreak Detection Methods“, s Interneta, https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/jailbreak-detection-methods/, 9. travnja 2022.

[35] Apple: „Establishing Your App’s Integrity“, s Interneta, https://developer.apple.com/documentation/devicecheck/establishing_your_app_s_integrity, 9. travnja 2022.

[36] Shiva Saxena: „Mobile Security: How to establish your App's Integrity“, s Interneta, https://fueled.com/the-cache/posts/backend/security/how-to-establish-your-apps-integrity/, 9. travnja 2022.

[37] Apple: „Security“, s Interneta, https://developer.apple.com/security/, 5. ožujka 2022.

[38] Linux: „Linux manual page“, s Interneta, https://man7.org/linux/man-pages/man5/proc.5.html, 10. travnja 2022.

[39] Android: „SafetyNet Attestation API“, s Interneta, https://developer.android.com/training/safetynet/attestation, 2. travnja 2022.

[40] Alexey Alter-Pesotskiy: „How to download .ipa from App Store“, s Interneta, https://medium.com/xcnotes/how-to-download-ipa-from-app-store-43e04b3d0332, 12. travnja 2022.

[41] Konstantine Zuckerman, „Is penetration testing legal?“, s Interneta, https://cybri.com/is-penetration-testing-legal/, 14. lipnja 2022.

[42] Github: „frida-scripts“, s Interneta, https://github.com/interference-security/frida-scripts/blob/master/iOS/show-modify-method-return-value.js, 12. travnja 2022.

11. Popis oznaka i kratica

ROM – Read-Only Memory

ARM - Advanced RISC Machines

DFU - Device Firmware Update

SoC - system on a chip

AES - Advanced Encryption Standard

TLS - Transport Layer Security

API - Application Programming Interface

MAC address - a unique physical address assigned to each network adapter in a computer, or mobile device

IPC - interprocess communication

ART - Android Runtime

DVM - Dalvik Virtual Machine

JIT – Just-In-Time

TEE - Trusted Execution Environment

UUID - universally unique identifier

12. Popis slika i tablica

Slika 8.5. Dijalog detekcije frida-servera.

13. Prijevod engleskih pojmova

Third-party – treća strana

Open-source - otvoreni izvor

Secure boot – sigurno podizanje sustava

Sideloading – bočno učitavanje/instaliranje aplikacije

Low-level – niska razina

Root of trust – korijen povjerenja

Sandbox – u ovom kontekstu predstavlja zatvoreni sustav aplikacije

High-level – visoka razina

User-authentication-gated - provjere autentičnosti korisnika

User authenticator - autentifikator korisnika

Protected confirmation - zaštićena potvrda

Firmware - trajni softver programiran u ROM memoriju

Daemon - program koji radi kontinuirano i postoji u svrhu rukovanja povremenim zahtjevima za uslugu koje računalni sustav očekuje da primi

Sažetak

Sigurnost aplikacija je jedan od bitnijih aspekata razvoja mobilnih aplikacija kojeg bi programeri trebali biti svjesni. Dobrim poznavanjem mogućih prijetnji i njihovim ublažavanjem, razvijaju se sigurnije aplikacije. Najčešće prijetnje sigurnosti aplikacije su nesigurni komunikacijski kanali, neispravno korištenje lokalne pohrane, ugroženi operacijski sustav te obrnuti inženjering. Zadatak ovog rada je dati pregled mogućih opasnosti na Android i iOS platformi te ponuditi moguće mjere ublažavanja tih prijetnji.

Ključne riječi: prijetnje na mobilnim aplikacijama, obrnuti inženjering, sigurnost aplikacije, zaštita, ublažavanje prijetnji

Abstract

Application security is one of the most important aspects of mobile application development that developers should be aware of. With a good knowledge of possible threats and their mitigations, more secure applications are being developed. The most common threats to application security are insecure communication channels, improper use of local data protection, compromised operating system, and reverse engineering. The task of this paper is to provide an overview of possible dangers on the Android and iOS platforms and to offer possible measures to mitigate these threats.

Keywords: threats on mobile applications, reverse engineering, application security, protection, threat mitigation