Penetracijsko testiranje iOS aplikacija

Neven, Travaš

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Neven Travaš

PENETRACIJSKO TESTIRANJE IOS APLIKACIJA

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Neven Travaš

Matični broj: 0016117877

Studij: Baze podataka i baze znanja

PENETRACIJSKO TESTIRANJE IOS APLIKACIJA

DIPLOMSKI RAD

:

Doc. dr. sc. Tomičić Igor

Varaždin, rujan 2021.

Neven Travaš

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U radu će biti prikazana metodologija penetracijskog testiranja iOS aplikacija. Prikazat će se uobičajene ranjivosti sustava te tehnike i alati za iskorištavanje istih. Predstaviti iOS operacijski sustav te obraditi "White Box" i "Black Box" testiranje iOS aplikacije Rad će obuhvatiti i poveznicu prema realnom sektoru u kontekstu istraživanja odabranih praksi penetracijskog testiranja, a u praktičnom dijelu rada prikazat će se konkretni proces penetracijskog testiranja iOS aplikacije na primjeru OWASP iGoat i DVIA aplikacije.

Ključne riječi: iOS, penetratcijsko testiranje, sigurnost mobilnih uređaja, white box testiranje, black box testiranje, iOS ranjivosti, OWASP iGoat, DVIA

Sadržaj

1. Uvod

Pametni telefoni danas predstavljaju neizostavni dio svakodnevice i poslovanja svakog pojedinca. Toliko su postali sveprisutni da je više nemoguće zamisliti svijet bez njih. Koriste se za informiranje o raznim temama, navigaciju u prostoru, međusobnu komunikaciju i poslovanje, zabavu i rekreaciju te za identifikaciju i autorizaciju pojedinca. Međutim svaka nova tehnologija sa sobom nosi i nove sigurnosne rizike. Koliko god mobilni uređaji bili slični klasičnim računalnim sustavima i njihove aplikacije klasičnom softwareu, sigurnosni rizici s kojima se suočavaju često su drugačiji od rizika koji su primijećeni kod klasičnih računalnih ili web aplikacija. Tako primjerice klasična skeniranja potpisa poznatih virusa, koje provode većina antivirusnih programa kako bi uočili postojanje virusa u računalu, nema previše smisla za mobilne aplikacije, pošto se iste pokreću u svom sigurnom okruženju (engl. *sandbox*). Većina se aplikacija izvršava nativno na uređaju s podacima koji se nalaze na uređaju tako da su u jednu ruku i manji postoci mogućih napada kao što su XSS (engl. *Cross Site Scripting*). Buffer overflow napadi često isto ne daju željene rezultate zbog ograničenih resursa koje su aplikaciji dostupni od strane operacijskog sustava, naravno i za XSS i za buffer overflow napade postoje iznimke, odnosno aplikacije kod kojih isti mogu biti provedeni.

Glavna ideja mobilne sigurnosti zasniva se na sigurnosti podataka korisnika pojedinog uređaja ili aplikacije. Pošto se mobilne aplikacije u većini slučajeva oslanjaju na neke vanjske servise za dohvat ili obradu podataka, potrebno je osigurati ispravan i siguran način slanja i primanja poruka, što od strane mobilne aplikacije kao klijenta koji vrši poziv, što od strane udaljenog servera koji obrađuje poslane podatke i vraća rezultat na uvid korisniku. Podaci koji se uglavnom spremaju i koriste na mobilnim uređajima i unutar aplikacija su osobni podaci korisnika. U to ulaze osobne slike, bilješke, identifikacijski i autorizacijski podaci, lozinke i poslovni podaci. Svi ti podaci trebaju biti vezani isključivo uz pojedinca i u većini slučajeva ne smiju biti dostupni izvan okvira uređaja ili aplikacije.

U današnje vrijeme glavni mobilni operacijski sustavi iOS i Android imaju dobro definirane servise i metode za sigurnu pohranu podataka na mobilnim uređajima. Međutim i dalje je na razvojnim inženjerima pojedinih aplikacija da pravilno i efektivno upotrijebe dostupne alate (servise, metode, najsigurnije oblike kriptografije) kako bi pospješili mrežnu i internu sigurnost aplikacije te je učinili uistinu zaštićenom. U tom pogledu istražit će se sljedeći slučajevi:

- Načini verifikacije serverskih certifikata koristeći TLS certifikate (engl. *Certificate Pining*).
- Kriptiranje podataka spremljenih unutar lokalne baze podataka na uređaju.
- Provjera onesposobljavanja sigurnosnih mjera postavljenih od strane proizvođača operacijskog sustava ("jailbreaking")
- Analiza rukovanja aplikacije s tipom i količinom povjerljivih podataka temeljem čega se uspostavlja sigurnosni stupanj aplikacije.

Ovaj rad će pokušati odgovoriti na ta pitanja i proći kroz najbolju praksu osiguravanja iOS aplikacija te prikazati načine zloupotrebe sigurnosnih propusta kako bi se od istih moglo zaštititi.

2. Metode i tehnike rada

Rad se oslanja na slobodno dostupne iOS aplikacije iGoat i DVIA koje predstavlja ranjivu iOS aplikaciju nad kojima će se provesti penetracijsko testiranje. Opisat će se pojedini procesi i metode rada penetracijskog testiranja te objasniti koncepti "White Box" i "Black Box" testiranja. Penetracijsko testiranje provesti će se kroz sljedeće četiri faze:

- Otkrivanje
- Analiza
- Eksploracija
- Izvještavanje

Otkrivanje - Sadrži sakupljanje podataka i definiranje opsega testiranja. Sakupljanje podataka sastoji se od sakupljanja javno dostupnih podataka te internih dokumentacija i izvornog koda, s ciljem svladavanja i razumijevanja arhitekture aplikacije te definiranja i odvajanja klijentskih i serverskih sigurnosnih scenarija.

Analiza - Dijeli se na statičku i dinamičku analizu. Statička analiza odnosi se na pregled dokumentacije i izvornog koda prije samog pokretanja aplikacije. Ovisno o dostupnosti izvornog koda može se odnositi i na analizu podataka dobivenih iz reverzibilnog inženjeringa pojedine aplikacije. Dinamička analiza odnosi se na ponašanje aplikacije nakon njenog pokretanja. Osim analize same aplikacije potrebno je analizirati i mrežni promet kojeg aplikacija prima i šalje.

Eksploatacija - Faza unutar koje tester pronalazi i pokušava iskoristiti sigurnosne propuste. Nadalje gleda na koji način pronađeni propusti utječu na sigurnost korisnikovih podataka i na sam rad aplikacije.

Izvještavanje - Finalna faza penetracijskog testiranja, kao što joj i ime govori radi se o fazi sastavljanja izvještaja za naručitelja penetracijskog testiranja. U izvještaju tester zapisuje svoje pronalaske, definira njihovu opasnost i štetnost na sam rad aplikacije i na poslovanje poduzeća koje aplikaciju stavlja na tržište. Osim definiranja pronađenih problema pojašnjava kako reproducirati iste te nudi prijedloge s kojima bi se problemi mogli riješiti.

Osim aplikacija iGoat i DVIA u radu će se koristiti alati Charles Proxy, Frida, Objection te *jailbreakani* iPhone 6s.

3. Tipovi sigurnosne analize

U sljedećim sekcijama opisat će se generalni postupci koji se provode prilikom sigurnosnog testiranja iOS aplikacija. Slični su standardnim sigurnosnim testiranjima. Ovisno o pristupnosti izvornom kodu i razvojnoj dokumentaciji aplikacije koja se analizira razlikovat će se "White Box" i "Black Box" testiranja, a nad samom aplikacijom provodi se statička i dinamička analiza te sigurnosna verifikacija. Sigurnosna verifikacija razlikuje se ovisno o tome s kojim podacima, koje razine povjerljivosti aplikacija upravlja.

3.1. Black Box testiranje

Provodi se bez da tester ima ikakva prijašnja znanja o aplikaciji, njenoj strukturi i sastavu. Glavna ideja ovog tipa testiranja je da se tester ponaša kao pravi napadač i istraži moguće vektore napada na aplikaciju, koristeći javno dostupne podatke o aplikaciji. Osim što se koristi javno dostupnim podacima dublji uvid u aplikaciju pokušava steći i postupcima reverzibilnog inženjerstva, tako pokušava iskoristiti greške nemarnog developera koji možda nije sve potrebne podatke zaštitio na ispravan način. [1]

3.2. White Box testiranje

Kao što i ime nalaže radi se o potpunoj suprotnosti u odnosu na "Black Box" testiranje. Kod "White Box" testiranja tester ima pun uvid u aplikaciju. Poznaje njenu strukturu i dobro je upoznat sa svim funkcionalnostima i njihovom izvedbom. Informacije dobiva iz razvojne dokumentacije i razvojnih dijagrama ili samog izvornog koda koji je testeru dan na raspolaganje. Ako tester izvodi "White Box" testiranje dovoljno je da prođe kroz izvorni kod i provjeri jesu li svi podaci dobro zaštićeni, postoje li slučajevi u kojima se pozivaju metode koje nemaju poputno pokrivene rubne slučajeve.[1]

3.3. Analiza ranjivosti

Predstavlja proces traženja i pronalaska mogućih ranjivosti unutar aplikacije. Prvi je korak pri sigurnosnom testiranju neke aplikacije, a sastoji se od statične i dinamičke analize. Može se provoditi ručno ili pomoću gotovih alata za analizu. [2]

3.3.1. Statička analiza ranjivosti

Statička analiza ranjivosti uključuje analizu aplikacijskih komponenti bez pokretanja same aplikacije. Provodi se kod white-box testiranja i uključuje pregled izvornog koda aplikacije. Tehnike koje se koriste pri statičkoj analizi koda uključuju Taint analizu i Analizu toka podataka (engl. *Data Flow Analysis*) [1]

Taint analiza bazira se na provjerama korisničkog unosa, odnosno provjeru metoda na koje korisnik direktno utječe sa svojim akcijama. Provjerava se da li su svi mogući slučajevi pokriveni ovisno o parametrima koje korisnik može unijeti ili postoji mogućnost da korisnik iz vana može maliciozno djelovati na izvršenje programa iskorištavanjem nepokrivenog rubnog slučaja.[3]

Analiza toka podataka predstavlja proces analize hijerarhije objekata unutar aplikacije i prijenosa poruka, a time i podataka između objekata unutar aplikacije, ali i prenošenje pojedinih podataka u obliku objekta preko mreže prema nekoj trećoj strani. Kroz analizu toka podataka moguće je primijetiti curenje povjerljivih podataka, odnosno na kojim bi mjestima bilo potrebno upozoriti korisnika da njegova trenutna akcija može prouzročiti prijenos povjerljivih informacija preko neke mreže, najčešće interneta ka nekoj trećoj strani. [4]

Prednosti statičke analize uključuju: Dobro skaliranje i jednostavno pronalaženje sigurnosnih propusta poput buffer overflowa, SQL injectiona i sličnih. Dok mane uključuju: Ne uzimanje u obzir problema kod autentifikacije ili kontrole pristupa pojedinom resursu, visoki postotak lažnih negativa, nemogućnost upućivanja na curenje podataka te ne nalaženje propusta unutar konfiguracije pošto ista često nije dostupna iz koda.

3.3.2. Dinamička analiza ranjivosti

Dinamička analiza ranjivosti podrazumijeva pokretanje aplikacije i njenu analizu pri izvođenju. Može se provesti automatizirano ili manualno. Izvodi se na mobilnom klijentu i na serveru. Provjera na serveru često se gleda kroz pozive i odgovore pojedinih servisa koje aplikacija koristi. Kroz dinamičku analizu dobiva se uvid u podatke pri tranzitu iz i u aplikaciju, moguće greške u autorizaciji i autentifikaciji unutar aplikacije te greške na serveru. [1]

3.4. Sigurnosna autentifikacija i verifikacija

Službeni sigurnosni standard mobilnih aplikacija (engl.*Mobile Application Security Verification Standard*) definira dvije razine sigurnosne verifikacije (MASVS-L1 i MASVS-L2), kao i potrebne mjere za otpornost aplikacija na reverzibilni inženjering (MASVS-R). MASVS-L1 sadrži uobičajene sigurnosne regulative čije se korištenje savjetuje kod implementacije svih mobilnih aplikacija dok se MASVS-L2 odnosi na aplikacije koje koriste osjetljive podatke.

Ukoliko aplikacija korisniku omogućava spajanje na neki udaljeni server, a sama ne sadrži ili ne izmjenjuje nikakve povjerljive podatke sa serverom, dovoljno je autentifikaciju korisnika vršiti putem korisničkog imena i lozinke. Sama verifikacija korisnika provodi se na serveru ovisno o unesenim podacima. Na serveru je postavljen oblik politike prema kojoj je definirana valjana lozinka, ako korisnik pretjerani broj puta (konkretno definirano od strane pružatelja usluge) pogriješi lozinku najčešće se korisnikov račun zaključa na određeni vremenski period. Takav proces autentifikacije i verifikacije korisnika klasificira se kao MASVS-L1.

Aplikacija koja izmjenjuje bilo kakve povjerljive podatke sa serverom ili vrši neki oblik transakcije klasificira se kao MASVS-L2 te je takva aplikacija dužna implementirati dvo fak-

tornu autentifikaciju, nadalje u tekstu označena sa 2FA (engl.*Two Factor Authentication*). 2FA upozorava korisnika tijekom prijave ili provedbe transakcije te traži dodatnu autorizaciju u obliku otp-a koji se korisniku šalje nekim drugim komunikacijskim kanalom kojeg je naveo da mu odgovara. Tek kada se verificira sigurnosni kod zajedno sa podacima za prijavu korisnika, uspješno se verificira i sam korisnik te mu se dozvoljava provedba transakcije odnosno pristup aplikaciji. Međutim ne postoji izričito jedan način za testiranje mehanizma autentifikacije pošto ista može biti provedena na više načina. Primjerice: lozinka, PIN, ili uzorak kojeg korisnik prepoznaje. Nešto što korisnik posjeduje poput SIM kartice, otp-a (engl.*One Time Password*) ili fizičkog token uređaja koji generira sigurnosni kod. Također autentifikacija se može vršiti i preko jedinstvenih biometrijskih karakteristika svakog individualnog korisnika, kao što su otisak prsta, sken retine oka, glas ili prepoznavanje lica.[1]

Kada se radi testiranje autentifikacije potrebno je identificirati sve autentifikacijske faktore koje aplikacija koristi, locirati sve konačne točke koje omogućuju bitne funkcionalnosti aplikacije i ustanoviti da su svi faktori za autentifikaciju implementirani na svim serverima s kojima aplikacija komunicira i da se pokreću nebitno o dijelu aplikacije koja je napravila upit na server.

Ranjivosti koje se odnose na autentifikaciju pojavljuju se kada stanje autentifikacije nije jedinstveno kontrolirano na serveru, već klijent može utjecati i mijenjati stanje iste. Server bi trebao kod svakog poziva od strane klijenta vršiti provjeru kako bi u svakom trenutku znao da li je ispravan korisnik logiran i da li je isti autoriziran za preuzimanje podataka. [1]

3.4.1. Statička analiza autentifikacije

Kod provedbe statičke analize autentifikacije prvo se provjerava postoji li sigurnosna politika za autentifikaciju, odnosno po kojim pravilima korisnik mora zadati lozinku i da li se ta pravila poštuju u aplikaciji. Zatim je potrebno pronaći sve metode koje su povezane s lozinkom i utvrditi da li se u njima provodi verifikacija iste. Navedene korake potrebno je provjeriti na klijentu i na serveru. Osim same verifikacije lozinke, na serveru je dodatno potrebno provjeriti ispravnu odjavu pojedinog korisnika ovisno o načinu praćenja istog[1], pri čemu je bitno provjeriti da li su podaci pomoću kojih se korisnik prijavio uspješno uništeni.[1]

3.4.2. Dinamička analiza autentifikacije

Provodi se prijavom u aplikaciju i pokušajem dohvaćanja resursa za kojeg korisnik mora biti autoriziran i verificiran. Ako korisnik nije autoriziran ne bi smio doći do resursa isto tako ako korisnik nije verificiran ne bi smio imati pristup traženom resursu.[1]

[1] "stateless" - praćenje korisnika preko klijentskog kolačića ili "stateful" - praćenje korisnika preko korisničke sesije

4. Penetracijsko testiranje

Kao što je već navedeno u poglavlju "Metode i Tehnike rada" penetracijsko testiranje provodi se u četiri glavne faze:

- Otkrivanje
- Analiza
- Eksploatacija
- Izvještavanje

Sve faze već su ukratko opisane u prijašnjem poglavlju tako da će ovdje biti samo nadopunjene. U fazi otkrivanja definira se opseg testiranja. Opseg se sastoji od: definiranja testova koji će biti provedeni nad aplikacijom, koji će podaci biti dostupni u aplikaciji te koje pretpostavke organizacija želi utvrditi ili opovrgnuti sigurnosnim testiranjem aplikacije. Definiranje testova odnosi se na konkretne testove koji će se provoditi, primjerice: provjera *jailbreaka* i SSL pininga, pokušaj provedbe SQL injectiona i slično. Ovisno o korisničkim podacima s kojim aplikacija upravlja, zahtijeva li aplikacija dvo faktorsku autentifikaciju i autorizaciju ili ne te koliki stupanj rizika pojedini propusti predstavljaju organizaciji koja razvija aplikaciju. Također bitna stavka pripreme je testeru dati pravo i autorizirati ga za penetracijsko testiranje aplikacije, pošto se isto bez privole autora aplikacije može smatrati kaznenim djelom. Zatim slijedi analiza u koju primarno spada sakupljanje i klasifikacija sakupljenih podataka. Sakupljanje podataka sastoji se od promatranja ponašanja aplikacije i njenog ustroja, kako bi se dobio što bolji uvid u način rada aplikacije. Bitno je usredotočiti se na definirana ponašanja i analizirati kako će se aplikacija ponijeti u slučaju kršenja koraka provedbe definiranih ponašanja ili pri pokušaj zaobilaženja istih. Koraci provedbe definiranih ponašanja odnose se na korištenje aplikacije na način na koji je to zamislio njen kreator. Dok se zaobilaženje ili kršenje istih odnosi na pokušaj izazivanja ne predvidivog i ne ispravnog toka aplikacije. Ustroj aplikacije podrazumijeva arhitekturu aplikacije. Odnosno, na koji način se izmjenjuju podaci unutar aplikacije i definiraju korisničke sesije te da li aplikacija u sebi ima ugrađenu *jailbreak* detekciju. Pri tome se ustanovljuje točna verzija operacijskog sustava koji se izvršava na uređaju i ako se aplikacija spaja na internet, koristi li TLS. Analiziraju se kriptografske procedure i algoritmi koji se koriste za enkripciju podataka koji se šalju internetom te provjerava koristi li aplikacija tako zvani "certificate pining" za verifikaciju konačnog odredišta podataka. Osim ustroja same aplikacije treba obratiti pažnju i na ustroj servera i udaljenih servisa koje aplikacija koristi. Potrebno je provjeriti jesu li isti na neki način kompromitirani i može li to utjecati na sigurnost same aplikacije.

Nakon analize slijedi mapiranje aplikacije s prikupljenim podacima iz prethodne faze, odnosno ustanovljuju se ulazne točke u aplikaciju, tok podataka i izlazne točke. Mapirana aplikacija omogućava otkrivanje potencijalnih slabosti te rangiranje istih prema tome koje predstavljaju najveći sigurnosni rizik. U ovu fazu ulazi i kreiranje testova koji će se kasnije koristiti pri eksploataciji aplikacije. Potencijalni sigurnosni propusti rangiraju se prema: šteti koja može nastati za poduzeće iskorištavanjem primijećenog propusta, mogućem broju i jednostavnosti

reprodukcije ranjivosti, jednostavnosti iskorištavanja pojedine ranjivosti za napad, pogođenom broju korisnika iskorištavanjem pojedine ranjivosti za napad i jednostavnošću otkrivanja ranjivosti.

Konačno slijedi eksploatacija, faza u kojoj tester pokušava "probiti" aplikaciju, odnosno iz nje izvući željene podatke, poremetiti rad aplikacije ili u konačnici srušiti i onesposobiti aplikaciju. Pri tome se koristi ranjivostima pronađenim u koraku mapiranja i ustanovljuje utječu li primijećene ranjivosti doista na sigurnost i rad aplikacije.

Posljednji korak penetracijskog testiranja je izvještavanje. Sastoji se od sastavljanja izvješća klijentu u kojem tester obznanjuje ranjivosti koje je pronašao, kojim metodama ih je pronašao, kako ih se uspjelo iskoristiti, na kojim mjestima ih je pronašao i u kojem opsegu su primijećeni u aplikaciji. Također daje do znanja jesu li ranjivosti pronađene zahvaljujući informacijama koje je dobio od strane klijenta ili ih je sam otkrio. Za kraj ovisno o svom znanju i upoznatosti s testiranom tehnologijom, daje povratnu informaciju kako ispraviti iskorištenu ranjivost ili savjet na što se razvojni tim mora fokusirati kako bi ispravio sigurnosni propust.

5. Sigurnosni model iOS operacijskog sustava

Jezgra iOS-a bazirana je na Darwinu, open source Unix operacijskom sustavu kojeg je razvio Apple. Sve iOS aplikacije pokreću se u ograničenom okruženju (engl. *sandbox*) i međusobno su odvojene na razini datotečnog sustava, zbog čega imaju ograničen pristup sistemskim servisima. Apple je stvorio "App Store" kao jedinstveni portal/mjesto s kojeg dozvoljava preuzimanje aplikacija za njihove uređaje i tako smanjio mogućnost malicioznih aplikacija na iOS uređajima. Kroz "App Store" kontrolira koje se aplikacije smiju instalirati na iOS uređajima i po potrebi zabranjuje njihovo preuzimanje i pokretanje.

Iako Apple ulaže veliki napor u osiguravanje svojeg sustava i aplikacija sigurnosni propusti i dalje su mogući, pogotovo kod dijelova na koje direktno utječu developeri aplikacija. U to ulaze zaštita podataka korisnika, pravilno zapisivanje i čitanje iz keychaina, pravilna autentifikacija pomoću otiska prsta ili skena lica, mrežna sigurnost i slično.[1]

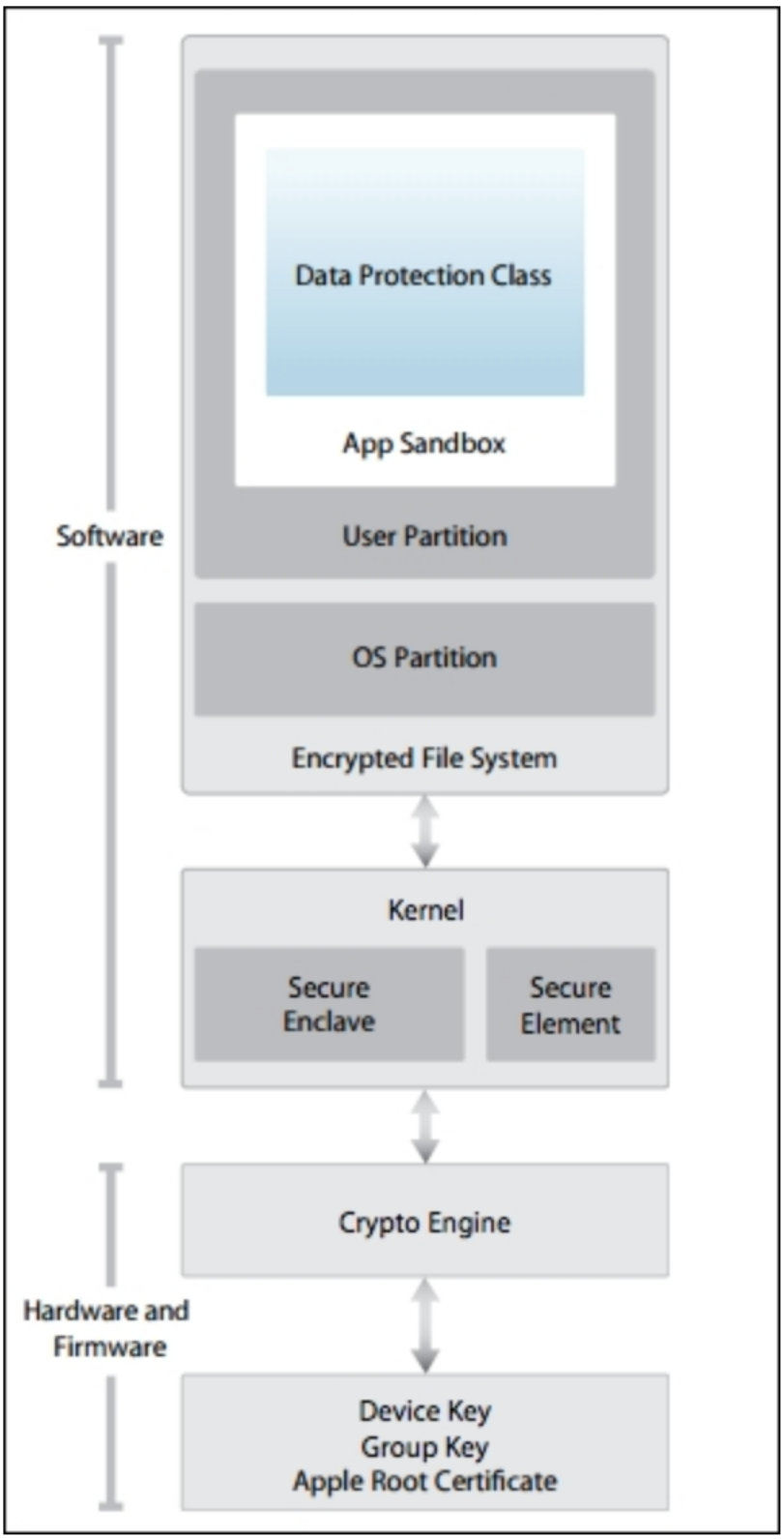
Apple razlikuje osam područja sigurnosti iOS-a:

- Sigurnost sustava - integrirani software i hardware koji su platforma za uređaje iPhone, iPad, i iPod touch.
- Enkripcija i zaštita podataka - arhitektura i dizajn koji štiti korisničke podatke ako je uređaj ukraden ili ako neautorizirana osoba pokuša modificirati podatke.
- Aplikativna zaštita - sustav koji omogućuje sigurno pokretanje i izvođenje aplikacija, bez kompromitiranja rada sustava (integritet sustava ostaje netaknut).
- Mrežna sigurnost - standardni mrežni protokoli koji osiguravaju sigurnu autentifikaciju i enkripciju podataka prilikom njihova prijenosa.
- Internet servisi - Apple-ova mrežna infrastruktura za slanje poruka, backup i sinkronizaciju.
- Apple Pay - Apple-ova implementacija sigurnog plaćanja.
- Kontrola uređaja - metode koje sprječavaju neautorizirano korištenje uređaja i daljinsko brisanje podataka ako je uređaj ukraden/izgubljen.
- Kontrola privatnosti - kontrola pristupa servisima koji se tiču lokacije i korisničkih podataka.

Opisi istih se obnavlja nakon izlaska svake veće nadogradnje iOS operacijskog sustava i dostupan je na (`https://support.apple.com/guide/security/welcome/web`).

5.1. Sigurnosna arhitektura

Appleovu sigurnosnu arhitekturu i međusobnu komunikaciju unutar nje prikazuje sljedeći dijagram:



Slika 1: Sigurnosni diagram iOS uređaja (Izvor: Apple, 2021)

Sigurnosna arhitektura iOS sustava započinje sa strukturom hardvera. Svaki iOS uređaj dolazi s dva ugrađena AES (engl. *Advanced Encryption Standard*) 256 - bitna ključa. Jedins-

tvenim identifikatorom uređaja (UID - engl. *Unique identifier*) i grupnim identifikatorom uređaja (GID - engl. *Group identifier*) koji su zapisani u aplikacijskom procesoru i sigurnosnoj enklavi. Ključevi su zaštićeni tako da ih nije moguće iščitati bilo kakvim softverom, već operacije enkripcije i dekripcije provode hardverske kriptografske jedinice koje imaju isključiv pristup ključevima. GID se koristi kako bi spriječio manipulaciju s firmwareom i ostalim kriptografskim operacijama koje se provode na uređaju, a nisu direktno vezane uz korisničke podatke, dok se UID koristi za zaštitu hijerarhije ključeva koji se koriste za enkripciju datotečnog sustava uređaja.

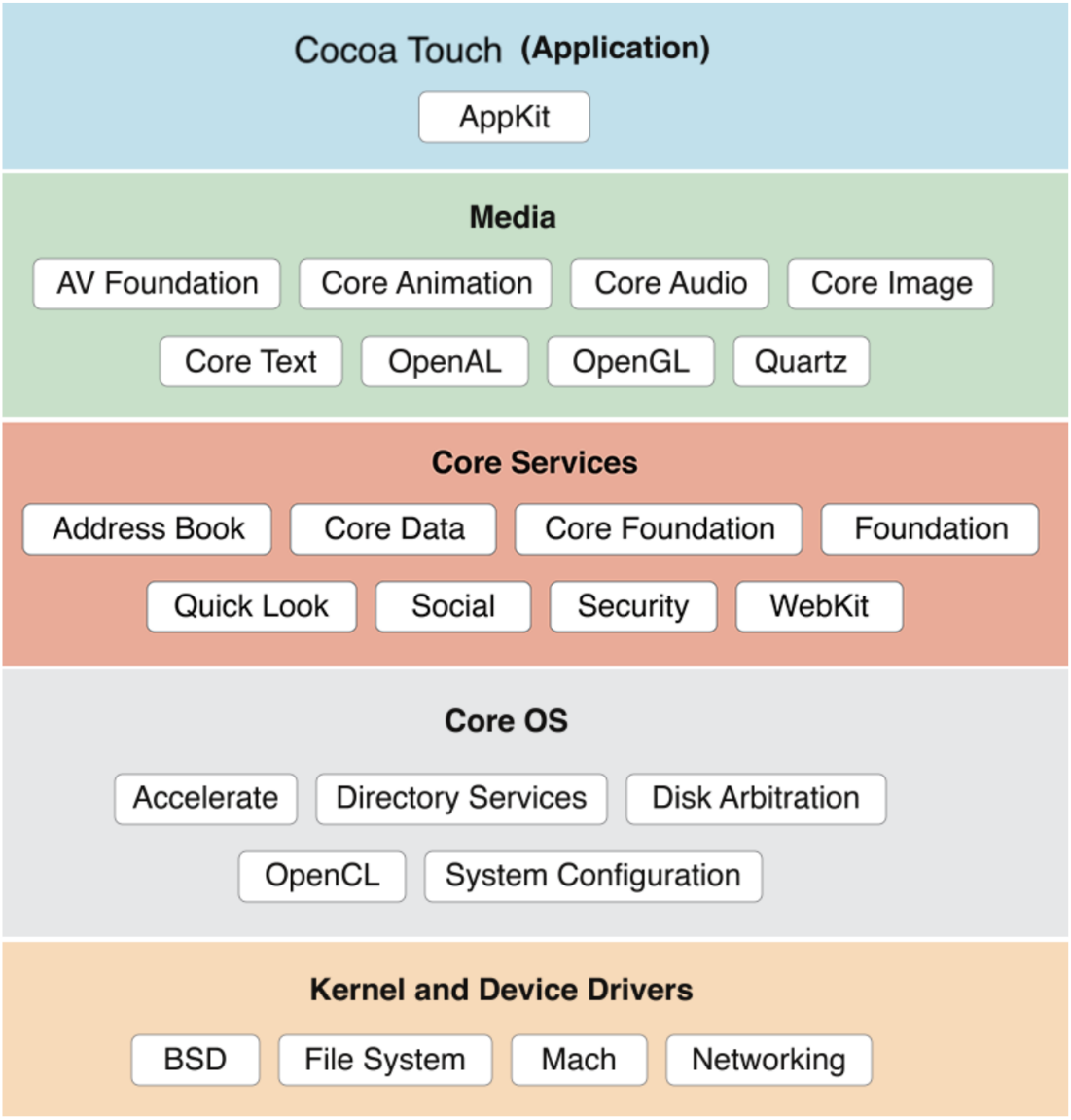
Nakon osiguravanja hardvera slijedi osiguravanje pokretanja sustava (engl. *Secure Boot*). Nakon što se pokrene, iOS uređaj učita instrukcije zapisane u ROM-u. ROM memorija sadrži nepromjenjive početne instrukcije i Apple Root certifikat. Slijedi provjera ispravnosti LLB-ovog (engl. *Low Level Bootloader*) certifikata. Ispravan LLB provjerava iBoot certifikat, koji provjerava sljedeći u nizu certifikat, a to je iOS kernel. Tek kad se ustanovi da su svi certifikati ispravno potpisani počinje proces pokretanja uređaja. Zakaže li samo jedna provjera uređaj odlazi u tako zvani "recovery mode", pri čemu ga je potrebno spojiti na računalo i provesti vraćanje istog na tvorničke postavke. Cijeli navedeni proces naziva se Sigurnosni pokretački lanac (engl. *Secure Boot Chain*). Proces se provodi kako bi se osiguralo da su sve komponente uređaja ispravne i originalno potpisane odnosno stavljene na tržište od strane Applea.

Uspješnim pokretanjem uređaja prelazi se u sljedeći korak sigurnosnih mjera, a to je potpisivanje koda koji se pokreće na uređaju. Svaka aplikacija koja se pokreće na iOS uređaju kako bi radila mora biti potpisana od strane Applea. Svaka aplikacija koju korisnik instalira na svoj uređaj mora biti preuzeta s App Storea koji je Appleov službeni prostor za preuzimanje i davanje aplikacija na uporabu.

Unutar iOS-a svaka aplikacija provodi se u svom zasebnom kontejneru zvanom "sandbox". Sandboxing odnosno kontejnerizacija aplikacija osigurava kontrolu pristupa unutar iOS-a. Time se limitira mogućnost oštećenja sistemskih i korisničkih podataka ako dođe do kompromitacije pojedine aplikacije. Skoro sve aplikacije koje se ikad pokreću unutar iOS operacijskog sustava pokreću se kao "mobile user", koji ima definirana ograničenja i kontrolu pristupa. Samo se pojedine sistemske aplikacije pokreću kao root. Tako sandbox kontrolira pristup resursima kojima aplikacija može pristupiti. Svi procesi unutar aplikacije ograničeni su na svoj vlastiti direktorij, koji može biti `/var/mobile/Containers/Bundle/Application/` ili `/var/containers/Bundle/Application/`, ovisno o verziji iOS-a. Nadalje naredbe poput `nmap` i `mmprotect` modificiraju sistemske naredbe kako bi onemogućili egzekuciju memorijskih stranica koje aplikacija zapisuje. Također sprječavaju provedbu dinamički generiranog koda, čime sprječava "code injection" koji je česta mana web-a. Dodatne predostrožnosti definirane su međusobnim odvajanjem svih aktivnih procesa i ograničavanjem pristupa hardverskim komponentama. Svaki aktivni proces unutar uređaja je zaseban i ne može se kreirati novi iz već postojećeg, iako svi na razini operacijskog sustava sadrže jedan te isti UID, a hardverskim komponentama uređaja nije moguće pristupiti direktno, već samo preko javno dostupnih servisa koje Apple stavlja na raspolaganje.

Ovisno o razini do koje developer želi dospjeti pristupa metodama i svojstvima pojedinih razvojnih okvira unutar određenog sistemskog sučelja. Najniže sučelje s kojim developer može

komunicirati je "CoreOS" sučelje, ono sadrži razvojne okvire za pristup značajkama na kojima se temelje i većina drugih tehnologija, ne nužno samo iOS.



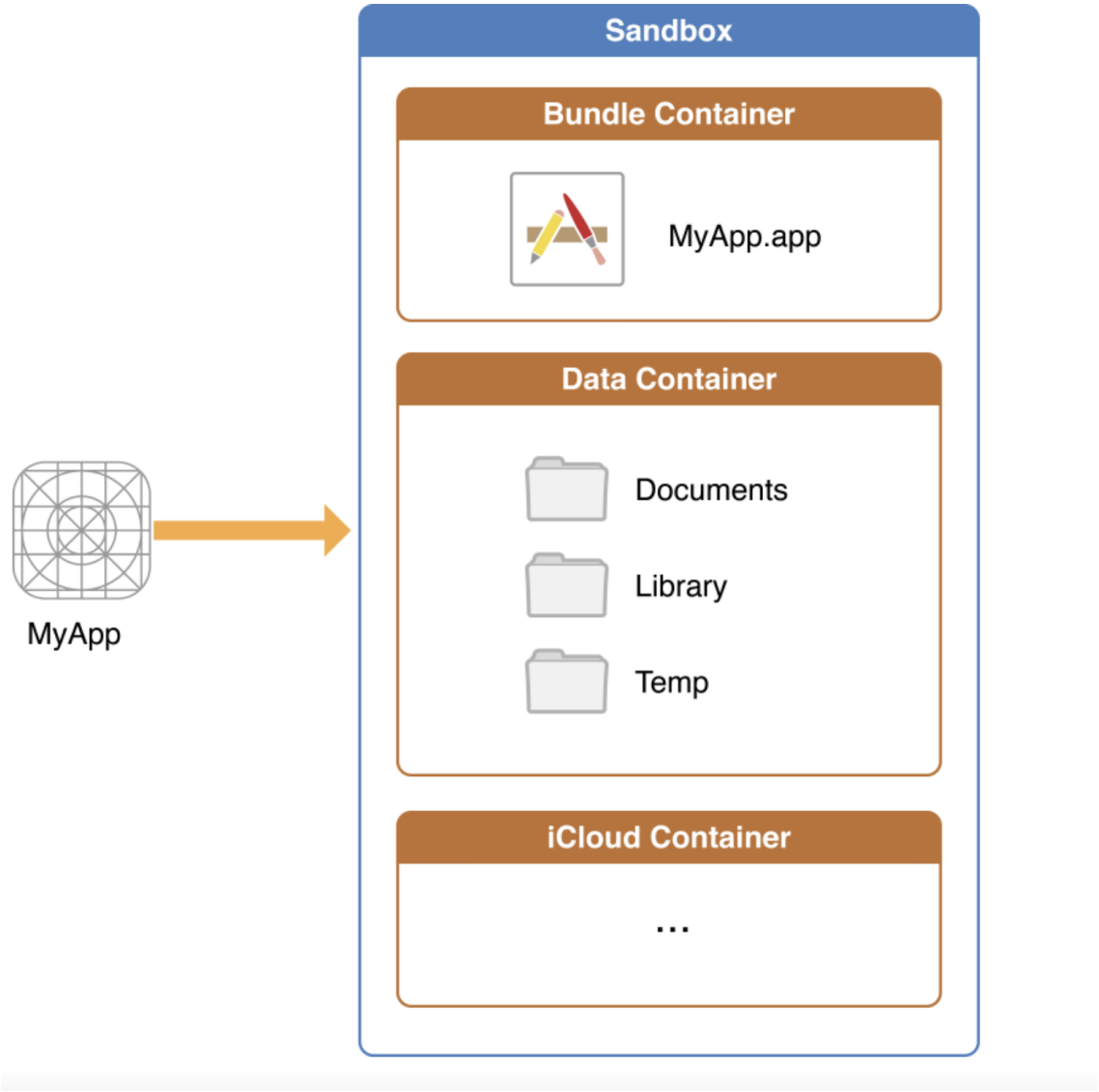
Slika 2: iOS sistemska sučelja (Izvor: Chauhan, 2017)

Unutar svakog sistemskog sučelja, koji je na slici prikazan u svojoj boji, nalaze se razvojni okviri koji developeru omogućuju manipulaciju podacima u pojedinim dijelovima OS-a.

Dodatna zaštita protiv "code injection" napada je i nasumično dodjeljivanje adresnog prostora koje se provodi prilikom svakog pokretanja OS-a te XN bit (engl. *execute never*), pomoću kojeg se pojedinim memorijskim segmentima tj. stranicama zabranjuje mogućnost pokretanja (isti se označavaju kao engl. *non-executable*).

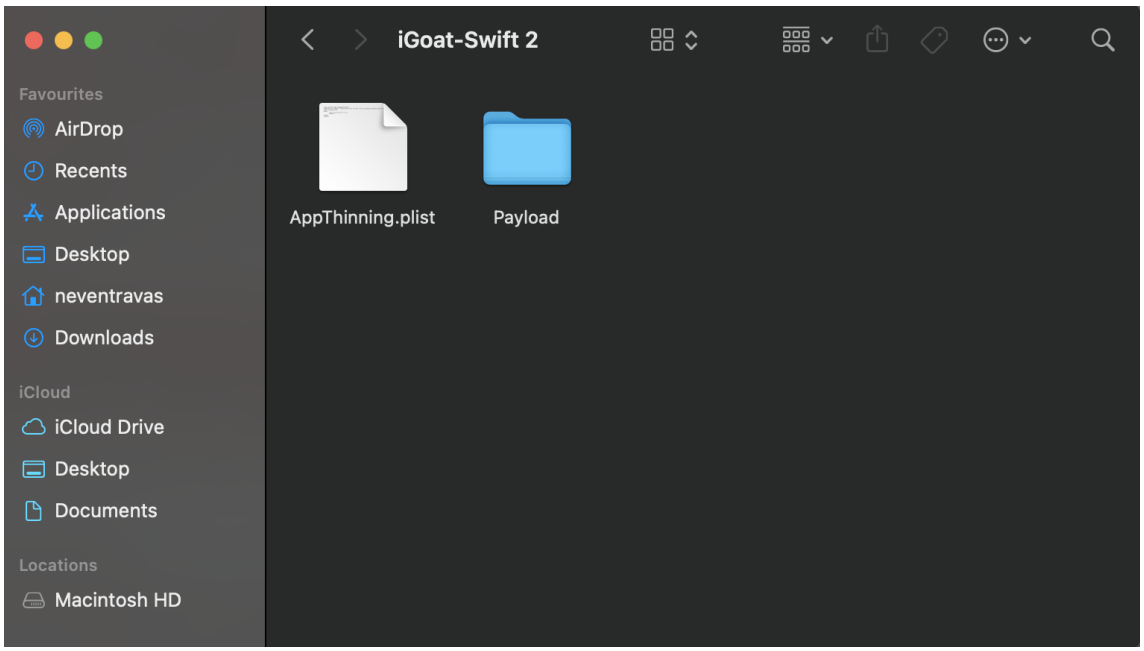
5.2. Struktura iOS aplikacija

Unutar svakog aplikacijskog sandboxa nalaze se *Bundle*, *Data* i *iCloud* kontejner.

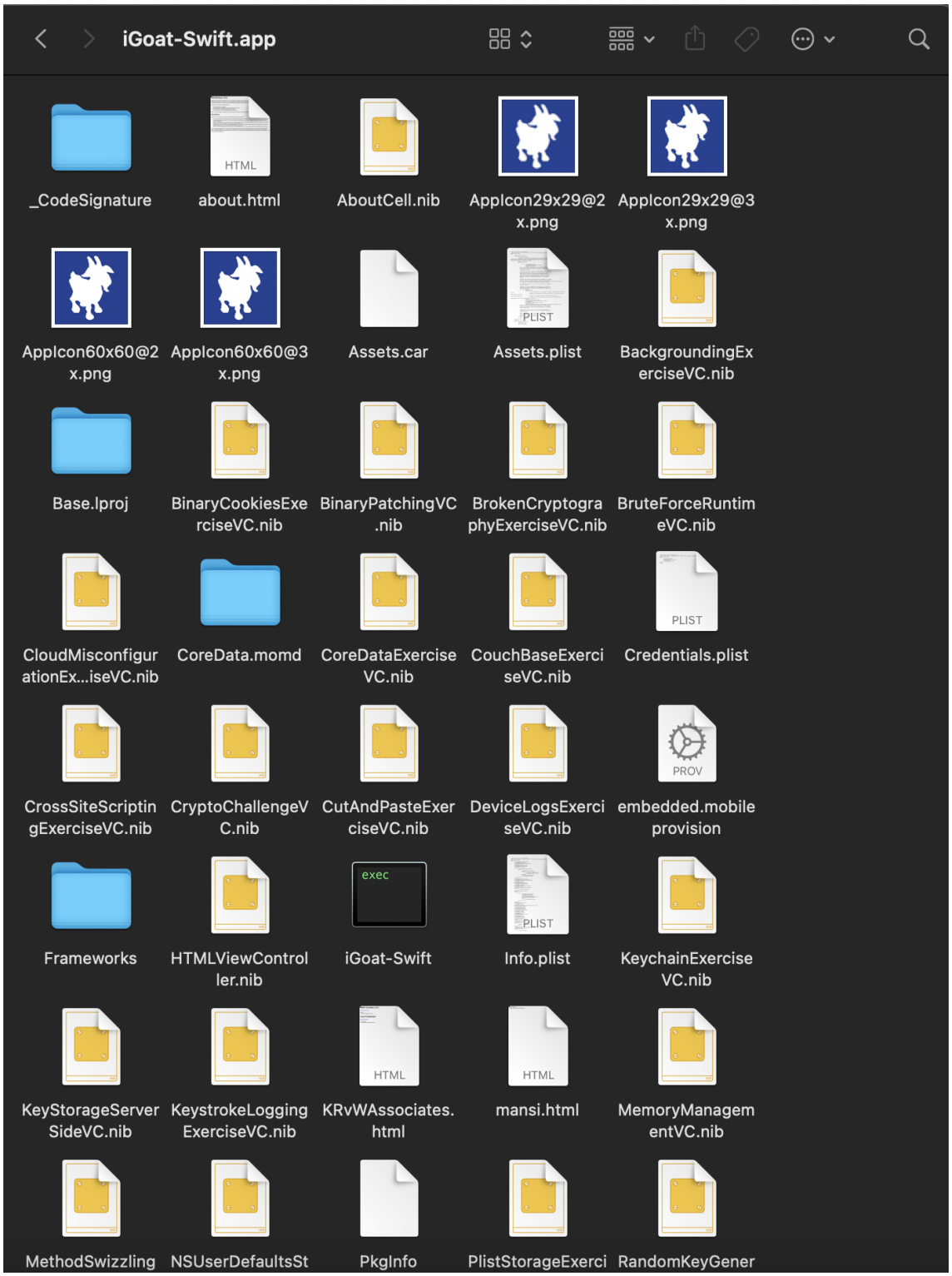


Slika 3: iOS Sandbox (Izvor: Apple, 2018)

Bundle kontejner sadrži sve što je aplikaciji potrebno za uspješno pokretanje. Sastoji se od .plist datoteke, binarne egzekucijske datoteke i datoteka s vanjskim resursima kao što su vanjske biblioteke, razvojni okviri i slično. Sve to zajedno kompresirano u jednu IPA (engl. *iOS App Store Package*) datoteku čini aplikaciju. Svaka iOS aplikacija distribuira se u obliku IPA datoteke. Radi se o ZIP kompresiranoj arhivi koja sadrži kod i resurse koji su potrebni za pokretanje aplikacije. Sama IPA datoteka u sebi sadrži direktorijsku strukturu, a najviša razina vidljiva je na slici tri. "AppThinning.plist" sadrži pojedine meta podatke o aplikaciji kao što su developer ili tim koji ju je razvio te verziju aplikacije, dok `/Payload/Application.app` sadrži ARM kompajlirani kod, odnosno samu aplikaciju i statičke resurse koji su vezani za aplikaciju. [7]



Slika 4: Struktura IPA datoteke - najviša razina (Izvor: autor, 2021)



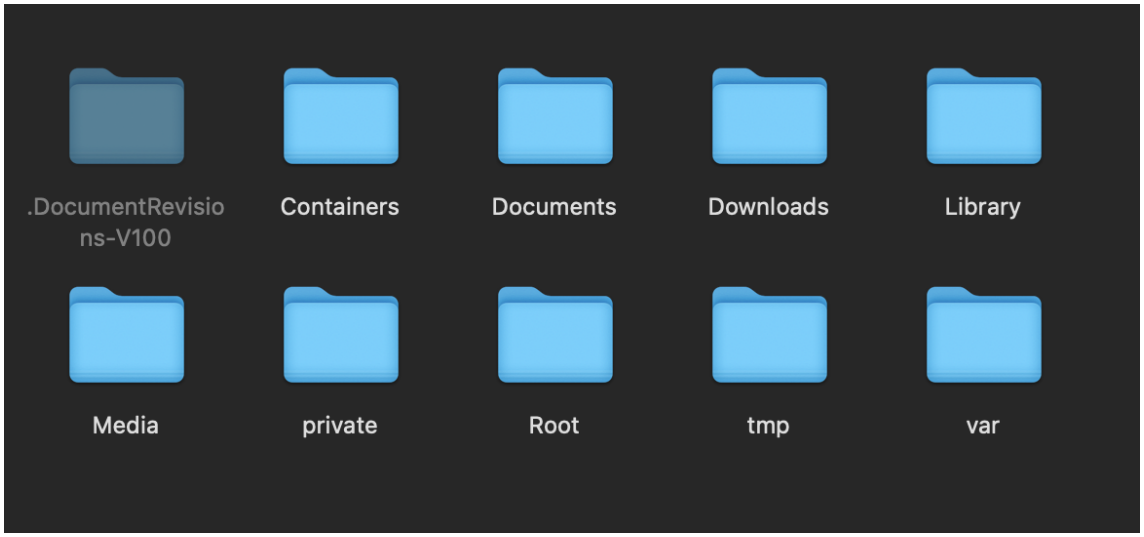
Slika 5: Struktura IPA datoteke - /Payload/Application.app (Izvor: autor, 2021)

Unutar svakog sandboxa pojedine aplikacije nalazi se *Data Container* koji sadrži datoteke koje developer želi pospremiti na uređaj prilikom instalacije aplikacije. Datoteke uglavnom služe za keširanje pojedinih podataka iz aplikacije, spremanje pojedinih podataka, ili sadrže neke osnovne podatke potrebne za sigurnosnu kopiju aplikacije, ako dođe do greške. Minimalno aplikacija sadrži *Documents*, *Library*, *Preferences* i *tmp* direktorije. [7]

Documents direktorij sadrži podatke koje je korisnik preuzeo kroz aplikaciju ili koje je kreirao koristeći aplikaciju te njima može pristupiti kroz aplikaciju. Također može se koristiti za spremanje zapisnika o greškama koje su se pojavile u aplikaciji. [7]

Library, sadrži minimalno 3 različita pod direktorija, a to su: *Application Support* koji sadrži sve aplikacijske podatke osim onih spremljenih u *Documents*, *Caches* koji sadrži analitičke podatke koje aplikacija sakuplja ili dobivene odgovore na pojedine upite kako bi se ubrzao prijenos potrebnih podataka do korisnika i *Preferences* koji sprema korisničke preference unutar aplikacije. *Caches* unutar sebe još sadrži *Snapshot* direktoriji koji sprema "screenshot" aplikacije kada se ista postavi u pozadinski način rada, a glavna datoteka *Preferences* direktorija je <BundleID>.plist koja sadrži korisničke preference (engl. *User Defaults*). Primjerice pozadinu aplikacije, raspored elemenata i slično.

tmp uglavno sadrži podatke za sigurnosnu kopiju aplikacije ili podatke koje je potrebno izbrisati nakon što se prekine ili završi pojedini proces. [7]



Slika 6: Struktura Data Container direktorija (Izvor: autor, 2021)

iCloud Container sadrži podatke ako se aplikacija povezuje na isti. Najčešće ima u sebi dva direktorija i to: *Documents* koji sadrži datoteke koje se ažuriraju od strane korisnika i šalju na iCloud kako bi isti bio stalno sinkroniziran s lokalnim stanjem u aplikaciji i *Data* koji sadrži potrebne podatke za rad s iCloudom koje korisnik nema pravo uređivati. [7]

6. Preduvjeti za sigurnosno testiranje

U prijašnjim poglavljima opisana je struktura iOS sustava i njegovih aplikacija. Sada slijedi postavljanje testnog okruženje i kreće se u penetracijsko testiranje. Za najbolju provedbu penetracijskog testiranja iOS aplikacija predlaže se upotreba Mac računala, zbog Xcode razvojnog okruženja i iOS SDK-a koji su dostupni isključivo na MacOS-u.

Svako sigurnosno testiranje započinje s postavljanjem testnog okruženja. Trenutno će testno okruženje predstavljati Mac računalo i iPhone 6s mobilni uređaj. Kroz sljedeća potpoglavlja obradit će se konfiguracija računala i uređaja za jednostavno penetracijsko testiranje iOS aplikacija. Kako bi se uspješno konfigurirao bit će potrebno *jailbreakati* testni uređaj te na računalo instalirati *Frida*, *Objection* i *Charles Proxy* alate. Osim toga potrebo je na računalu imati instalirani Xcode i Xcode developer tools te imati Wi-Fi konekciju koja omogućava promet između više spojenih klijenata. [1]

6.1. Jailbreak iOS uređaja

Jailbreak iOS uređaja često se uspoređuje sa *rootanjem* Android uređaja, međutim sam proces *jailbreakanja* dosta se razlikuje od *rootanja* Android uređaja. *Rooting* uključuje instalaciju `su` (engl. *Switch User*) binary-a. `Su` omogućuje promjenu korisnika koji pokreće i upravlja pojedinim procesima, tako je moguće postaviti se kao *root* korisnik pri pokretanju određenih procesa. Također je moguće *flashati* ROM uređaja i zamijeniti cjelokupni sustav s prilagođenim *rootanim* ROM-om. Na iOS uređajima nije moguće *flashati* prilagođeni ROM, pošto iOS uređaji dozvoljavaju pokretanje isključivo softvera koji je potpisan od strane Applea. Svrha *jailbreaka* je onemogućiti mehanizam koji provjerava potpise, odnosno omogućiti pokretanje ne potpisanog softvera. *Jailbreak* je zajednički naziv za sve alate i procese koji se koristi pri onemogućavanju potpisivanja. [1]

Bitno je napomenuti da *Jailbreak* sam po sebi nije ilegalan, međutim Apple ga ne preporučuje pošto isti deaktivira pojedine ugrađene sigurnosne mehanizme, zbog čega postoje aplikacije koje imaju ugrađene mehanizme za provjeru da li je uređaj na kojem se pokreću *jailbreakan*. Glavne prednosti "*ailbreakanja* uključuju *root* pristup datotečnom sustavu, mogućnost pokretanja ne potpisanih aplikacija, u koje ulaze većina alata za sigurnosnu analizu, lakše debugiranje, dinamička analiza te pristup Objective-C ili Swift izvršnom okruženju. [1]

Unutar pojma *Jailbreak* razlikuju se četiri tipa *jailbreaka* : *Tethered jailbreak* oblik je *jailbreaka* koji se poništava nakon ponovnog pokretanja uređaja, zbog čega je potrebno isti provađati ponovo, povezivanjem uređaja na računalo nakon svakog ponovnog pokretanja. Tako je i dobio naziv vezani (engl. *Tethered jailbreak*). Smatra se najneprihvatljivijim za krajnje korisnike pošto se uređaj na kojem je proveden ne može ponovo pokrenuti bez pristupa računalu ako ostane bez baterije. Nakon njega slijedi *Semi-tethered jailbreak* koji je sličan *Tethered jailbreaku*, s razlikom da ako uređaj ostane bez baterije moguće ga je ponovo pokrenuti, međutim isti u tom trenutku neće biti *jailbreakan*, već je *jailbreak* potrebno ponovno provesti preko računala. Sljedeći je *Semi-untethered jailbreak* koji nakon prvotne provedbe *jailbreaka* na uređaj instalira

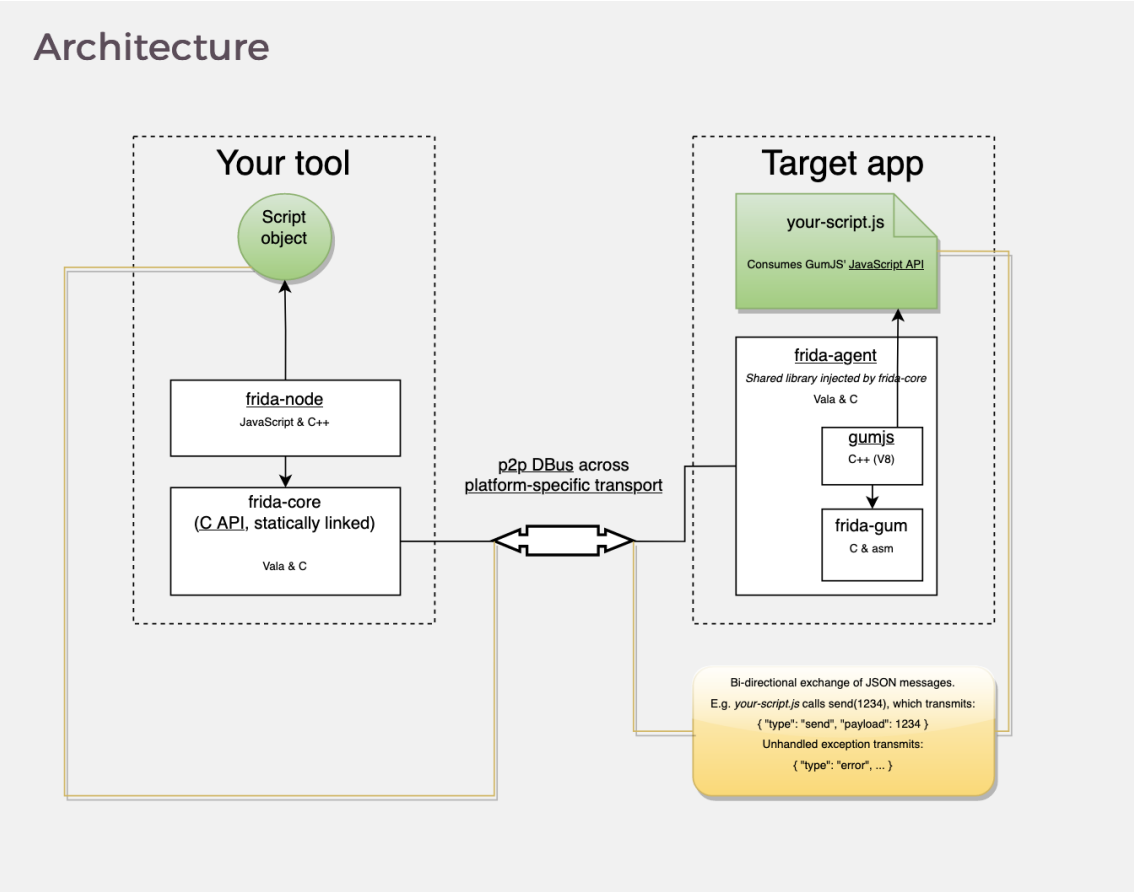
aplikaciju unutar koje postoji opcija "jailbreak". Prednost je u odnosu na *semi-tethered* to što nakon ponovnog pokretanja uređaja, koji zatim prelazi u ne *jailbreakani* način rada nije potrebno provoditi *jailbreak* putem računala, već je dovoljno samo pokrenuti aplikaciju i aktivirati opciju "jailbreak", koja će uređaj vratiti u *jailbreakani* način rada. Zadnji na listi i najrjeđi oblik *jailbreaka* je *Untethered jailbreak*, tip *jailbreaka* kojeg nije potrebno provoditi svaki puta kada se uređaj ponovno pokrene. Oslanja se na ranjivost same jezgre operacijskog sustava, a koliko ga je teško postići govori i činjenica da je zadnji bio uspješno proveden za iOS 9 pod nazivom "Pangu9". Iskoristio je `IOMobileFrameBuffer exploit` pomoću kojeg je dobiveni pristup jezgri u koju je zatim ubačen proizvoljan kod (CVE-2016-4654). [8]

Zainteresirani konkretne dostupne tipove *jailbreaka*, njihov opis i karakteristike mogu pronaći na `https://www.theiphonewiki.com/wiki/Jailbreak_Exploits`.

6.2. Frida

Frida je besplatan i *open source* dinamički set softverskih alata za kontroliranje i *debugiranje* procesa u izvođenju. Omogućava korisnicima pokretanje vlastitih skripti za vrijeme izvođenje pojedinih procesa. Odnosno dodavanje vlastitih JavaScript skripti ili biblioteka u već postojeće Windows, MacOS, GNU/Linux, iOS, Android i QNX aplikacije. Može se koristiti za analizu kriptiranih mrežnih protokola i praćenje API (engl. *Application Programming Interface*) poziva pojedinih aplikacija. Također uvelike olakšava "Black Box" testiranje pojedinih aplikacija, pošto omogućava laku analizu pojedinih aplikacijskih direktorija, aktivnih procesa na uređajima i ispis pojedinih podataka s uređaja. Kroz Fridu korisnik može "ubrizgati" svoj računalni kod direktno u memoriju procesa koji se trenutno izvodi. Kada se doda na neku aktivnu aplikaciju Frida koristi `ptrace` kako bi preuzeo dretvu aktivnog processa. Zatim ta dretva alocira dio memorije i u nju učitava početne naredbe, koje stvaraju novu dretvu koja se spaja s Fridinim *debug* serverom, koji se vrti na uređaju. Sa servera se pokreće biblioteka koja sadrži Frida agenta. Agent zatim stvara dvosmjernu komunikaciju između aplikacije i pokrenutog alata ili prilagođene skripte te vraća prvotnu dretvu aplikaciji i aplikacija nastavlja normalno dalje s izvršavanjem. [9]

Konkretna arhitektura vidljiva je na sljedećoj slici.



Slika 7: Arhitekturalna struktura Fride (Izvor: Ravnås, 2021)

Tri su načina dodavanja Fride:

1. Injektiranje, odnosno ubacivanje kroz gore navedeni proces kada je Frida pokrenuta u pozadini na nekom klijentskom uređaju. Navedeni način mogući je samo ako je uređaj *rootan* odnosno *jailbreakan*
2. Ugnježđivanje Fridine biblioteke unutar aplikacije koju se želi analizirati. Također zahtijeva *rootan* uređaj pošto se `ptrace` naredba ne može koristiti ako korisnik nema administratorske ovlasti.
3. Učitavanje prije ostalih procesa na uređaju. Moguće je postaviti da se Frida učita prije nego li se učita bilo koji drugi proces.

Za instalaciju Fride potrebno je imati instalirani Python 3 i pip package manager, te pokrenuti naredbu:

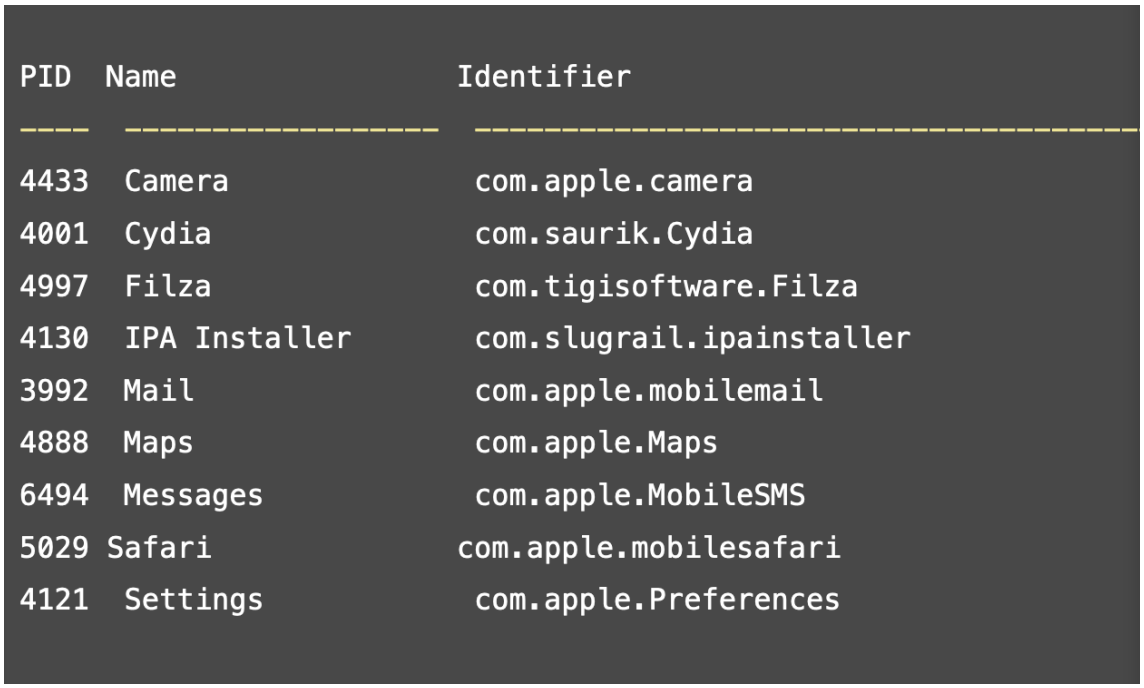
```
$ pip install frida-tools
```

Nakon uspješne instalacije Fride na računalo, potrebno ju je ubaciti u aplikaciju ili uređaj. U svrhe ovog rada koristi se *jailbreakani* iOS uređaj te je na isti potrebno instalirati Frida server. Instalacija servera provodi se odlaskom u Cydia aplikaciju, zatim pod "Sources" odabrati "Edit" u gornjem desnom kutu i u dobiveno polje unijeti "build.frida.re". Nakon dodavanja izvora vratiti

se u Cydiu. Nakon uspješne instalacije Frida servera na uređaju, potrebno je spojiti uređaj s računalom preko USB-a i na računalu provesti naredbu

```
frida-ps -U
```

Uspješnom instalacijom i pokretanjem Fride na uređaju, gornja naredba izlistati će procese koji se nalaze na istom. Slika 8 prikazuje izlistane procese.



Slika 8: Frida izlistani aktivni procesi na uređaju (Izvor: autor, 2021)

Napomena: Nakon instalacije Frida servera možda će biti potrebno potražiti i pokrenuti isti na uređaju. To je najlakše izvesti kroz `ssh` spajanje na uređaj i zatim iz *root* direktorija pokrenuti frida-server.

6.3. Objection

Objection je set softverskih alata izgrađen na Fridi koji omogućava sigurnosno testiranje dekriptirane iOS aplikacije tijekom njenog izvršavanja. Neki od značajki uključuju inspekciju aplikacijskog *keychaina* kao i ostalih artefakata koji su prisutni na disku tokom ili nakon izvršavanja aplikacije. Objection omogućava ubacivanje objekata u aplikaciju tijekom izvršavanja i pokreće ih unutar sigurnosnog konteksta aplikacije. Neke od operacija koje omogućava su naredbe tipa `ls`, koja će omogućiti pretraživanje datotečnog sustava aplikacije ili `ios sslpinning disable` koja će preskočiti pozivanje metode koja provodi "Certificate pining". Također moguće je preuzimanje datoteka iz datotečnog sustava ukoliko korisnik nad datotekom ima ovlaštenje za čitanjem te dodavanje izmjenjene datoteke ukoliko korisnik nad istom ima ovlaštenje za pisanje. Uz to u sebi sadrži i konzolno okruženje za rad sa SQLite bazom podataka. Omogućuje ispis trenutne procesorske memorije, rad sa učitanim modulima i njihovo eksportiranje. Ispis podataka spremljenih u korisničkim preferencama, kolačićima i .plist datotekama te

keychain sadržaja. Zaobilazak bimetrijskih provjera i ukoliko je aplikacija pisana u Objective-C programskom jeziku ispis klasa i klasnih metoda. [10]

Objection je moguće preuzeti koristeći pip3.

```
pip3 install objection
```

Nakon uspješne instalacije moguće je koristiti naredbu "Objection" kojom se pristupa njegovom setu alata. Kroz naredbu "Objection –help" ispisuju se moguće konfiguracijske opcije alata. Više informacija o pojedinim opcijama moguće je pronaći na službenom GitHubu Objectiona, a dio koji će se koristiti u radu bit će pobliže objašnjen tijekom korištenja.

```
$ objection --help
Usage: objection [OPTIONS] COMMAND [ARGS]...


       _     _         _   _
   ___| |_  |_|___ ___| |_|_|___ ___
  | . | . | | | -_|  _|  _| | . |   |
  |___|___|_| |___|___|_| |_|___|_|_|
          |___|(object)inject(ion)

       Runtime Mobile Exploration
          by: @leonjza from @sensepost

  By default, communications will happen over USB, unless the --network
  option is provided.

Options:
  -N, --network       Connect using a network connection instead of USB.
                      [default: False]
  -h, --host TEXT     [default: 127.0.0.1]
  -p, --port INTEGER  [default: 27042]
  -g, --gadget TEXT   Name of the Frida Gadget/Process to connect to.
                      [default: Gadget]
  --help              Show this message and exit.

Commands:
  device_type  Get information about an attached device.
  explore      Start the objection exploration REPL.
  patchapk     Patch an APK with the frida-gadget.so.
  patchipa     Patch an IPA with the FridaGadget dylib.
  version      Prints the current version and exists.
```

Slika 9: Konfiguracijske opcije Objection softverskog alata (Izvor: SensePost, 2017)

7. OWASP iGoat i Damn Vulnerable iOS App (DVIA)

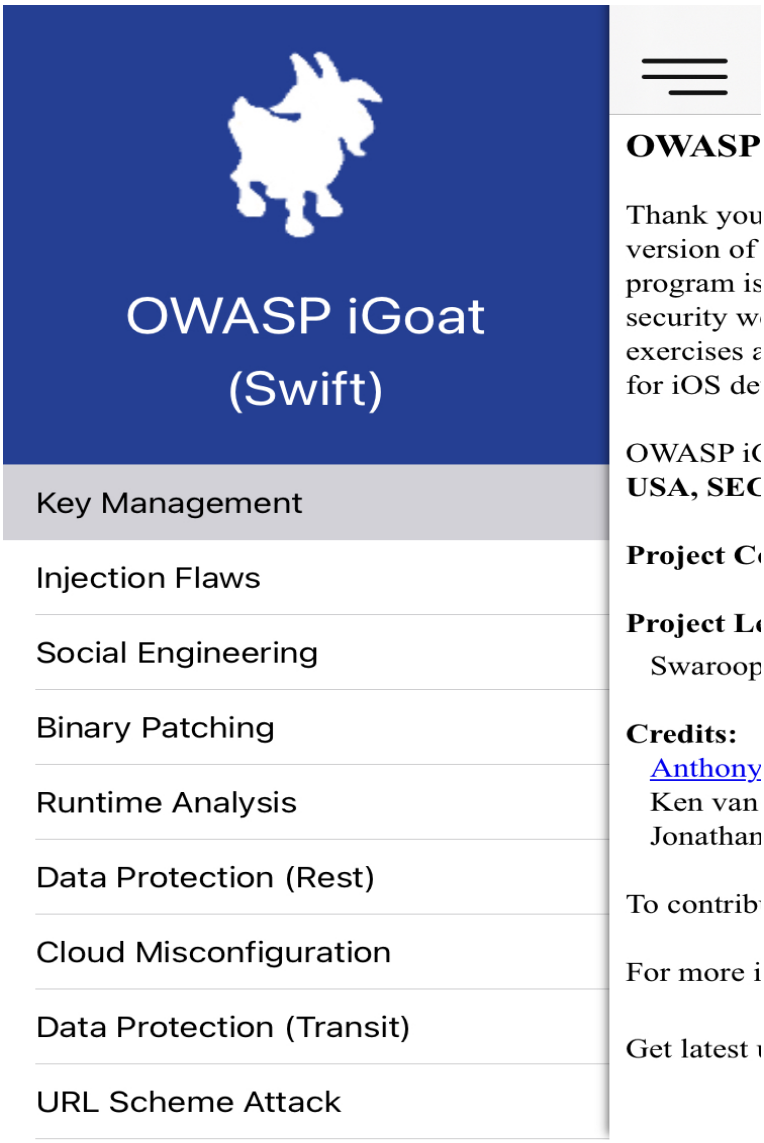
7.1. Instalacija aplikacija OWASP iGoat i DVIA

Sigurnosnu aplikaciju OWASP iGoat može se preuzeti sa `https://github.com/OWASP/iGoat-Swift.git` naredbom `git clone .` Nakon uspješnog preuzimanja potrebno je prepisati njen certifikat sa svojim developer certifikatom. To se može provesti na više načina, u ovom slučaju proces prepisivanja proveden je kroz aplikaciju iOS App Signer, koju je moguće preuzeti na `https://dantheman827.github.io/ios-app-signer/`. Nakon što se preuzeta IPA datoteka prepiše moguće ju je jednostavno *sideloadati* na uređaj kroz Xcode razvojno okruženje. Odlaskom na dodavanje novog simulatora/uređaja unutar razvojnog okruženja i klikom na znak plusa kod instaliranih aplikacija odabrati novo potpisanu IPA datoteku.

Damn Vulnerable iOS App (DVIA) dodatna je testna aplikacija koja uz iGoat pruža niz sličnih scenarija za penetracijskog testiranje iOS aplikacija. Njen autor je Avatar Prateek Gianchandani a moguće ju je preuzeti sa `https://github.com/prateek147/DVIA-v2`. Instalirava se jednako kao i iGoat.

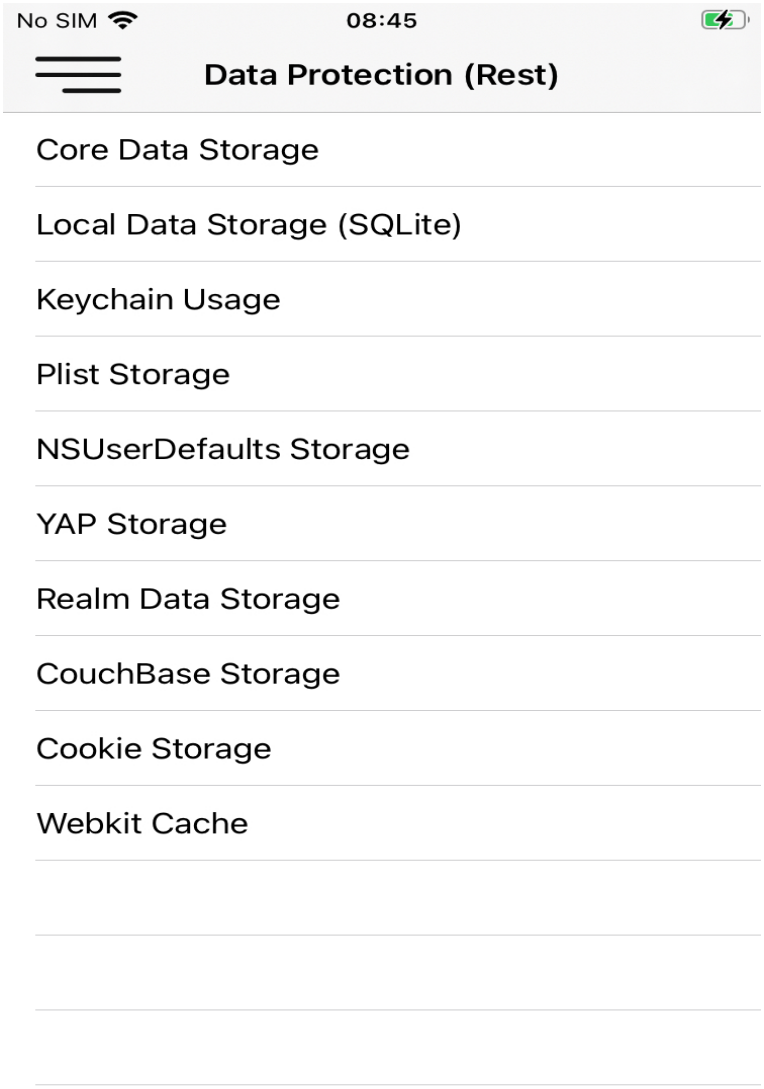
7.2. Penetracijsko testiranje aplikacije OWASP iGoat

Penetracijsko testiranje iGoat aplikacija fokusirat će se na fazu eksploatacije. Odlaskom na službene iGoat stranice i GitHub stranicu javno dostupnog projekta vidljivo je da se radi o nesigurnoj aplikaciji izrađenoj za usavršavanje sigurnosnog testiranja. Kroz iGoat developeri se mogu upoznati s najčešćim sigurnosnim propustima koji ih mogu zadesiti u svakodnevici i kako ih prepoznati, a zatim je na njima da ih isprave. Odnosno da pripaze da ih ne ponavljaju u svojim projektima. Aplikacija se sastoji od jednostavnog početnog ekrana i bočnog izbornika u kojem su posloženi izazovi ovisno o tipu sigurnosne ranjivosti.

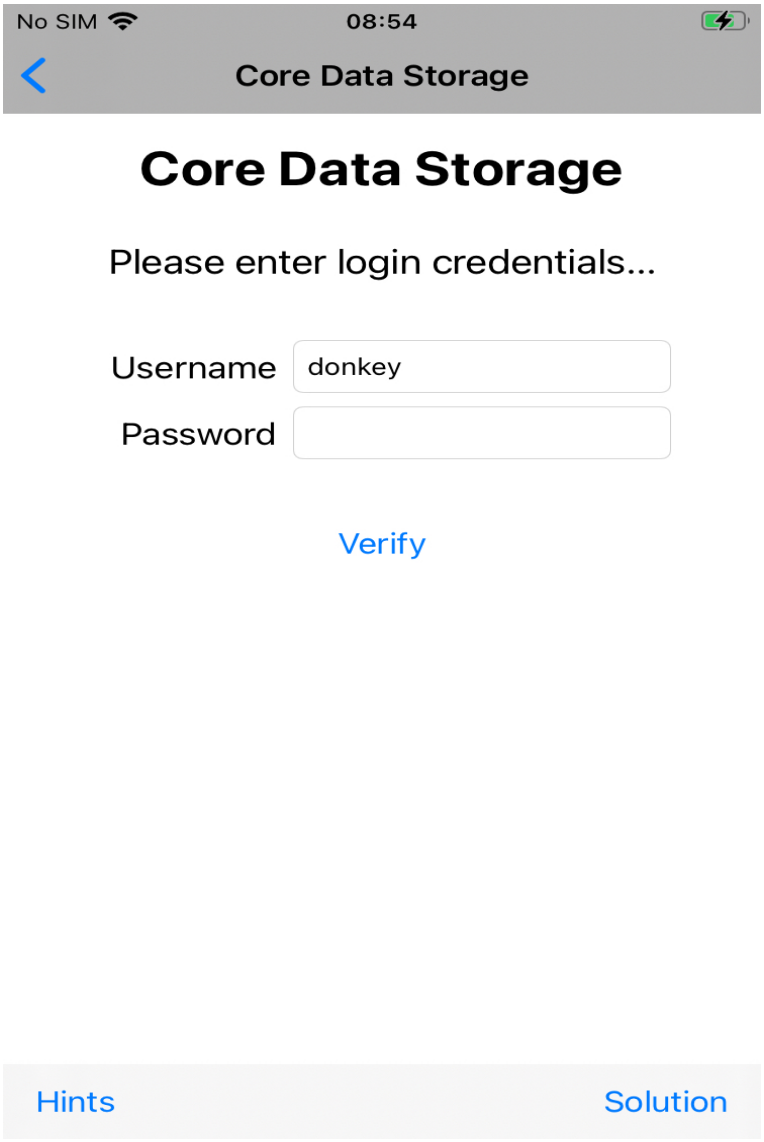


Slika 10: iGoat - izbornik ranjivosti (Izvor: OWASP, 2017)

Svaka ranjivost sadrži konkretne tipove napada na pojedini dio aplikacije ili sustava. Promatra li se nesigurnost pohranjenih podataka unutar aplikacije može se promatrati veći broj mogućih napada na nesigurne podatke. Napadač može napasti lokalno spremljene podatke: napadom na Appleov ORM, napadom na samu bazu podataka na uređaju, napadom na *Keychain* koji sprema lozinke i certifikate, napadom na korisničke preference, napadom na *web cash* ili kolačiće na klijentu i slično. Nakon što se odabere konkretna ranjivost dobiva se zadatak, uglavnom je to ekran kroz koji korisnik može unijeti pojedine podatke i ovisno o unesenim podacima dobiva poruku o uspjehu ili neuspjehu.

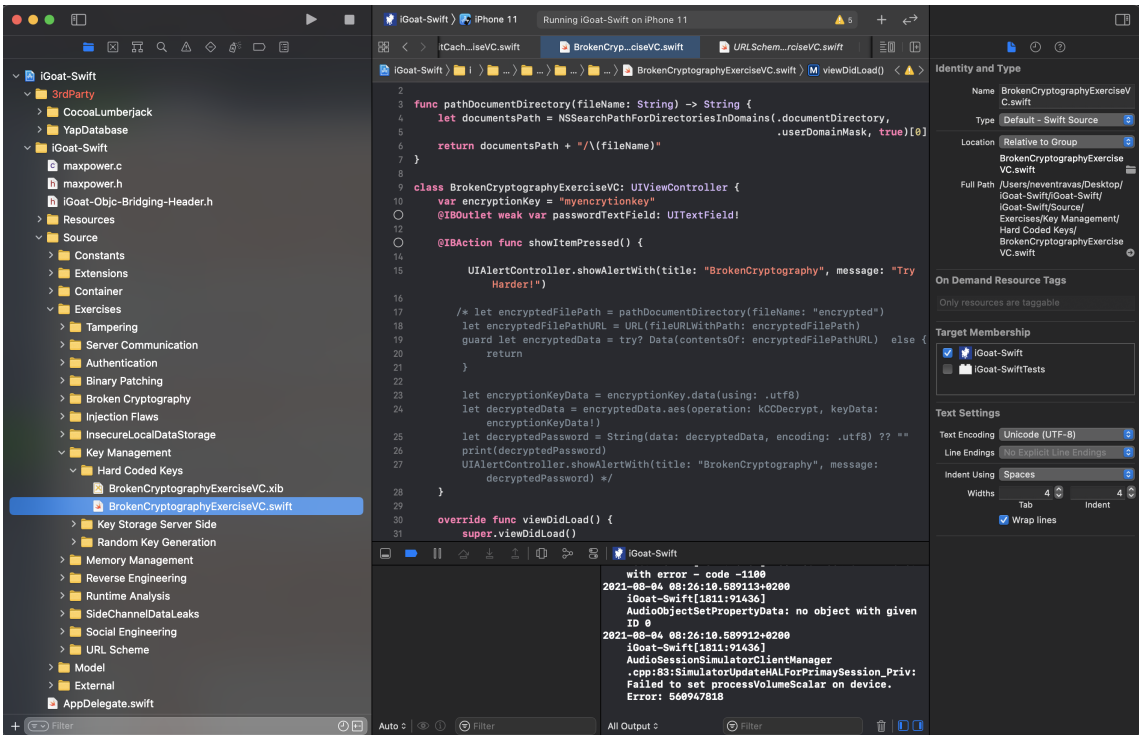


Slika 11: iGoat - mogući vektori napdadi (Izvor: OWASP, 2017)



Slika 12: iGoat - ekran zadatka (Izvor: OWASP, 2017)

Osim prolaska kroz aplikaciju, kako bi se primjetili sigurnosni propusti s korisničke razine, sa GitHub stranice (`https://github.com/OWASP/iGoat-Swift`) preuzima se i izvorni kod koji sam u sebi sadrži komentare i strukturno je podijeljen ovisno o promatranoj ranjivosti. Neke se ranjivosti tako mogu činiti trivijalnima pošto se iz izvornog koda mogu iščitati konkretni propusti, ali to ne znači da ih treba zanemariti. U ovom radu takvi propusti bit će opisani i ispravljeni u kodu, u sklopu *white box* testiranja, ali isto tako pokušat će se do njih doći i "iz vana" emitirajući napadača koji pokušava "probiti" aplikaciju. Primjerice doznati administratorsku lozinku, a ista je hard kodirana u izvornom kodu.



Slika 13: iGoat - izvorni kod (Izvor: OWASP, 2017)

Nakon što su prikupljeni podaci o aplikaciji kreće se u korak analize. U konkretnoj analizi bit će opisani problemi koji su pronađeni u aplikaciji i zašto predstavljaju sigurnosnu opasnost za aplikaciju. Prije nego se krene bitno je utvrditi razliku između sigurnosnog rizika i sigurnosne ranjivosti. Sigurnosni rizik odnosi se na zamijećene propuste u aplikaciji koji bi mogli biti iskorišteni kako bi se razotkrila ranjivost u aplikaciji. Tako sigurnosni rizik može i ne mora predstavljati opasnost, pošto njegova razina ovisi o pojedinim uvjetima i ljudima koji rade na aplikaciji. Odnosno svako poduzeće koje izdaje aplikaciju na tržište to čini s postotkom rizika, koji se regulira ovisno o šteti koja može nastati određenim sigurnosnim propustom. Preporuka je sigurnosni rizik smanjiti što je više moguće, ali njegovo postojanje ne dovodi aplikaciju nužno u opasnost. Sigurnosni rizik tek postaje opasnost kada se kroz isti potakne ili izluči pojedina ranjivost koja se može iskoristiti za napad na aplikaciju ili poduzeće. Nakon što se opiše pojedina ranjivost kreće se u eksploataciju, odnosno pokušaj dohvaćanja povjerljivih podataka kao ne autorizirana osoba. Pojedine eksploatacije bit će popraćene procesom i opisom, dok će druge biti samo opisane i ispravljene. Opisne eksploatacije odnosit će se na ranjivosti koje se mogu pokazati opasnima ako neovlaštena osoba dobije pristup izvornom kodu aplikacije.

7.2.1. Dohvaćanje ne kriptiranih podataka iz lokalne SQLite baze podataka

Prvi primjer zaštite lokalnih podataka odnosit će se na analizu prisustva "hard kodiranih" autentifikacijski podataka unutar aplikacije odnosno unutar lokalne baze podataka. Provedbom "White Box" testiranje moguće je zamijetiti već sljedeću grešku u kodu.

```
user = NSEntityDescription.insertNewObject(forEntityName: "User", into: context) as?
     User
if let user = user {
    user.email = "john@test.com"
    user.password = "coredbpassword"
    coreDataHelper.saveContext()
    }
```

Prikazani kod sprema autentifikacijske podatke u čitljivom obliku (engl. *Plain Text*) bez ikakvog heširanja ili kriptiranja ulaznih podataka. Tako je već iz koda jednostavno iščitati tražene vrijednosti. Pošto akreditacijski podaci za pojedinog korisnika nisu sigurno spremljeni moguće ih je izvući i pročitati direktno iz baze podataka koja se nalazi na uređaju. Datoteka koja sadrži zapis SQLite baze na uređaju, nakon popunjavanja postavlja se na sljedeću lokaciju:

```
/private/var/mobile/Containers/Data/Application/C3AA8582-B002-4F25-B373-B0095259AF23
    /Library/Application Support
```

Kao što je već navedeno u radu, svaka aplikacija koja se instalira na iOS uređaj sprema se u Application direktorij pod *mobile* korisnikom. Spajanjem uređaja na istu mrežu kao i računala moguće je spojiti računalo i uređaj koristeći SSH (engl. *Secure Shell*) protokol.

Napomena: Iako iPhone sam po sebi podržava SSH protokol, odnosno može sadržavati SSH server na koji se može spojiti, Apple ga ne isporučuje sa svojim uređajima. Kako bi se moglo SSH-at u pojedini uređaj potrebno ga je *jailbreakati* i instalirati open SSH server putem *Cydiae* ili neke druge aplikacije za preuzimanje neslužbenih aplikacija.

Unutar terminala na računalu pokreće se naredba

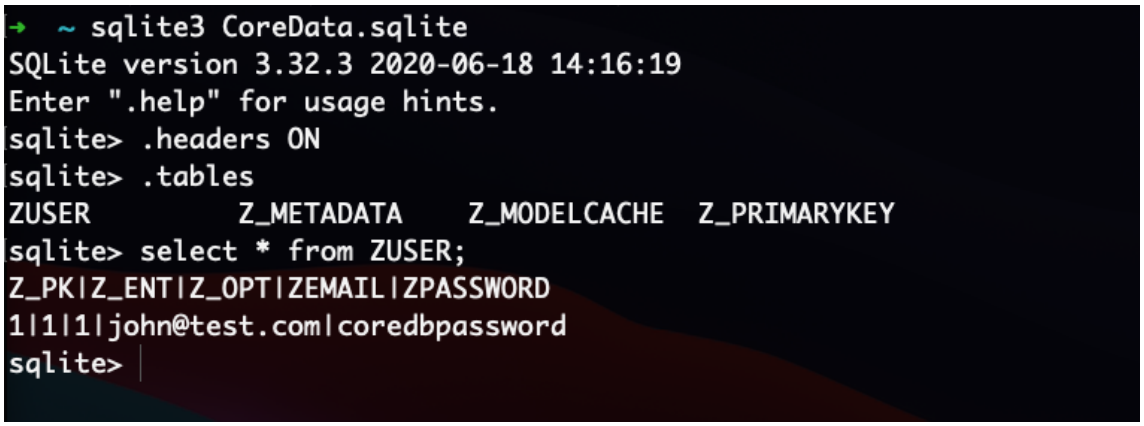
```
ssh root@192.168.1.4 (IP adressa lokalne mreze).
```

Provedbom naredbe traži se lozinka koja je u naprijed zadana kao "alpine". Nakon uspješnog spajanja dobiva se pristup datotečnom sustavu uređaja. Kroz analizu strukture iOS aplikacije poznato je gdje se sprema datoteka baze podataka. Tester se postavlja u *Application* direktorij i filtrira dostupne aplikacije na uređaju. Filtriranje je moguće na više načina, ovdje je provedeno kroz sljedeću naredbu:

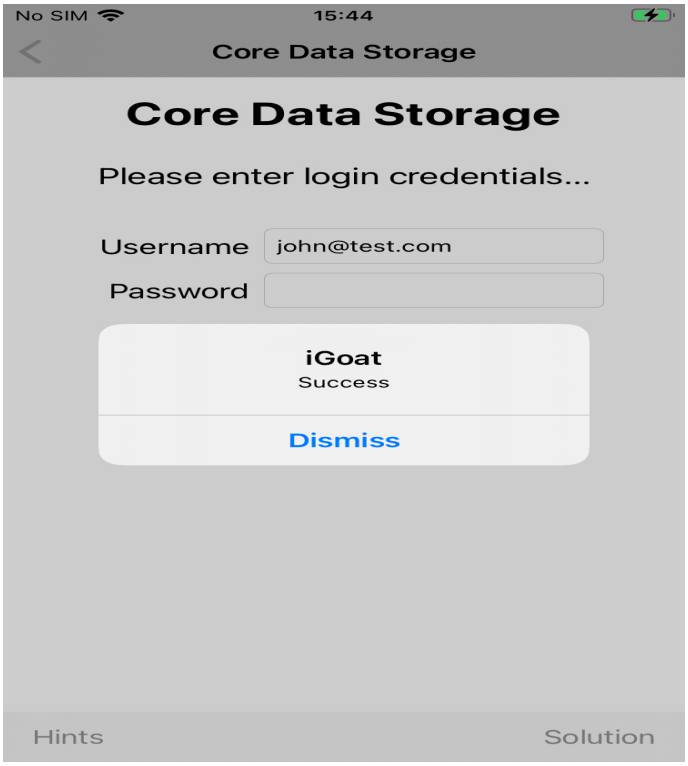
```
find * | grep "iGoat-Swift"
```

Nakon zaprimanje povratne informacije potrebno se prebaciti u traženi direktoriji unutar čije strukture se ide sve do *Application Support* direktorija koji sadrži željenu datoteku. Pronalaskom željene datoteke, u ovom slučaju `CoreData.sqlite` istu se može kopirati u korisnički direktorij. Nakon kopiranja datoteke koristeći `sftp` (engl. *Secure File Transfer Protocol*) naredbom `get CoreData.sqlite` moguće je preuzeti datoteku na računalo. Preuzetu datoteku

dalje se otvara na računalu u nekom SQLite uređivaču ili u terminalu naredbom `sqlite3 CoreData.sqlite`. Zatim se pretražuje koje su dostupne tablice u bazi i izlistava se njihov sadržaj. Pošto podaci nisu zaštićeni, selektiranjem korisnika iz baze dobiva se rezultat sa slike 14, a upisom dobivenih podataka u formu na uređaju dobiva se rezultat sa slike 15.



Slika 14: iGoat - nesigurno lokalno spremanje povjerljivih podataka (Izvor: autor, 2021)



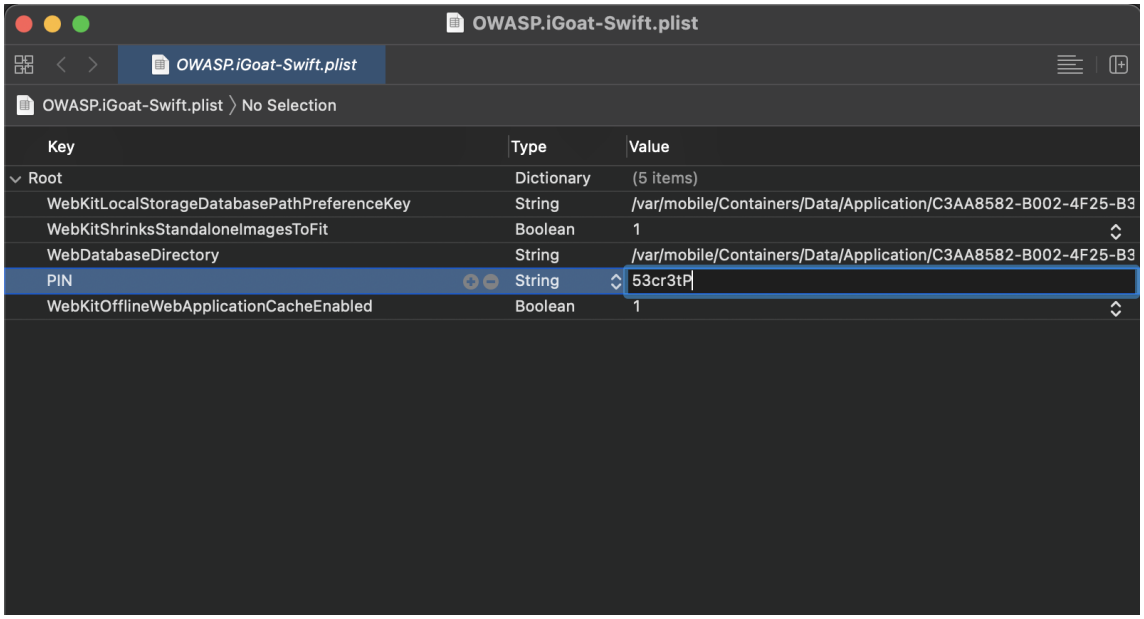
Slika 15: iGoat - eksploatacija nesigurnog lokalnog spremišta podataka (Izvor: OWASP, 2017)

7.2.2. Nesigurno spremanje lozinki u korisničkim preferencama

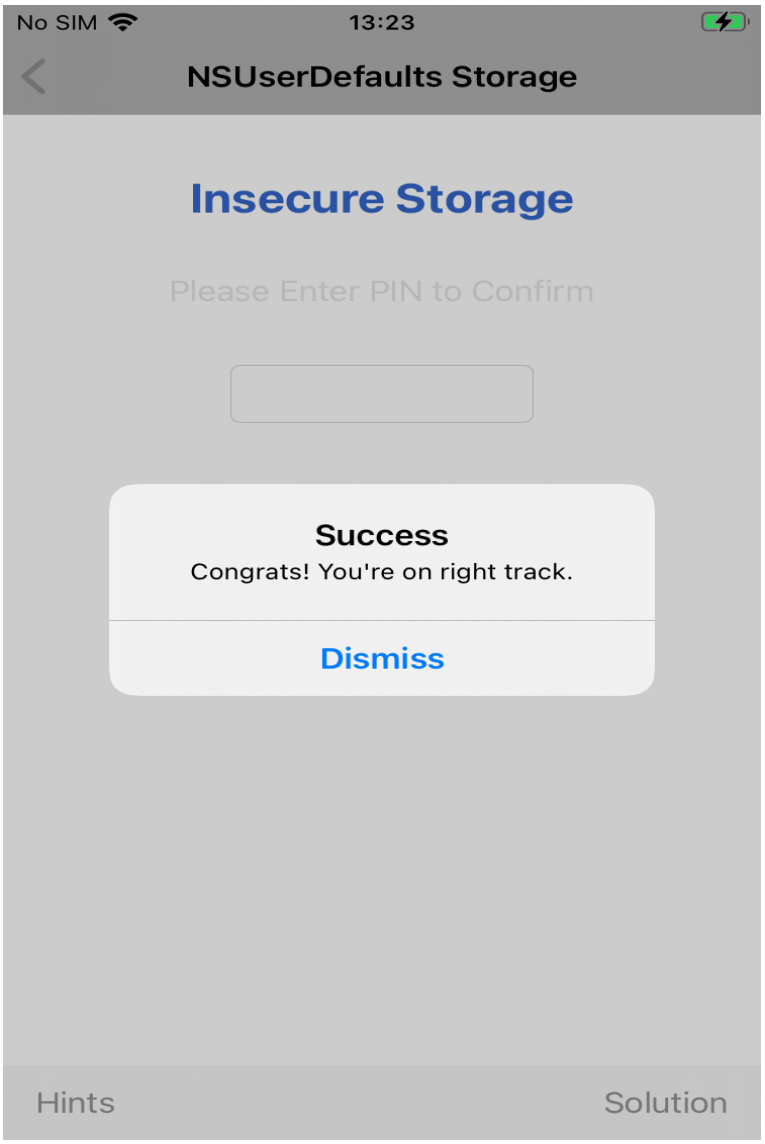
Još jedan primjer neispravno spremljenih povjerljivih podatka je spremanje lozinki unutar korisničkih preferenci (engl. *User Defaults*). Iste se spremaju u obliku rječnika u .plist datoteku unutar aplikacijskog kontejnera te ako korisnik ima pristup datotečnom sustavu može ih jednostavno pročitati. U konkretnom slučaju korisničke preference smještene su u *OWASP.iGoat-Swift.plist* datoteci koja se nalazi na sljedećoj putanji:

```
/private/var/mobile/Containers/Data/Application/C3AA8582-B002-4F25-B373-B0095259AF23
    /Library/Preferences
```

Provjerom datoteke jasno se pronalazi PIN odnosno korisnička lozinka. Prikazano na slici 16, a unosom pina u ekran aplikacije dobiva se poruka sa slike 17.



Slika 16: OWASP.iGoat-Swift.plist - Datoteka korisničkih preferenci (Izvor: autor, 2021)



Slika 17: iGoat - uspješno iskorištavanje ne kriptiranih korisničkih preferenci (Izvor: OWASP, 2017)

7.2.3. Nesigurno spremanje lozinki u Keychainu

Osim korisničkih preferenci dodatna i najsigurnija varijanta za spremanje osjetljivih podataka je tako zvani "Apple Keychain". Podaci spremljeni u *keychainu* kriptirani su s dva različita AES-256-GCM ključa. Jedan ključ je metapodatkovni ključ i koristi se za kriptiranje svih podataka osim `kSecValue`[1] vrijednosti. Drugi je tajni ključ i njime se kriptira `kSecValueData`[2] vrijednost. Metapodatkovni ključ zaštićen je sigurnosnom enklavom i kešira se u aplikacijskom procesoru, čime se omogućuje brzo pretraživanje zapisa unutar *keychaina*. Dok se tajni ključ nalazi unutar sigurnosne enklave i nije tako jednostavno dostupan. Sam *keychain* implementiran je kao SQLite baza podataka. Zahvaljujući pristupnim grupama koji su definirani unutar *keychain* API-a preko kojeg se komunicira s definiranom bazom, moguće je da više procesa (aplikacija) istovremeno koriste *keychain* ovisno o pristupnoj grupi u koju spadaju. Međutim

[1] kSecValue je kriptirani kSecValueData
[2] kSecValueData je ključ čiju vrijednost čine podaci onoga što ga sadrži.[13]

sadržaj unutar *keychaina* mogu dijeliti samo aplikacije koje je razvio isti developer. [14]

Ovisno u kojem trenutku pojedina aplikacija zatraži autentifikacijske podatke ili pojedine certifikate, razlikuju se pet tipova dostupnosti (engl.*Availability*) pojedinih podataka unutar *keychaina*:

- dotupno dok je uređaj otključan (engl. *When unlocked*)
- dotupno dok je uređaj zaključan (engl. *While locked*)
- dotupno nakon prvog otključavanja (engl. *After first unlock*)
- stalno dostupno (engl. *Always*)
- dostupno nakon postavljanja lozinke (engl. *Passcode enabled*)

Ovisno o tipu dostupnosti razlikuje se zaštita koja se primjenjuje na podatak (engl. *File data protection*) tako se razlikuju sljedeći tipovi zaštite:

- `NSFileProtectionComplete`
- `NSFileProtectionCompleteUnless Open`
- `NSFileProtectionCompleteUntil FirstUserAuthentication`
- `NSFileProtectionNone`
- bez zaštite

Pristup pojedinom podatku iz *keychaina* posebno se definira ovisno o tipu dostupnosti i tipu zaštite (engl. *Keychain data protection*):

- `kSecAttrAccessibleWhenUnlocked`
- stalno dostupno
- `kSecAttrAccessibleAfterFirstUnlock`
- `kSecAttrAccessibleAlways`
- k`SecAttrAccessibleWhenPasscodeSetThisDeviceOnly`

Availability	File data protection	Keychain data protection
When unlocked	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
While locked	NSFileProtectionCompleteUnlessOpen	N/A
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Always	NSFileProtectionNone	kSecAttrAccessibleAlways
Passcode enabled	N/A	kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly

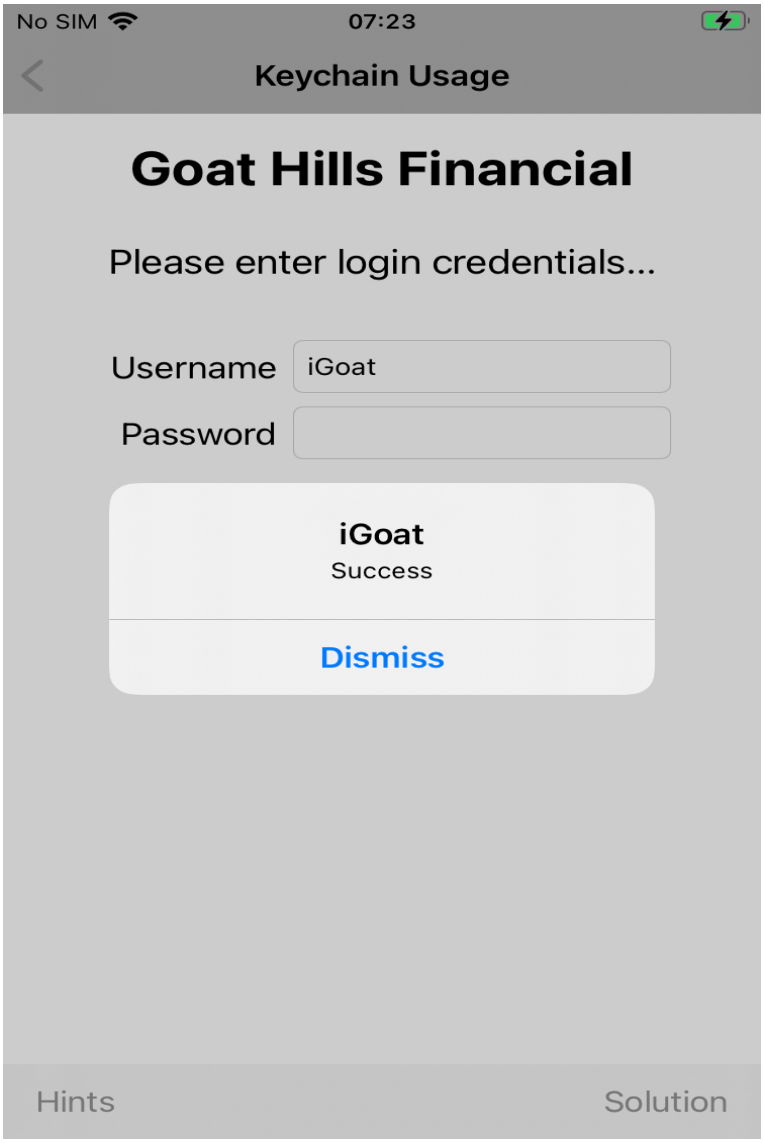
Slika 18: Dostupnost i zaštita podataka iz keychaina ovisno o tipu dostupnosti (Izvor: Apple, 2021)

Iako *keychain* sam po sebi štiti spremljene podatke, kod spremanja podataka iste se uvijek preporučuje heširati, pošto ovisno o tipu pristupa, podatak može biti ekstrahiran iz aplikacije. Izvlačenje podatka moguće je provesti kroz "keychain dump", odnosno ispis svih podataka iz *keychaina*. Tako slijedi primjer u kojem se korisnička lozinka sprema u obliku čitljivog teksta te se ista izvlači kroz "keychain dump" koristeći *Objection*.

```
OWASP.iGoat-Swift-v2 on (iPhone: 14.4) [usb] # ios keychain dump
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created                  Accessible    ACL    Type      Account    Service    Data
-----------------------  ------------  -----  --------  ---------  ---------  ------
2021-08-14 05:18:33 +0000  WhenUnlocked  None   Password  iGoat      SaveUser   taoGi
OWASP.iGoat-Swift-v2 on (iPhone: 14.4) [usb] #
```

Slika 19: Ne zaštićeni podatak unutar keychaina (Izvor: autor, 2021)

Slika prikazuje uspješan ispis lozinke iz *keychaina* koristeći *Objection* i ugrađenu naredbu "ios keychain dump". Naredba interno prolazi po *keychainu* i mapira pojedine zapise u rječnik čije vrijednosti dekodira, pretvara u *String* i ispisuje na ekran. Slika 20 zatim prikazuje uspješnu prijavu sa pronađenom lozinkom "taoGi".



Slika 20: Uspješna autentifikacija sa nezaštićenim podacima iz keychaina (Izvor: OWASP, 2017)

Međutim ako se developer potrudi heširati spremljene lozinke, moguće je uvelike poboljšati njihovu sigurnost, tako je u sljedećem primjeru implementirana klasa koja će omogućiti SHA256 heširanje pojedinog *Stringa*. Na *string* bit će dodani i nasumično generirani *salt* za još veći stupanj sigurnosti. U primjeru *salt* će biti spremljen na klijentu, dok se u praksi preporučuje sigurnosno čuvanje *salta* na serveru kako bi klijent i server mogli po potrebi dohvaćati heširanu vrijednost.

```
import Foundation
import CommonCrypto

extension Data {
    public func sha256() -> String{
        return hexStringFromData(input: digest(input: self as NSData))
    }

    private func digest(input : NSData) -> NSData {
        let digestLength = Int(CC_SHA256_DIGEST_LENGTH)
        var hash = [UInt8](repeating: 0, count: digestLength)
        CC_SHA256(input.bytes, UInt32(input.length), &hash)
        return NSData(bytes: hash, length: digestLength)
    }

    private  func hexStringFromData(input: NSData) -> String {
        var bytes = [UInt8](repeating: 0, count: input.length)
        input.getBytes(&bytes, length: input.length)

        var hexString = ""
        for byte in bytes {
            hexString += String(format:"%02x", UInt8(byte))
        }

        return hexString
    }
}

public extension String {
    func sha256() -> String{
        if let stringData = self.data(using: String.Encoding.utf8) {
            return stringData.sha256()
        }
        return ""
    }
}
```

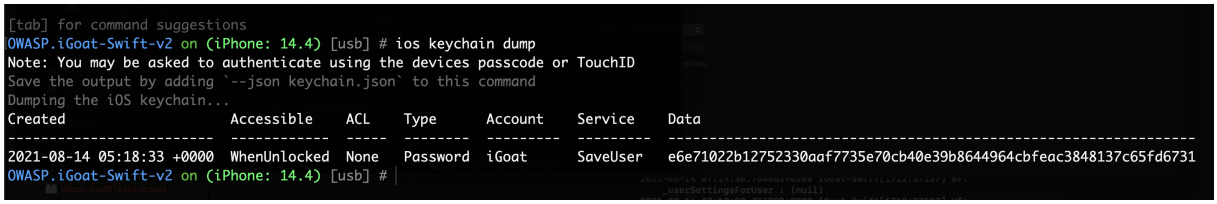
Nakon omogućavanja heširanja *stringa*, slijedi provođenje heširanja.

```
    func passwordHash(for user: String, password: String) -> String {
        let salt = "x4vV8bGgqqmQwgCoyXFQj+(o.nUNQhVP7ND"
        return "\(password).\(user).\(salt)".sha256()
    }
```

```
    func secureStore(userName: String, password: String) {
    do {
        let finalHash = passwordHash(for: userName, password: password)
        // This is a new account, create a new keychain item with the account
            name.
        let passwordItem = KeychainPasswordItem(service: "SaveUser",
                                                account: userName,
                                                accessGroup: nil)

        // Save the password for the new item.
        try passwordItem.savePassword(finalHash)
    } catch {
        fatalError("Error updating keychain - \(error)")
    }
}
```

Nakon pokretanja izmjena i ispisa sadržaja iz trenutnog *keychaina* vidljive su izmjene prikazane na slici 21.



Slika 21: Heširani popdaci unutar keychaina (Izvor: autor, 2021)

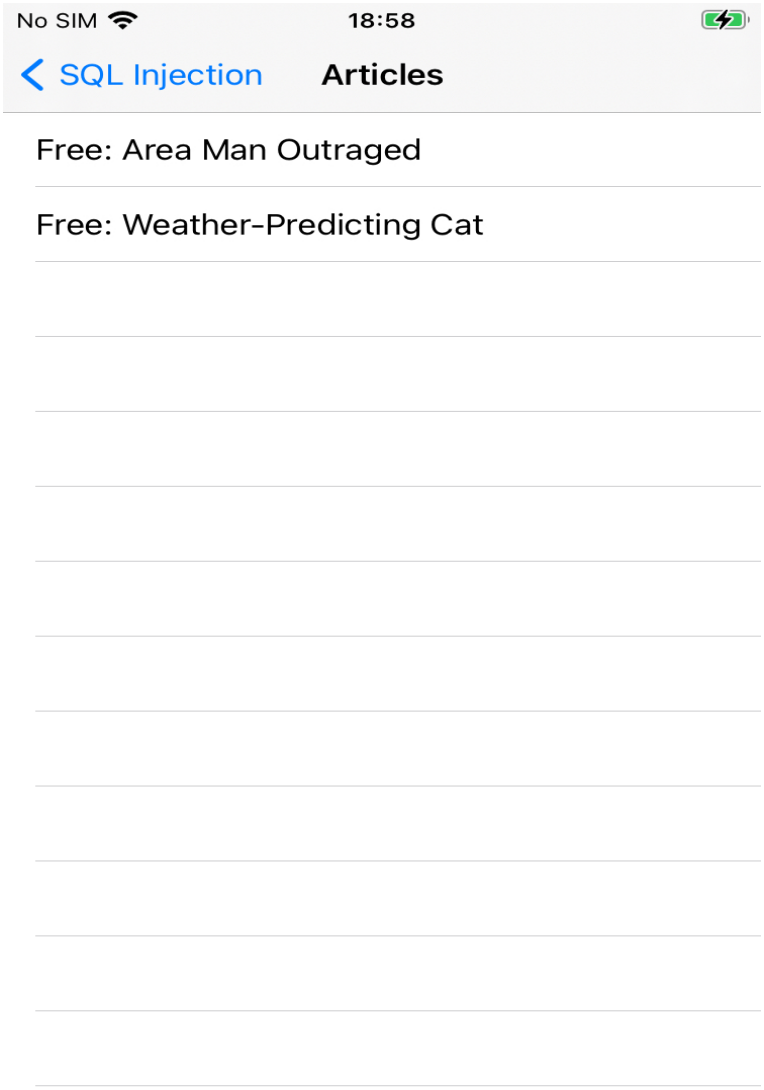
7.2.4. SQL injection u iOS aplikacijama

SQL injection postupak je uvjeravanja računala da za vrijeme provođenja programa korisničke unose u stvarnom vremenu obrađuje kao SQL naredbe. Time napadač dobiva uvid u sadržaj baze podataka pojedine aplikacije i može po želji mijenjati njen sadržaj. Tako stječe potpunu kontrolu nad podacima s kojima radi aplikacija. Opasnost od *SQL injectiona* pojavljuje se kada developer ne pročišćuje podatke koje korisnik unaša u tražilicu, a iste koristi za postavljanje upita na bazu. Jedna ne primjerena i po aplikaciju i njene podatke štetna implementacija upita i njegovog popunjavanja može izgledati kao:

```
let searchStr = "\(searchField.text!)"
let query = "SELECT title FROM article WHERE title LIKE '\(searchStr)' AND premium=0
    "
```

Kod navedenog primjera aplikacija zaprima direktan unos od korisnika bez da isti prvotno pročisti ili ograniči korisnikov unos s pojedinim restrikcijama, primjerice zabrane korištenja specijalnih znakova (tipa navodnici ili znak jednako). S takvim propustom aplikacija postaje ranjiva na *SQL injection*. Jedino što napadač tada mora znati je o kojoj se bazi podataka radi (kod iOS uređaja to je gotovo uvijek SQLite). Znanjem baze podataka i njenih naredbi, napadač može zavarati aplikaciju da zahvaljujući interpolaciji *stringova* upit u tražilicu interpretira kao SQL naredbu, a ne kao običan tekst, kako je to developer zamislio. Koraci provedbe *SQL injec-*

tiona mogu biti sljedeći: Napadač provjeri koje rezultate je moguće dobiti prilikom normalnog korištenja tražilice (Slika 22).



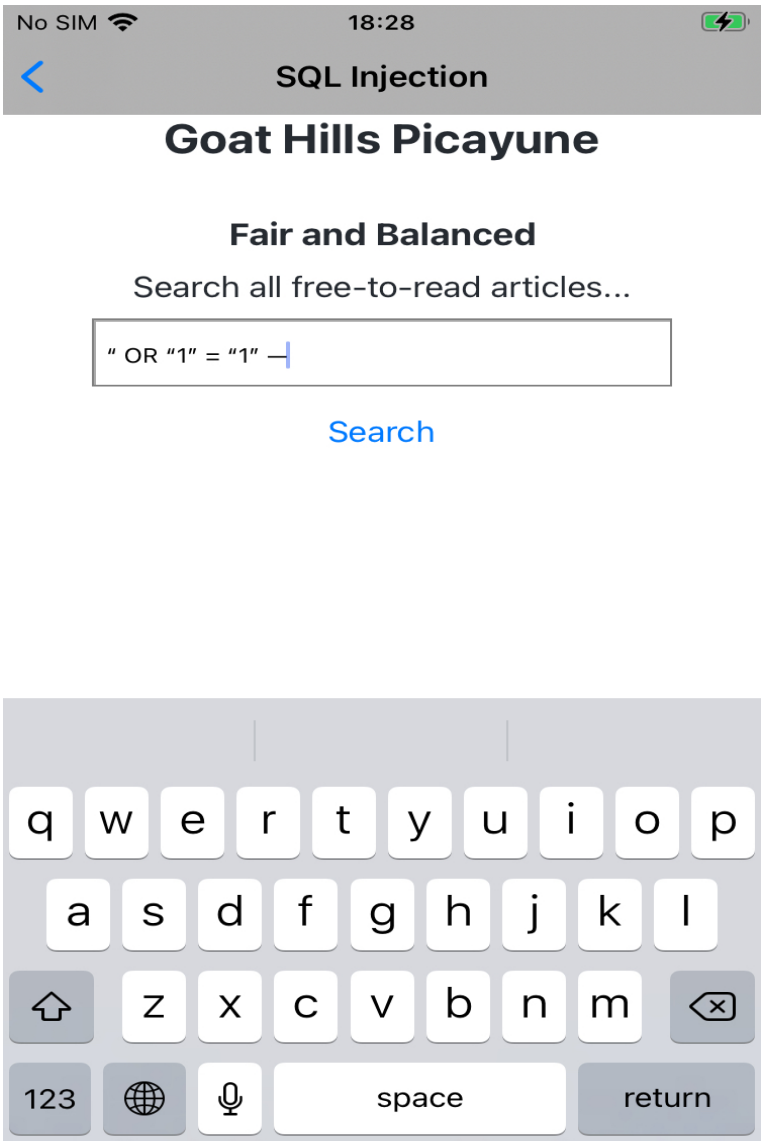
Slika 22: Dostupni članci iz baze podataka (Izvor: OWASP, 2017)

Rezultati normalne pretrage vraćaju dostupne članke, međutim isti su označeni kao "Free" što upućuje na postojanje plaćenih ili premium članaka. Znatiželjni napadač zatim kreće u akciju. Pošto se u većini slučajeva na iOS uređajima koristi SQLite baza podataka dovoljno je da je napadač upoznat sa sintaksom i naredbama koje se koriste u toj bazi podataka. Zatim može započeti napad. Napad će u većini slučajeva biti vrlo jednostavne prirode. Napadač će probati izlistati sve tražene zapise pojedine tablice kojoj vidi da ima pristup kroz dostupnu tražilicu. Tako u ovom slučaju želi izlistati sve zapise tablice "article". Naredba koju će napadač unijeti mogla bi izgledati ovako:

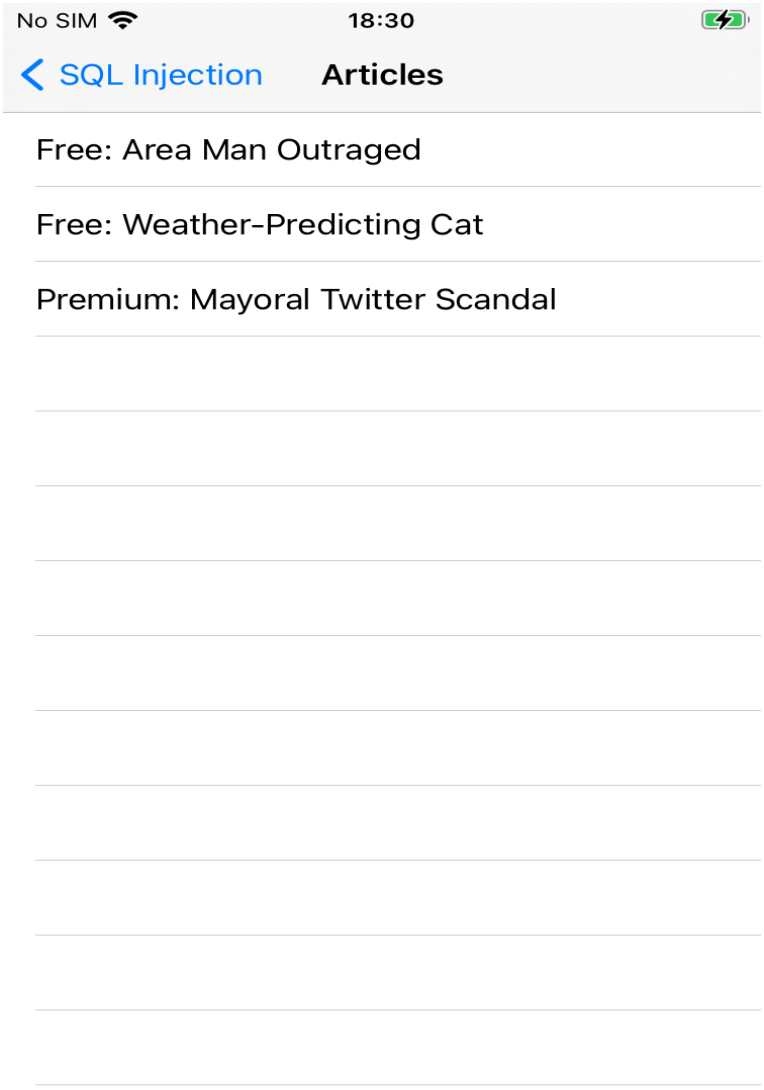
```
' OR  '1' = '1' --
```

Prvi navodnik na početku upita služi da zavara aplikaciju da kao unosnu riječ krene gledati zapis koji slijedi tek poslije njega. Tako prva naredba koja će se gledati prilikom unosa stringa

je logički operator ILI. Napadač može pretpostaviti da baza kod izvođenja upita radi neki oblik SELECT-a kojim dohvaća zapise iz tablice. Dodamo li na istu logički operator ILI nakon kojeg slijedi tvrdnja koju će uređaj interpretirati kao točnom, kompletan upit će vratiti TRUE tj. istinu i u slučaju SELECT upita odabrati sve upite iz selektirane tablice. Tako poslije logičkog ILI slijedi uspoređivanje da li je 1 = 1 što je uvijek točno. Za kraj kako bi konačno nadmudrili uređaj dodaju se dvije crtice koje znače komentiraj liniju, tako svaku sljedeću naredbu ili uvijte nakon crtica uređaj neće gledati kao dio naredbe. Unesemo li taj upit u nedovoljno sigurnu tražilicu dobivamo prikaze sa slike 23 i slike 24.



Slika 23: Nesigurna tražilica odnosno (engl. *Injection point*) (Izvor: OWASP, 2017)



Slika 24: Uspješno proveden SQL Injection, dohvaćen nedostupni sadržaj) (Izvor: OWASP, 2017)

Najbolja prevencija SQL Injectiona je validacija korisničkog unosa, bilo kroz definirani regex koji provjerava korisnikov upit i ne dozvoljava pretraživanje ako isti nije zadovoljen ili zabranu unosa specijalnih znakova u tražilice. Odabrani način zaštite treba prilagoditi poslovnoj logici aplikacije i namjeni pojedine tražilice.

7.3. Penetracijsko testiranje aplikacije DVIA

DVIA posložena je slično kao i iGoat. Sastoji se od početnog ekrana i bočnog izbornika s pojedinim zadacima. Odabrana je kao dodatak iGoat aplikaciji pošto iGoat nije dugo održavan te su neke njegove funkcionalnosti zastarjele i nije ih bilo moguće analizirati. Tako će se pojedine bitni dijelovi testiranja vezani uz analizu mrežnog prometa, *jailbreak* detekciju i *certificate pining* provesti nad DVIA aplikacijom.

7.3.1. Analiza podatkovnog prometa u iOS aplikacijama

Danas se svaka aplikacija u nekom trenutku spaja na internet i izmjenjuje podatke preko mreže koristeći HTTP, odnosno HTTPS protokol. To je čini ranjivom na *Men in the middle* mrežne napade i *Packet sniffing*, odnosno nadgledavanje njenog mrežnog prometa. U nastavku bit će opisan proces testiranja i zaštite mrežne komunikacije unutar aplikacije

Svaka aplikacija koja komunicira s udaljenim serverom mora nekako proslijediti podatke na server. Sljedeće poglavlje prikazat će razliku između slanja podataka HTTP protokolom s i bez TLS zaštitnog sloja. Također iOS u sebi ima i dodatnu zaštitu koja je a priori aktivna, i uključuje zabranu spajanje i slanje podataka na server koji nema implementiran TLS. Ta zaštita je za potrebe testiranja i rada ugašena. Dodatnu zaštitu koja je postavljena kao osnovna od iOS-a 13 čini točno definirana regulativa kada se pojedini TLS certifikat smatra validnim i kada ga iOS aplikacija ne odbacuje. Tako se smanjuje mogućnost lažnog predstavljanja drugog servera kao ciljanog servera. Osim same regulative za kreiranje validnih certifikata, postoji još i tako zvani "Certificate pining" pomoću kojeg aplikacija najčešće kroz integraciju javnog ključa pojedinog certifikata prihvaća i spaja se samo na server koji sadrži certifikat čiji javni ključ odgovara javnom ključu spremljenom u aplikaciji, a odbacuje sve ostale certifikate, nebitno da li su isti od sustava označeni kao oni kojima se vjeruje (engl. *trusted*) ili ne. [15]

iOS aplikacija će pojedini TLS certifikat smatrati valjanim ako su zadovoljeni sljedeći uvjeti:

- TLS certifikat izdan od strane CA (engl. *Certificate authority*) mora koristiti RSA ključeve minimalne veličine 2048 bita
- Algoritam za heširanje certifikata mora spadati u SHA-2 obitelj hash algoritama
- TLS certifikat mora sadržavati DNS servera u polju "Subject Alternative Name". Također polje "CommonName" mora sadržavati ime DNS servera
- TLS certifikat mora sadržavati "ExtendedKeyUsage (EKU)" ekstenziju koja sadrži jedinstveni identifikator servera (*id-kp-serverAuth <OID>*)
- TLS certifikat mora biti validan maksimalno 825 dana

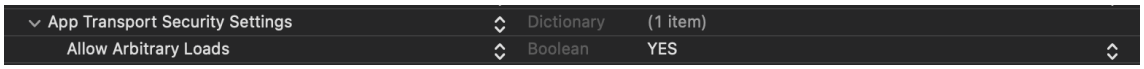
ExtendedKeyUsage (EKU) oblik je ekstenzije certifikata preko koje se definira koja je točno upotreba pojedinog certifikata. Na primjer, određuje hoće li se pomoću certifikata vršiti autentifikacija korisnika ili servera. Tip upotrebe definira se imenom pojedine vrijednosti nakon koje se dodaje OID (engl. *Object identifier*) pojedinog čvora u mreži na kojeg se restrikcija odnosi. Mogući oblici restrikcija prikazani su u tablici 1. [16]

Tablica 1: ExtendedKeyUsage vrijednosti i značenja

Vrijednost	Značenje
serverAuth	SSL/TLS Web Server Authentication.
clientAuth	SSL/TLS Web Client Authentication.
codeSigning	Code signing.
emailProtection	E-mail Protection (S/MIME).
timeStamping	Trusted Timestamping
OCSPSigning	OCSP Signing
ipsecIKE	ipsec Internet Key Exchange
msCodeInd	Microsoft Individual Code Signing (authenticode)
msCodeCom	Microsoft Commercial Code Signing (authenticode)
msCTLSign	Microsoft Trust List Signing
msEFS	Microsoft Encrypted File System

(Izvor: OpenSSL, 2018)

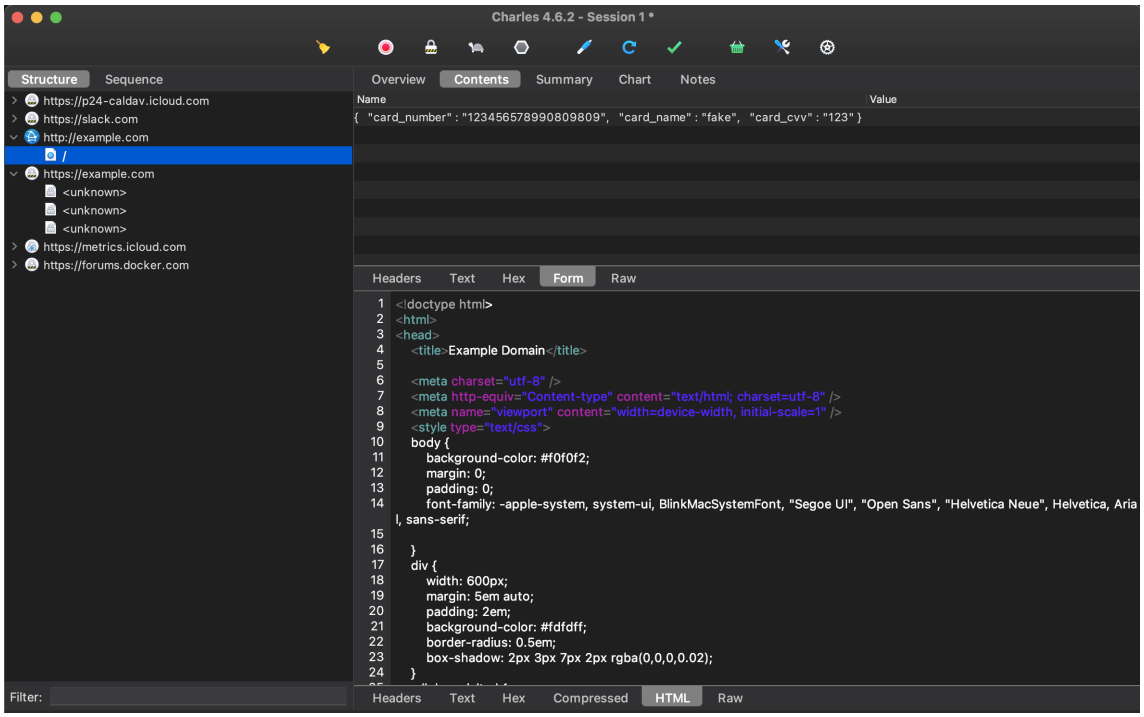
Nakon teorijskog pojašnjenja vrijeme je za analizu i prikaz razlika između slanja podataka mrežom s i bez TLS certifikata. Kako bi se na iOS-u omogućila komunikacija sa serverom bez TLS certifikata potrebno je u Info.plist datoteci projekta postaviti parametar "NSAllowsArbitraryLoads" na *true*. Unutar Info.plist potrebno je dodati novi red "App Transport Security Settings" i unutar toga dodati opciju "AllowsArbitraryLoads" i njenu vrijednost postaviti na YES.



Slika 25: Omogućavanje komunikacije sa serverom koji nema TLS (Izvor: autor, 2021)

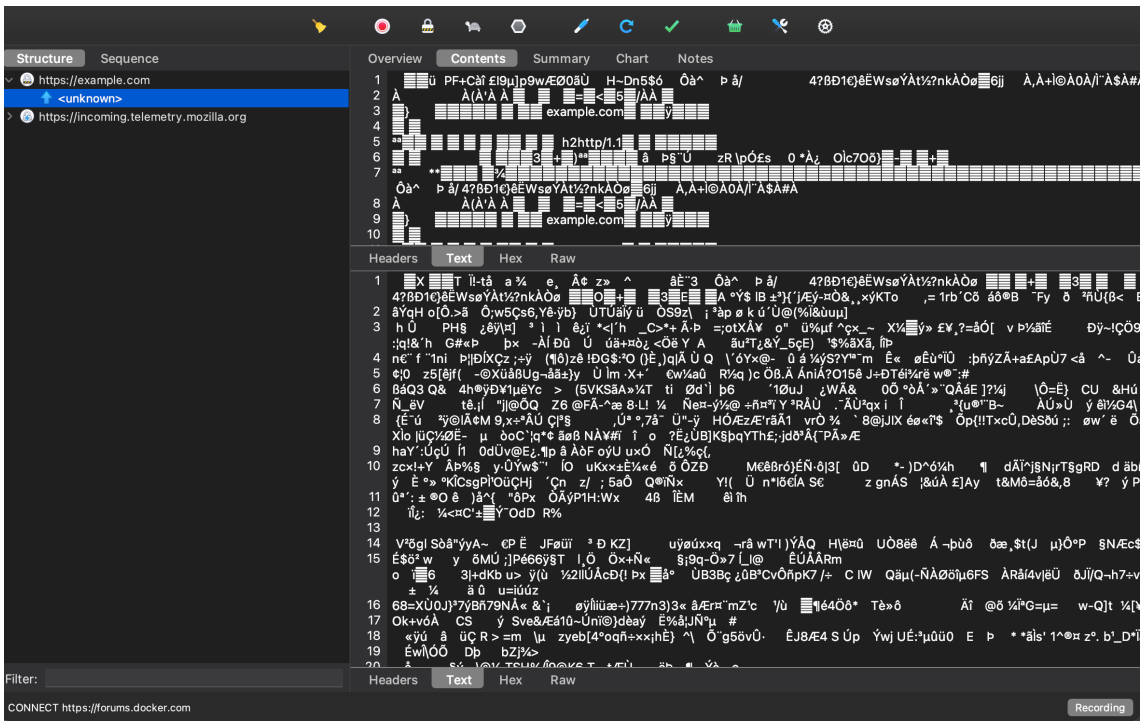
Za analizu mrežnog prometa najbolje je uspostaviti posrednika (engl. *Proxy*) koji će se izvoditi na nekom lokalnom računalu te će se podaci iz aplikacije slati na posrednika koji će ih preusmjeravati na server. Tako se može steći dobar uvid u podatke koji izlaze iz aplikacije, i njihovu zaštitu, a isto tako može se testirati da li se server ispravno ponaša dođe li do izmjene podataka koji se šalju. U ovom slučaju kao posrednika koristit će se *Charles Proxy*. [17]

Za testiranje mrežnog prometa unutar DVIA aplikacije potrebno je otići u bočni izbornik pod opciju "Network Layer Security" te na novo otvorenom ekranu unijeti tražene informacije i odabrati način slanja istih. Ako se za način slanja povjerljivih informacija unesenih na ekranu odabere HTTP, a napadač u tom trenutku analizira mrežni promet isti će imati sve vidljive informacije, kao što prikazuje slika 26.



Slika 26: Slanje podataka putem čistog HTTP-a (Izvor: autor, 2021)

Dok ako se koristi HTTPS, tj. HTTP s TLS-om napadač se ipak mora više potruditi oko dobivanja željenih informacija, pošto su iste kriptirane, slika 27.



Slika 27: Slanje podataka putem HTTPS-a (Izvor: autor, 2021)

7.3.2. Certificate pining

Certificate pining je proces referenciranja pojedinog *hosta* (računala/aplikacije unutar mreže) s njegovim certifikatom ili javnim ključem. Nakon što se utvrdi certifikat ili javni ključ pojedinog *hosta*, isti se "pina" tj. dodaje na onog *hosta* s kojim se želi omogućiti komunikacija. Javni ključ ili certifikat se može dodati u aplikaciju prilikom razvoja ili prilikom prvog nailaska na isti u mreži. *Pinning* se oslanja na činjenicu da se u naprijed zna s kojim serverom klijent komunicira temeljem certifikata ili javnog ključa kojeg pojedini server može dekriptirati.

Usadi li se javni ključ pri razvoju u aplikaciju, hash istog se hardkodira u aplikaciju. Zatim se pri svakom pozivu na server provjerava da li javni ključ iz aplikacije odgovara javnom ključu servera, ako je tako dozvoljava se spajanje sa serverom, inače se zahtjev odbacuje. Kod oblika dodavanja prilikom prvog nailaska, javni se ključ sprema u aplikaciju prilikom uspostavljanja prvog poziva na server. Pri tome server "odašilje" svoj javni ključ i ako isti nije spremljen u aplikaciji, odnosno aplikacija ga ne prepoznaje traži se dozvola o spremanju novog javnog ključa u aplikaciju. Drugi način smatra se nesigurnijim pošto je teoretski moguće da napadač presretne prvi poziv na server i u tom trenutku aplikaciju preusmjeri na svoj server i proslijedi joj vlastiti javni ključ. [18], [19]

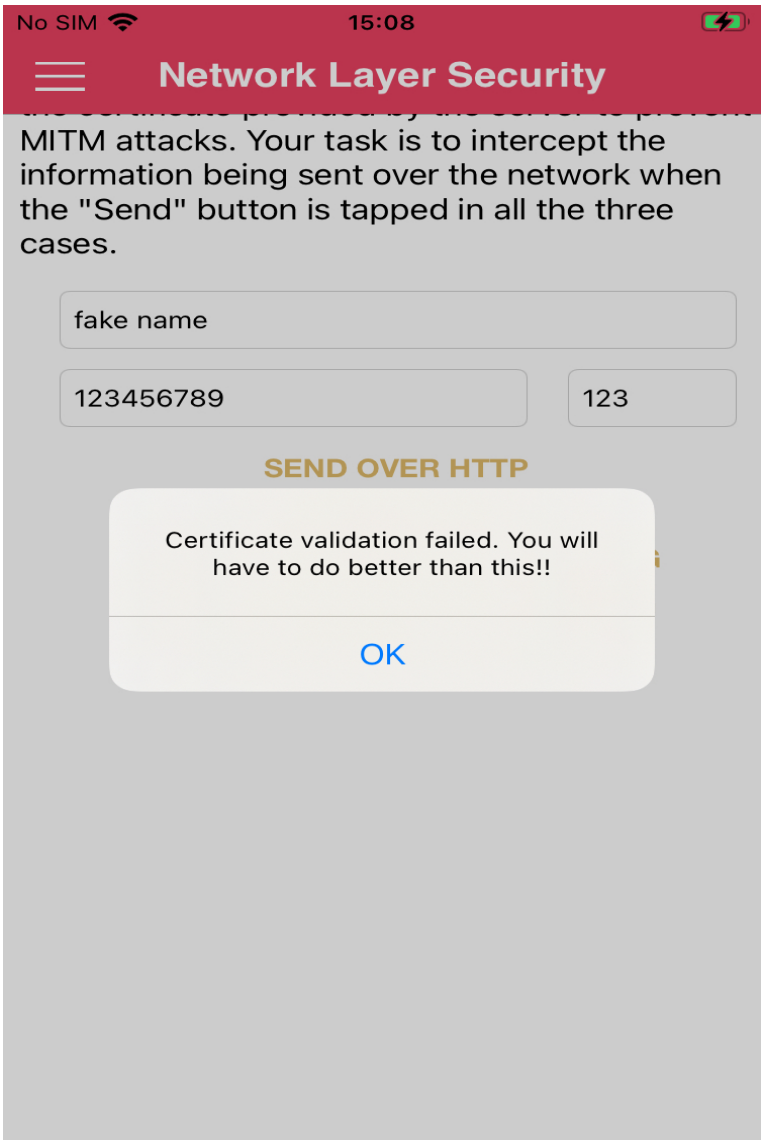
Certificate pining na iOS uređaju moguće je provesti kroz *NSURLConnection* klasu i njezine delegacijske metode, tako u nastavku slijedi prikaz jednostavnog *certificate pininga* koristeći *NSURLConnection* klasu. Primjer se odnosi na DVIA aplikaciju i obuhvaća certifikate te hardkodirani javni ključ.

```
func connection(_ connection: NSURLConnection, willSendRequestFor challenge:
    URLAuthenticationChallenge) {
    //Provjerava da li veza na koju se klijent spaja ima certifikat od
        povjerenja, ako nema izlazi iz metode.
    guard let serverTrust = challenge.protectionSpace.serverTrust else { return
        }
// Preuzmi certifikat ukoliko isti postoji certifikat od povjerenja, činae đizai
    .
    guard let certificate = SecTrustGetCertificateAtIndex(serverTrust, 0) else {
         return }
// Lokalno preuzmi podatke o certifikatu
    let remoteCertificateData = SecCertificateCopyData(certificate)
    if isSSLPinning { // zastavica koja se postavlja prije poziva metode
        isSSLPinning = false
        // Lokalno spremi podatke o preuzetom javnom certifikatu sa servera *
            Napomena provjeriti da aplikacija u sebi žsadri valjani certifikat
            preuzet sa servera
        let skabberCertificateData = NSData(contentsOfFile: Bundle.main.path(
            forResource: "example", ofType: "der")!)
        if remoteCertificateData == skabberCertificateData {
        // javni certifikati servera i klijenta su ispravni
            DVIAUtilities.showAlert(title: "", message: "Request␣Sent␣using␣
                Certificate␣pinning,␣lookout!", viewController: self)
            let credential = URLCredential(trust: serverTrust)
            // proslijedi poziv na server sa ispravnim podacima za
                autentifikaciju
```

```
                challenge.sender?.use(credential, for: challenge)
            } else {
                DVIAUtilities.showAlert(title: "", message: "Certificate␣validation␣
                    failed.␣You␣will␣have␣to␣do␣better␣than␣this!!", viewController:
                     self)
                challenge.sender?.cancel(challenge)
            }
        }
        //Ukoliko se radi o pinanju škoritenjem javnog čkljua
        if isPublicKeyPinning && !isSSLPinning {
            isPublicKeyPinning = false
            if let serverCertificate = SecTrustGetCertificateAtIndex(serverTrust, 0)
                 {
                if #available(iOS 10.3, *) {
                    let serverPublicKey = SecCertificateCopyPublicKey(
                        serverCertificate)
                    let serverPublicKeyData:NSData =
                        SecKeyCopyExternalRepresentation(serverPublicKey!, nil )!
                    let keyHash = sha256(data: serverPublicKeyData as Data)
                    if (keyHash == pinnedPublicKeyHash) {
                        DVIAUtilities.showAlert(title: "", message: "Request␣Sent␣
                            using␣Public␣Key␣Pinning,␣lookout!", viewController:
                            self)
                        let credential = URLCredential(trust: serverTrust)
                        challenge.sender?.use(credential, for: challenge)
                        return
                    }else{
                        DVIAUtilities.showAlert(title: "", message: "Certificate␣
                            validation␣failed.␣You␣will␣have␣to␣do␣better␣than␣this
                            !!", viewController: self)
                        challenge.sender?.cancel(challenge)
                    }
                } else {
                    DVIAUtilities.showAlert(title: "", message: "This␣feature␣is␣not
                        ␣yet␣supported␣on␣older␣ios␣versions", viewController: self)
                    challenge.sender?.cancel(challenge)
                }
            }
        }
    }
```

7.3.3. Zaobilaženje Certificate pininga

Pokretanjem DVIA aplikacije i odlaskom pod "Network Layer Security" otvara se ekran za testiranje *certificate pininga*. Pokušajem slanja podataka na server s uključenim *certificate piningom* vraća poruku sa slike 28.



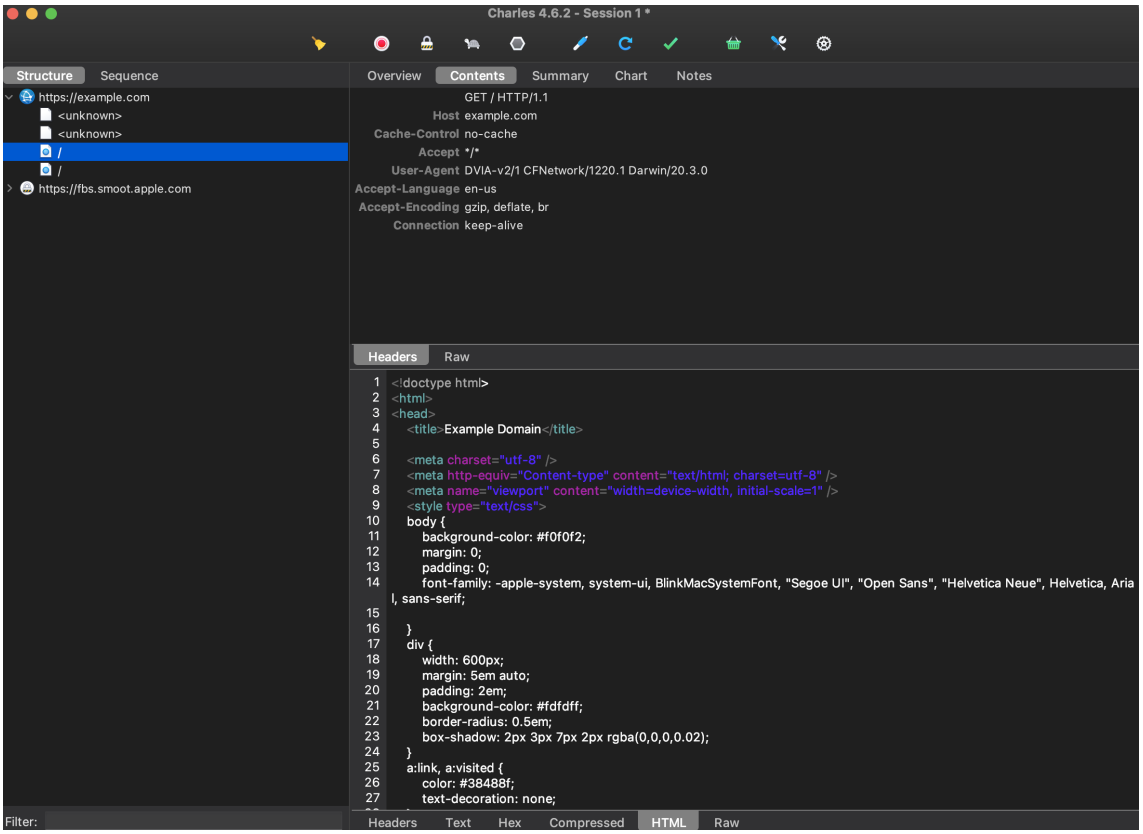
Slika 28: Neuspješno spajanje zbog neispravnog certifikata (Izvor: Gianchandani, 2018)

Zaobilaženje *certifiacte pininga* provesti će se koristeći Fridu i Objection. Pošto se u radu koristi *jailbreakani* iOS uređaj, kako bi se dobio pristup aplikaciji potrebno je SSH-at u uređaj i na njemu pokrenuti instalirani Frida server. Zatim s naredbom `objection -gadget com.highaltitudehacks.DVIAswift-1 explore` pokrenuti aplikaciju i preuzeti kontrolu nad njom te nakon uspješnog spajanja pokrenuti naredbu `ios sslpinning disable`. Uspješnim pokretanjem alata dobiva se sadržaj prikazan na slici 29.

```
[tab] for command suggestions
....highaltitudehacks.DVIAswift-1 on (iPhone: 14.4) [usb] # ios sslpinning disab
le
(agent) Hooking common framework methods
(agent) Found NSURLSession based classes. Hooking known pinning methods.
(agent) Hooking lower level SSL methods
(agent) Hooking lower level TLS methods
(agent) Hooking BoringSSL methods
(agent) Registering job 227338. Type: ios-sslpinning-disable
....highaltitudehacks.DVIAswift-1 on (iPhone: 14.4) [usb] # (agent) [227338] Cal
led SSL_CTX_set_custom_verify(), setting custom callback.
(agent) [227338] Called custom SSL context verify callback, returning SSL_VERIFY
_NONE.
```

Slika 29: Pokretanje ios sslpinning disable (Izvor: autor, 2021)

Nakon pokretanja i ponovnim odlaskom na "Network Layer Security" snimanjem mrežnog prometa moguće je zamijetiti sljedeći odgovor od strane servera:



Slika 30: Uspješno snimanje sadržaja mrežnog prometa nakon izbjegavanja Certificate pininga (Izvor: autor, 2021)

Iz primjera je vidljivo koliko je danas jednostavno zaobići *certificate pining* i *public key pining*. Zbog čega nije dovoljno implementirati "pining" kao jedinu zaštitu od Napada predusretanja mrežnog prometa (engl. *Man in the middle (MITM)*). Naravno to sve ovisi o tipu aplikacije koja se razvija i stupnju sigurnosti kojeg aplikacija mora zadovoljavati.

7.3.4. Jailbreak detekcija

U poglavlju "Jailbreak iOS uređaja" pojašnjeno je što je to *jailbreak*. Ovdje će se provesti analiza načina detekcije *jailbreakanog* uređaja, ali i načini suzbijanja detekcije. Načini suzbijanja bit će analizirani kako bi se bolje upoznalo s radom iOS sustava, alatima i načinima s kojima je moguće zavarati uređaj. Time će se pokazati činjenica da detekcija *jailbreaka* čini samo jednu malu sigurnosnu nadogradnju koju je poželjno ugraditi, ali se aplikacija koja u sebi ima ugrađenu *jailbreak* detekciju ne smije de facto smatrati sigurnom.

Jailbreak sam po sebi ne čini uređaj nesigurnim. Međutim postoje dva glavna razloga zašto se rade provjere za isti i zašto se *jailbreakani* uređaj može smatrati manje sigurnim. Prvi razlog smatra da pokretanje ne autorizirane aplikacije na uređaju, pogotovo ako se ista pokreće van aplikacijskog okruženja (engl. *sandbox*) uvelike otežava definiranje sigurnosnih svojstava uređaja. Dok drugi kaže da korisnici *jailbreakanih* uređaja često ne ažuriraju svoje uređaje pošto je *jailbreak* ograničen na pojedine verzije operacijskog sustava, čime uređaj dovode u neminovnu opasnost i ograničavaju njegovu funkcionalnost. Uzevši to u obzir prvi se korak provjere sigurnosti pojedinog uređaja često svodi na *jailbreak* detekciju i provjeru verzije operacijskog sustava. [21]

Postoje razne tehnike provjera *jailbreaka*. Iste često nisu javne kako bi se osiguralo od prijevremenih slučajeva zaobilaženja. Prijevremenih, jer je *jailbreak* i detekcija istog stalna igra mačke i miša od strane razvojnih inženjera koji ga žele razotkriti i onih koji ga žele uspješno sakriti. Zbog čega je malo vjerojatno da se s vremenom neće doći do načina zaobilaženja pojedinog mehanizma otkrivanja *jailbreaka*. U nastavku bit će prikazani najčešći oblici *jailbreak* detekcije i neka od zaobilaženja istih. Većina *jailbreak* detekcija spadaju u sljedeće kategorije:

- Provjera datotečnog sustava i prava aplikacije unutar istog
- Provjera prijavljenih URI shema
- Provjera odgovora pojedinih sistemskih API-a
- Inspekcija dinamičkih linkova

Kroz provjeru datotečnog sustava provjerava se jesu li dodane nove datoteke na uređaj koje su isključivo vezane za *jailbreak*. Poput alternativnih trgovina aplikacija, primjerice Cydia. Sadrži li uređaj `/Applications/Cydia.app` može ga se klasificirati kao *jailbreakanog*. Ostale česte datoteke i direktoriji uključuju:

- `/private/var/stash`
- `/private/var/lib/apt`
- `/private/var/tmp/cydia.log`
- `/private/var/lib/cydia`
- `/private/var/mobile/Library/SBSettings/Themes`

- /Library/MobileSubstrate/MobileSubstrate.dylib
- /Library/MobileSubstrate/DynamicLibraries/Veency.plist
- /Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
- /System/Library/LaunchDaemons/com.ikey.bbot.plist
- /System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
- /var/cache/apt
- /var/lib/apt
- /var/lib/cydia
- /var/log/syslog
- /var/tmp/cydia.log
- /usr/sbin/sshd
- /usr/libexec/ssh-keysign
- /usr/sbin/sshd
- /usr/bin/sshd
- /usr/libexec/sftp-server
- /etc/ssh/sshd_config
- /etc/apt
- /bin/bash
- /bin/sh
- /Applications/Cydia.app
- /Applications/RockApp.app
- /Applications/Icy.app
- /Applications/WinterBoard.app
- /Applications/SBSettings.app
- /Applications/MxTube.app
- /Applications/IntelliScreen.app
- /Applications/FakeCarrier.app
- /Applications/blackra1n.app

Osim provjere postojanja pojedinih datoteka i direktorija, također dobra provjera je i pravo pristupa pojedinom dijelu datotečnog sustava kojem aplikacija ne bi smjela moći pristupiti. Najjednostavniji primjer bio bi pokušaj kreiranja ili čitanja zapisa unutar *root* direktorija. Uspijeli čitanje ili pisanje van aplikacijskog kontejnera uređaj je *jailbreakan*. Slijedi provjera URI shema. Provjera prijavljenih URI shema uključuje provjeru postojanja shema tipa: `cydia://` . Ako postoji shema koja dozvoljava direktne poveznice s Cydia trgovine, uređaj se svrstava u *jailbreakane*. Developer provjeru *jailbreaka* može provesti i kroz pojedine sistemske API-e. Provjera odgovora pojedinih sistemskih API-a odnosi se na pozive pojedinih Appleovih servisa tipa `fork()` i `syscall()` koji bi oba trebala vratiti nil ako uređaj nije *jailbreakan* ili `dlyId` metoda kao što su `dyld_image_count()` ili `dyld_get_image_name()` koje vraćaju broj učitanih dinamičkih biblioteka odnosno njihova imena. Ako je broj veći od očekivanog ili sadrži nepoznate dinamičke biblioteke uređaj se smatra *jailbreakanim*. Zadnja provjera smatra se validnom i spada u inspekciju dinamičkih linkova, pošto česti načini zaobilaženja *jailbreak* detekcije uključuju učitavanje pojedinih dinamičkih biblioteka u aplikaciju koje u sebi sadrže metode za zaobilaženje *jailbreak* detekcije. [21]–[23]

Za kraj poglavlja prikazat će se pojedini načini prevare jailbreak detekcije. Uglavnom su svi načini slični i sastoje se od četiri glavna koraka:

1. Zamijeni svaku metodu koja izgleda kao da otkriva *jailbreak* s posebnom inačicom.
2. Kada se pokrene nova inačica metode provjeri da li se kroz tu metodu provjerava *jailbreak*.
3. Ako je to slučaj, kao povratni rezultat iz metode vrati "false".
4. Ako to nije slučaj, dozvoli provedbu originalne metode.

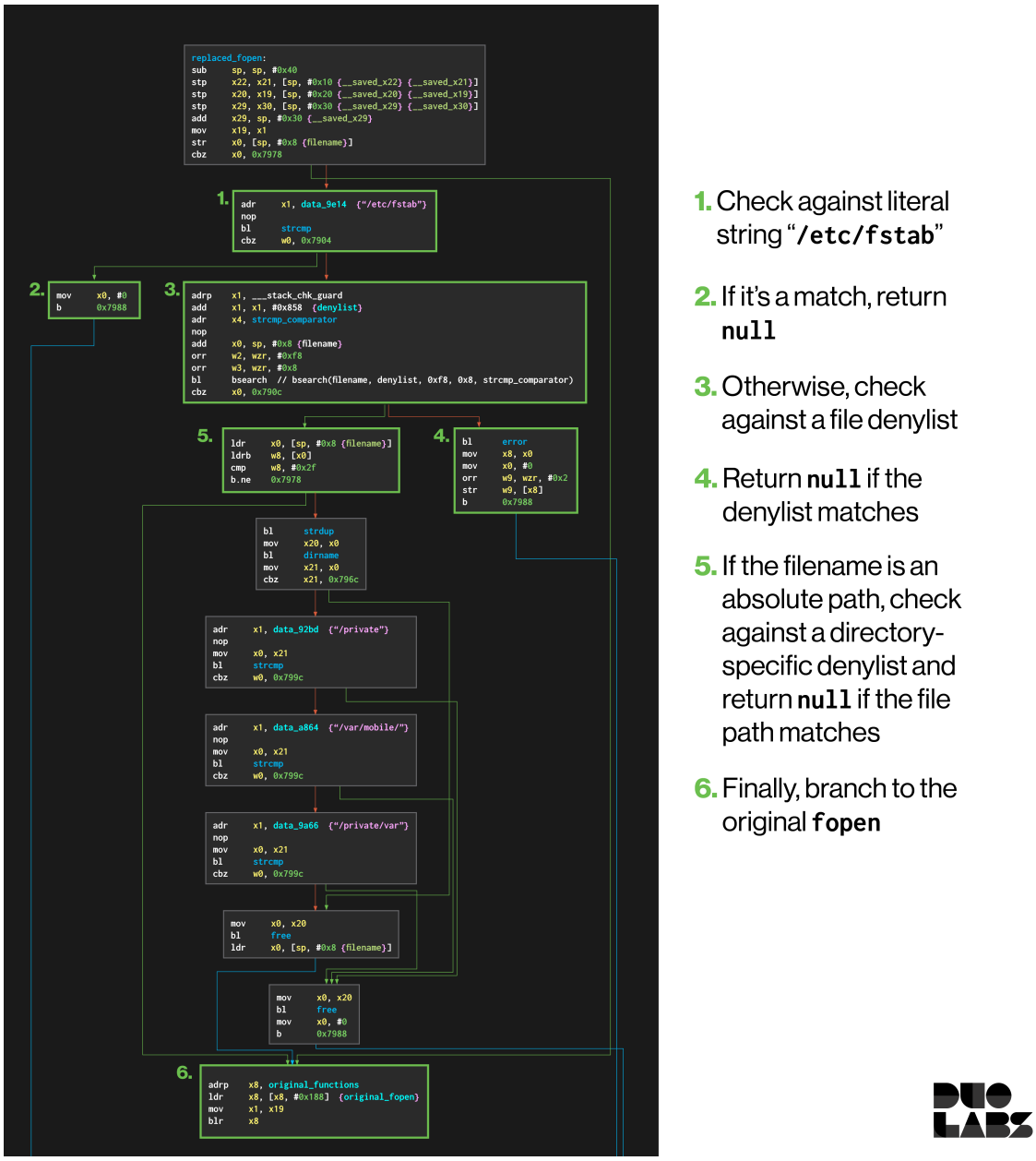
Objective-C, a i Swift, ako je metoda označena sa @dynamic ili @objc pozivaju metode i šalju njihove rezultate u obliku poruka tijekom izvođenja. Odnosno svaka metoda koja se definira istovremeno definira i objekt kojem se prosljeđuje rezultat izvedene metode, pošto je u Objective-C-u sve objekt, pa čak i klasa. Takav način pozivanja metoda omogućuje tako zvani "Method swizzling", odnosno izmjenu metoda i njihovih funkcionalnosti prilikom izvođenja programa. Konkretno prilikom izvršavanja programa može se mijenjati implementacija selektora koji se prosljeđuje odnosno pozove prilikom izvršavanja metode i to tako da se mijenjaju parametri nad kojima se pozivaju sistemske metode `class_addMethod()` ili `method_setImplementation()` [21], [24], [25]

Osim navedenih tu je još i `fileExistsAtPath` metoda, koja se u Objective-C-u koristi za provjeru postojanja dodatnih artefakata (datoteka i direktorija) koji se stvaraju pokretanjem procesa *jailbreakanja*. Zamjenom odgovora `fileExistsAtPath` metode da uvijek vrati "false" za listu poznatih *jailbreak* artefakata jedan je od načina prevencije otkrivanja *jailbreakanog* uređaja.

Drugi načini uključuju manipulaciju i editiranje tako zvane "Linker Table", odnosno vrstu tablice u memoriji koja drži reference na pojedine vanjske artefakte koji se koriste u aplikaciji. Najme, kada se u aplikaciju učita dinamička biblioteka, aplikacija mora znati od kud je pokreće

i od kuda dohvaća njene podatke. Tako na iOS-u ako se pozove metoda `printf` učitana iz dinamičke biblioteke, metoda se ne zove direktno već se zove metoda koja je smještena u `stubs section`. Nju čini dinamička biblioteka koja je učitana u proces koji se alocira tijekom pokretanja aplikacije. Na toj memorijskoj adresi nalazi se `jmp` naredba na adresu unutar `la_symbol_ptr` (engl. *lazy symbol pointers*) ili `nl_symbol_ptr` (engl. *non-lazy symbol pointers*), u kojem je zapisana `printf` naredbe unutar dinamičke biblioteke. Nakon čega se konačna adresa metode `printf` zapisuje u tablicu pokazivača kako bi aplikacija znala od kuda pozvati koju metodu. [21], [26]

Kroz navedeni primjer vidljivo je da ukoliko se želi promijeniti funkcionalnost `printf` metode, potrebno je zamjeniti samo adresu s koje se `printf` metoda poziva iz tablice pokazivača. Na taj način kada se pronađe metoda koja bi mogla služiti za *jailbreak* provjeru, istoj je moguće dodijeliti drugu funkciju.



1. Check against literal string "**/etc/fstab**"
2. If it's a match, return **null**
3. Otherwise, check against a file denylist
4. Return **null** if the denylist matches
5. If the filename is an absolute path, check against a directory-specific denylist and return **null** if the file path matches
6. Finally, branch to the original **fopen**

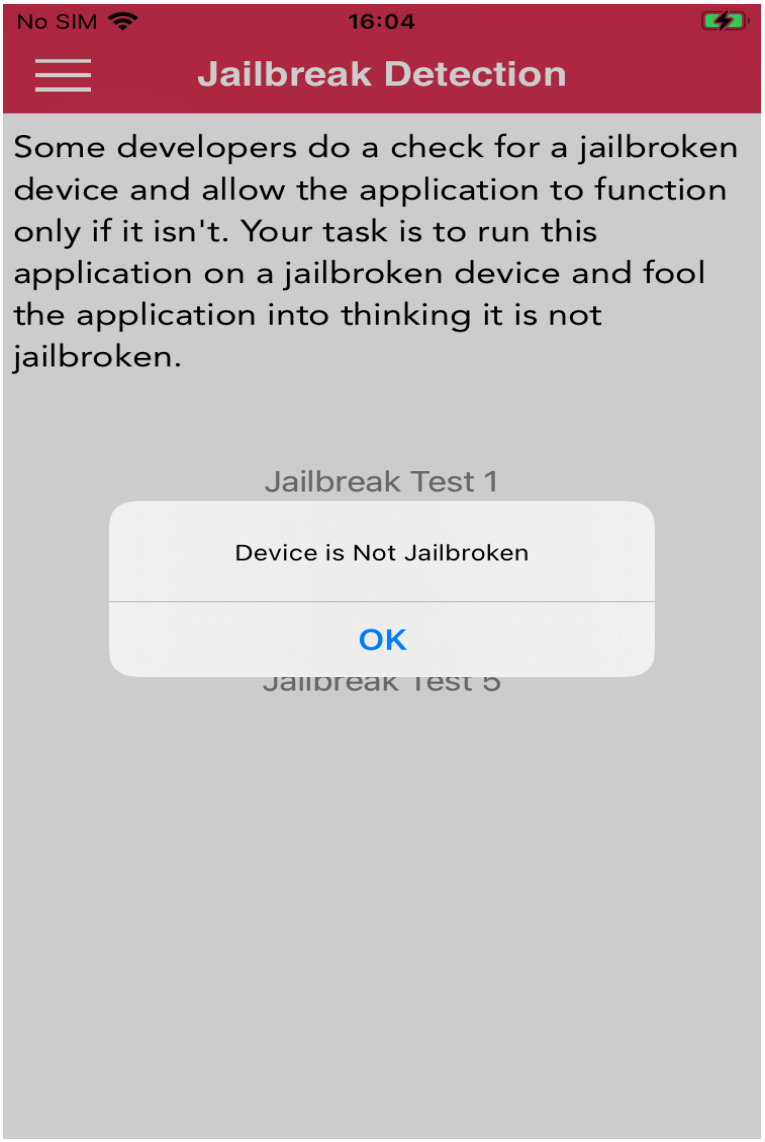
Slika 31: Primjer izmjene funkcionalnosti fopen (Izvor: Mooney, 2019)

Iz slike 30 vidljiva je izmjene `fopen` metode čija je originalna zadaća otvaranje datoteke na proslijeđenoj lokaciji. Završetkom poziva metoda vraća referencu na otvoreni "file handler" ili null ako nije moguće otvoriti datoteku na traženoj lokaciji. Ukoliko `fopen` vrati vrijednost pokušavajući otvoriti znani *jailbreak* artefakt, uređaj je sigurno *jailbreakan*. Primjer sa slike međutim provjerava putanje do poznatih *jailbreak* artefakata smještenih u listi i vraća null ako se isti pronađe. Time se postiže zavaravanje metode. Metoda misli da pojedini artefakti ne postoje. Također ako se pokuša otvoriti datoteka koja nije navedena u listi, ista će se uspješno otvoriti. Dodatne metode koje se može modificirati kako bi se zavarala *jailbreak* detekcija su `fork()`, koju je moguće navesti da stalno vraća -1 ili 0, kao indikaciju da forkanje nije dozvoljeno i tako stvoriti privid da aplikacija ne izlazi iz svog kontejnera. [21]

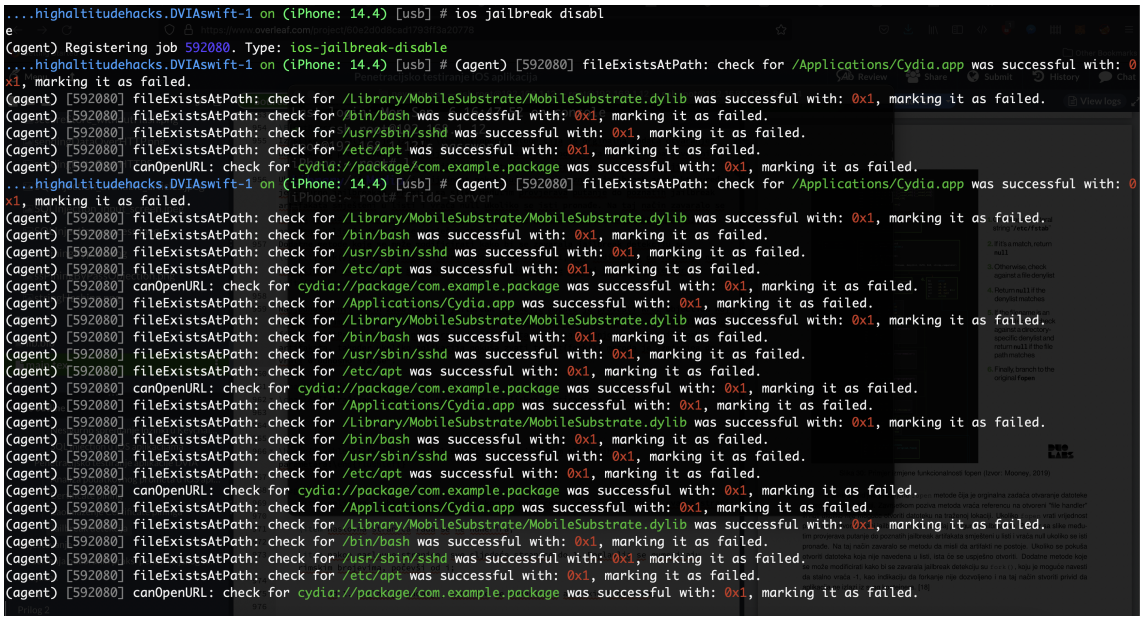
Najjednostavniji način zaobilaženja *jailbreak* detekcije moguće je provesti kroz *Objec-*

tion koji u sebi već ima "ios jailbreak disable" opciju koja se zasniva na izmjeni `NSFileManager fileExistsAtPath` metode i provjerava da li se ista zvala s putanjom znanih *jailbreak* artefakata, također ako aplikacija pokuša zvati textttfork() metoda je izmijenja da vrati 0 odnosno "false".

Na sljedećim slikama bit će prikazana uspješno zavaravanje DVIA aplikacije da uređaj nije *jailbreakan* (Slika 31) te ispis pojedinih putanja koje su uspješno maskirane (Slika 32).



Slika 32: DVIA uspješno zavarana jailbreak detekcija (Izvor: Gianchandani, 2018)



```
....highaltitudehacks.DVIAswift-1 on (iPhone: 14.4) [usb] # ios jailbreak disabl
e
(agent) Registering job 592080. Type: ios-jailbreak-disable
....highaltitudehacks.DVIAswift-1 on (iPhone: 14.4) [usb] # (agent) [592080] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0
x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [592080] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
....highaltitudehacks.DVIAswift-1 on (iPhone: 14.4) [usb] # (agent) [592080] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0
x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [592080] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [592080] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [592080] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [592080] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [592080] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
```

Slika 33: Objection ios jailbreak disable - uspješno maskirane putanje jailbreak artifakata (Izvor: autor, 2021)

Pregledom izvornog koda *Objection jailbreak* klase potvrđuju se već navedeni načini zaobilaženja *jailbreaka*. U nastavku slijedi opis konkretnog koda.

```
// U čpoetku definiraju se putanje do pojedinih jailbreak artifakata

const jailbreakPaths = [
  "/Applications/Cydia.app",
  "/Applications/FakeCarrier.app",
  "/Applications/Icy.app",
  "/Applications/IntelliScreen.app",
  "/Applications/MxTube.app",
  "/Applications/RockApp.app",
  "/Applications/SBSetttings.app",
  "/Applications/WinterBoard.app",
  "/Applications/blackra1n.app",
  "/Library/MobileSubstrate/DynamicLibraries/Veency.plist",
  "/Library/MobileSubstrate/MobileSubstrate.dylib",
  "/System/Library/LaunchDaemons/com.ikey.bbot.plist",
  "/System/Library/LaunchDaemons/com.saurik.Cy@dia.Startup.plist",
  "/bin/bash",
  "/bin/sh",
  "/etc/apt",
  "/etc/ssh/sshd_config",
  "/private/var/stash",
  "/private/var/tmp/cydia.log",
  "/usr/bin/cycript",
  "/usr/bin/ssh",
  "/usr/bin/sshd",
  "/usr/libexec/sftp-server",
  "/usr/libexec/sftp-server",
  "/usr/libexec/ssh-keysign",
```

```
  "/usr/sbin/sshd",
  "/var/cache/apt",
  "/var/lib/cydia",
  "/var/log/syslog",
  "/var/tmp/cydia.log",
];

export namespace iosjailbreak {

  // Prepisuje fileExistsAtPath: fza putanje navedene unutar "jailbreakPaths"
  const fileExistsAtPath = (success: boolean, ident: string): InvocationListener =>
      {

    return Interceptor.attach(
      ObjC.classes.NSFileManager["- fileExistsAtPath:"].implementation, {
        onEnter(args) {

          // Zastavica za provjeru da li je potrebno manipulirati povratnu
              vrijednost metode prilikom izlaska iz iste
          this.is_common_path = false;

          // Lokalno pospremi čuitanu putanju kako bi se je moglo đusporeivati sa
              listom putanji
          this.path = new ObjC.Object(args[2]).toString();

          // provjeri žsadri li "jailbreakPaths" čuitanu putanju
          if (jailbreakPaths.indexOf(this.path) >= 0) {

            // čOznai putanju kao validnu
            this.is_common_path = true;
          }
        },
        onLeave(retval) {

          // Prekini šizvravanje i đizai iz metode ukoliko putanja nije
              klasificirana kao validna
          if (!this.is_common_path) {
            return;
          }

          // Ovisno o željenom stanju izmjenjujemo vrijednost koja se ćvraa iz
              metode
          switch (success) {
            case (true):
              // Igorira se šuspjeno žpronalaenje putanje, odnosno ukoliko je
                  ćvraena pojedina putanja izlazi se iz metode
              if (!retval.isNull()) {
                return;
              }
              send(
                c.blackBright(`[${ident}] `) + `fileExistsAtPath: check for ` +
                c.green(this.path) + ` failed with: ` +
                c.red(retval.toString()) + `, marking it as successful.`,
```

```
            );
          // Ukoliko je đpronaen jailbreak artifakt  zamijeni povratnu vrijednost
              metode
          // i đizai iz metode, vrati "false".
            retval.replace(new NativePointer(0x01));
            break;

          case (false):
            // Ukoliko šnita nije đpronaeno đizai iz metode, odnosno ignoriraj
                takav čsluaj.
            if (retval.isNull()) {
              return;
            }
            send(
              c.blackBright(`[${ident}] `) + `fileExistsAtPath: check for ` +
              c.green(this.path) + ` was successful with: ` +
              c.red(retval.toString()) + `, marking it as failed.`,
            );

            retval.replace(new NativePointer(0x00));
            break;
        }
      },
    },
  );
};


// Prepisuje fopen: za putanje definirane u  "jailbreakPaths"
const fopen = (success: boolean, ident: string): InvocationListener => {

  return Interceptor.attach(
    Module.findExportByName(null, "fopen"), {
      onEnter(args) {

        this.is_common_path = false;


        this.path = args[0].readCString();


        if (jailbreakPaths.indexOf(this.path) >= 0) {

          this.is_common_path = true;
        }
      },
      onLeave(retval) {

        if (!this.is_common_path) {
          return;
        }

        switch (success) {
```

```
          case (true):
            // ignore successful lookups
            if (!retval.isNull()) {
              return;
            }
            send(
              c.blackBright(`[${ident}] `) + `fopen: check for ` +
              c.green(this.path) + ` failed with: ` +
              c.red(retval.toString()) + `, marking it as successful.`,
            );

            retval.replace(new NativePointer(0x01));
            break;

          case (false):
            // ignore failed lookups
            if (retval.isNull()) {
              return;
            }
            send(
              c.blackBright(`[${ident}] `) + `fopen: check for ` +
              c.green(this.path) + ` was successful with: ` +
              c.red(retval.toString()) + `, marking it as failed.`,
            );

            retval.replace(new NativePointer(0x00));
            break;
        }
      },
    },
  );
};

// Prepisuje žpronalaenje Cydia URI Sheme
const canOpenURL = (success: boolean, ident: string): InvocationListener => {

  return Interceptor.attach(
    ObjC.classes.UIApplication["- canOpenURL:"].implementation, {
      onEnter(args) {

        this.is_flagged = false;

        // Extract the path
        this.path = new ObjC.Object(args[2]).toString();

        if (this.path.startsWith('cydia') || this.path.startsWith('Cydia')) {
          this.is_flagged = true;
        }
      },
      onLeave(retval) {

        if (!this.is_flagged) {
          return;
```

```
        }

        switch (success) {
          case (true):
            if (!retval.isNull()) {
              return;
            }
            send(
              c.blackBright(`[${ident}] `) + `canOpenURL: check for ` +
              c.green(this.path) + ` failed with: ` +
              c.red(retval.toString()) + `, marking it as successful.`,
            );

            retval.replace(new NativePointer(0x01));
            break;

          case (false):
            // ignore failed
            if (retval.isNull()) {
              return;
            }
            send(
              c.blackBright(`[${ident}] `) + `canOpenURL: check for ` +
              c.green(this.path) + ` was successful with: ` +
              c.red(retval.toString()) + `, marking it as failed.`,
            );

            retval.replace(new NativePointer(0x00));
            break;
        }
      },
    },
  );
};

  // ćVraa "false" ukoliko se špokuava pozvati fork unutar aplikacije
const libSystemBFork = (success: boolean, ident: string): InvocationListener => {

  const libSystemBdylibFork: NativePointer = Module.findExportByName("libSystem.B.
      dylib", "fork");

  return Interceptor.attach(libSystemBdylibFork, {
    onLeave(retval) {

      switch (success) {
        case (true):
          // already successful forks are ok
          if (!retval.isNull()) {
            return;
          }
          send(
            c.blackBright(`[${ident}] `) + `Call to ` +
            c.green(`libSystem.B.dylib::fork()`) + ` failed with ` +
```

```
            c.red(retval.toString()) + ` marking it as successful.`,
          );

          retval.replace(new NativePointer(0x1));
          break;

        case (false):
          if (retval.isNull()) {
            return;
          }
          send(
            c.blackBright(`[${ident}] `) + `Call to ` +
            c.green(`libSystem.B.dylib::fork()`) + ` was successful with ` +
            c.red(retval.toString()) + ` marking it as failed.`,
          );

          retval.replace(new NativePointer(0x0));
          break;
      }
    },
  });
};
}
```

Analizirani kod za *jailbreak* detekciju preuzet je sa službene GitHub stranice alata Objection (`https://github.com/sensepost/objection/blob/master/agent/src/ios/jailbreak.ts`)

8. Istraživanja odabranih tehnika penetracijskog testiranja u praksi

Istraživanje odabranih tehnika penetracijskog testiranja u praksi oslonit će se na analizu izvještaja penetracijskog testiranja triju sigurnosnih ustanova, "Rhino Security Labs", "NowSecure" i "AppSec Labs". Od svakog poduzeća preuzeti je izvještaj penetracijskog testiranja pojedine aplikacije koji će se analizirati i usporediti sa sigurnosnim testovima provedenim u radu. Od strane "Rhino Security Labs" analizirat će se projekt "Mobile Application Penetration Test Assessment Report" koji njihovim klijentima daje uvid u tipove testiranja koji se provađaju nad klijentskim aplikacijama. Prikazan je kao izvještaj koji opisuje ranjivost te istu klasificira po stupnju rizika koji je pronađen pojedinim testom. Također upućuje na ispravnosti i mane pronađene unutar aplikacije te daje preporuku za poboljšanje sigurnosti. Nadalje "NowSecure" proveo je testiranje Concordiumovog crypto novčanika, kojeg je podijelio na dva glavna dijela: Sažetak pronalaska i Preporuke sprječavanja te detaljna analiza pojedinih ranjivosti. Dok je "AppSec Labs" proveo penetracijsko testiranje nad KZenovim ZenGo crypto novčanikom, koji su svoje testiranje podijelili na definiranje metodologije testiranja, kroz koju su definirali okvir testiranja i njegove limitacije te generalizirali stupanj prijetnji pojedine ranjivosti.

8.1. Rhino Security Labs

Sigurnosna je kompanija iz Seattlea, SAD. Bavi se mrežnim, cloud (Amazon Web Services - AWS), web i mobile penetracijskim testiranjem. Posjeduju renomirane klijente iz raznih grana industrije, od auto industrije (Ford), ugostiteljstva (Burger King), bankarstva (InvestCloud, Firdt National Bank) do sveučilišta (University of Meriland) i poduzeća u sigurnosnom sektoru zaštite podataka (Datto). [27] Sljedeće izvješće penetracijskog testiranja provedenog od strane Rhino Security Labsa moguće je preuzeti sa `https://rhinosecuritylabs.com/landing/penetration-test-report/`. U nastavku slijedi njegova analiza.

Kroz svoje sigurnosno testiranje Rhino Security Labs vodi se sljedećom metodologijom:

1. Izviđanje
2. Automatizirano testiranje
3. Eksploatacija i Verifikacija
4. Izvještavanje
5. Sanacija

Izviđanje je korak otkrivanja što je više moguće informacija o aplikaciji koja dolazi na testiranje. Provode ga automatizirani skeneri, poput Nmap-a, kao i dohvaćanje otiska aplikacije (engl. *Application fingerprint*). Pod time se smatra ustanoviti koje servise koristi pojedina aplikacija, na

kojem se operacijskom sustavu izvodi te s kojim tipom servera komunicira. Zatim slijedi automatizirano testiranje, odnosno korak provođenja aplikacije kroz svojevrsne skenere ranjivosti, zbog bržeg otkrivanja mogućih ranjivosti koje se kasnije ispituju. Svaka ranjivost se dodatno ručno provjerava kako bi se izbjeglo lažno očitavanje. Pomoću dobivenih rezultata zatim se provodi eksploatacija i verifikacija, koje podrazumijevaju manualnu sigurnosnu analizu aplikacije. U konačnici slijedi izvještavanje, tj. prezentiranje pronalazaka penetracijskog testiranja klijentu, zajedno sa savjetima za ispravak. Izvješće se sastoji od sažetka, definiranja pojedinih jakosti i mana aplikacije, prikaza konkretnih ranjivosti s najkritičnijima pri vrhu te koracima prezentiranja i iskorištavanja uočenih ranjivosti zajedno sa prikazom kompromitiranih sadržaja. Nakon izvještavanja Rhino Security Labs klijentu nudi i dodatni izborni korak sanacije. Odnosno proces ponovnog testiranja pojedinosti aplikacije nakon ispravka grešaka od strane klijenta, kako bi se osiguralo ispravno ponašanje.

Prolaskom kroz izvješće vidljivo je da Rhino Security Labs klasificira sigurnosne ranjivosti ovisno o jednostavnosti i vjerojatnosti provođenja pojedinog sigurnosnog napada te utjecaju pojedinog napada na poslovanje poduzeća. Nakon pojašnjenja načina klasifikacije ranjivosti slijede sažeci dobrih i loših strana aplikacije. Tako je u ovom konkretnom slučaju testiranja zamijećeno da aplikacija ima kvalitetno implementiranu korisničku autentifikaciju i autorizaciju te da ista ima implementiran sigurnosni mehanizam protiv "stack smashinga"[1]. S druge strane primijećeno je da aplikacija "propušta" osjetljive podatke kroz pojedine metode te da nema implementiranu niti *jailbreak* detekciju niti *certificate pining*. Temeljem zamijećenog predlaže se implementacija *jailbreak* detekcije i *certificate pininga*.

Rizike i definicije pojedinih ranjivosti podijelili su u pet kategorija. Prvu kategoriju pri vrhu čini Kritična kategorija, u nju spadaju one ranjivosti koje su krucijalne u smislu same sigurnosti organizacije, odnosno njihovim iskorištavanjem moguće je nanijeti trajnu štetu aplikaciji ili čak i samoj organizaciji kroz aplikaciju. Zatim slijedi Visoka kategorija. Slično kao i kod kritične, no za razliku od kritične za koju se smatra da je treba trenutno riješiti, ranjivosti koje spadaju u visoku kategoriju imaju najviši prioritet unutar radnog dana poduzeća i ne zahtijevaju trenutačnu reakciju. Srednja kategorija sadrži one ranjivosti koje nisu često na prvu prepoznatljive zbog čega ih je često potrebno dodatno pojasniti. Međutim to ne znači da bi ih trebalo ignorirati. Niska kategorija, uključuje ranjivosti koje imaju minimalan utjecaj na okruženje unutar kojeg se promatraju i često su teoretske prirode, u smislu ako se ostvare pojedini uvjeti postoji mogućnost da budu iskoristivi. Njihova sanacija često se "zanemaruje" pošto ne predstavljaju visoku rizičnost za aplikaciju niti organizaciju. Osim navedenih postoji i informativna kategorija, u koju ne spadaju ranjivosti kao tave već sigurnosni rizici koji se smatraju dovoljno malima da ih se može zanemariti. Takvi rizici se ne smatraju ranjivostima, ali u nekom slučaju to mogu postati, zbog čega ih se dodaje u izvješće.

Nakon definiranja podijele slijede konkretne ranjivosti zamijećene u aplikaciji. Pronađene su jedna kritična, tri visoke i dvije srednje kategorije ranjivosti koje se vežu uz iOS aplikaciju. U kritičnu spada nesigurno keširanje osjetljivih podataka, dok su u visokoj nedostatak

[1] Stack Smashing je vrsta ranjivosti pri kojoj se stog pojedine aplikacije natjera na "prelijevanje" (engl. *overflow*) dodavanjem prevelike količine podataka na stog čime se može izazvati rušenje aplikacije. Također dodavanjem proizvoljne količine podataka na stog moguće je dodati i nepoželjni kod čime bi bilo moguće preusmjeriti kontrolni tok programa ili dobiti administratorske ovlasti nad istim. [28]

certificate pininga, dozvola učitavanja stranica i komunikacije sa serverom bez TLS certifikata te personalizirana poruka korisniku kod resetiranja lozinke. Srednju kategoriju zauzimaju izostanak enkripcije podataka spremljenih u lokalnu bazu podataka i izostanak *jailbreak* detekcije. [29]

Moguće je primijetiti da su primijećene ranjivosti i među onima koje su prezentirane u radu. Tako u kritičnu ranjivost može spadati nesigurno spremanje povjerljivih podataka u korisničke preference, obrađeno u istoimenom poglavlju ili u *Keychain*, ili u ne kriptiranu bazu podataka. Dok su navedene visoke prijetnje obrađene u poglavljima "Certificate pining" i "Analiza podatkovnog prometa iOS aplikacija". Što se tiče personalizirane poruke korisniku, takva ranjivost nije konkretno provedena u radu, međutim ista je definirana kao ranjivost pošto bi sve poruke koje se tiču vraćanja lozinki pojedinih korisnika trebale biti generičke, kako bi se napadač teže mogao predstaviti kao tražena osoba. Takav tip napada spada u socijalni inženjering i lažno predstavljanje. Od srednjih prijetnji ako se analizira čuvanje lokalnih podataka u ne kriptiranoj bazi isto može postati i visoka ili čak kritična ranjivost ako se lokalno spremaju pojedini povjerljivi podaci. Smatra se preporukom da niti jedna mobilna aplikacija koja radi s bilo kakvim povjerljivim podacima ne bi iste trebala čuvati na uređaju te ako aplikacija mora neke podatke čuvati na uređaju iste je potrebno kriptirati i osigurati. Više informacija o dohvaćanju nekriptiranih podataka iz lokalne baze podataka opisano je u istoimenom poglavlju rada. Konačna ranjivost koja je ostala je izostavak *jailbreak* detekcije, koje kao što je to i Rhino Security Labs dobro primijetio nije niti visoka niti kritična ranjivost pošto nužno ne mora dovoditi aplikaciju niti korisnika u opasnost. Međutim često korisnici koji imaju *jailbreakane* uređaje imaju i zastarjele verzije operacijskog sustava unutra kojeg je u međuvremenu moglo doći do otkrivanja pojedinih novih ranjivosti koje mogu naštetiti korisniku i eventualno aplikaciji i poduzeću bez da je korisnik toga svjestan. Zbog čega se preporučuje da svaka aplikacija implementira jailbreak detekciju i po potrebi ograniči dostupnost svoje aplikacije.

8.2. NowSecure

Sigurnosna je kompanija iz Chicaga, SAD. Specijalizirana u sigurnosti mobilnih uređaja i aplikacija, s klijentima poput Ubera, Shella, Emerson, Habit Master Consulting i drugim. U ovom dijelu rada analizirat će se Izvješće penetracijskog testiranja poduzeća NowSecure nad Concordiumovoj Mobile Wallet aplikaciji.

Okvir unutar kojeg je aplikacija testirana uključuje analizu podataka koju aplikacija ostavlja tijekom instalacije i korištenja na uređaju. Ista uključuje provjeru da li su pojedini podaci kriptirani, heširani ili spremljeni u čistom tekstu, o kakvom se tipu podatka radi i gdje su spremljeni na uređaju (baza podataka, Keychain, korisničke preference). Nadalje se radi analiza biometrijske autentifikacije te provjera povjerljivih podataka unutar memorije. Nakon analize lokalnih podataka slijedi analiza mrežnog prometa. Ona uključuje analizu protokola s kojima aplikacija komunicira preko mreže, način slanja osjetljivih informacija preko mreže, ispravno rukovanje sesijama ili kolačićima prilikom prijave i odjave korisnika te ispravnu evaluaciju identificiranja certifikata i dvo faktorsku autorizaciju. Dalje slijedi analiza bekenda, koja uključuje provjeru stupnja sigurnosti korištene kriptografije na serveru, provjeru ispravnog načina autori-

zacije klijenta kod komunikacije sa servisima, i način rukovanja sesijama i korisničkim podacima od strane servera. Za kraj ostaje reverzibilni inženjering aplikacije i provjera obfuskacije koda, provjera implementacije zabrane debugiranja aplikacije nakon izdavanja, pokušaj napada na slabu kriptografiju i vanjske biblioteke koje se koriste u aplikaciji, kao i zaobilaženje biometrije i certificate pininga.

Kao i kod prijašnjeg izvještaja stupanj ranjivosti dijeli se u pet kategorija: na kritičnu, visoku, srednju, nisku i informastivnu. Pri čemu u kritičnu spadaju one ranjivosti koje su krucijalne u smislu same sigurnosti organizacije, odnosno njihovim iskorištavanjem moguće je nanijeti trajnu štetu aplikaciji ili čak i samoj organizaciji kroz aplikaciju. Zatim slijedi Visoka kategorija. Slično kao i kod kritične, no za razliku od kritične za koju se smatra da je treba trenutno riješiti, ranjivosti koje spadaju u visoku kategoriju imaju najviši prioritet unutar radnog dana poduzeća i ne zahtijevaju trenutačnu reakciju. Srednja kategorija sadrži one ranjivosti koje nisu često na prvu prepoznatljive zbog čega ih je često potrebno dodatno pojasniti. Međutim to ne znači da bi ih trebalo ignorirati. Niska kategorija, uključuje ranjivosti koje imaju minimalan utjecaj na okruženje unutar kojeg se promatraju i često su teoretske prirode, u smislu ako se ostvare pojedini uvjeti postoji mogućnost da budu iskoristivi. Njihova sanacija često se "zanemaruje" pošto ne predstavljaju visoku rizičnost za aplikaciju niti organizaciju. Osim navedenih postoji i informativna kategorija, u koju ne spadaju ranjivosti kao tave već sigurnosni rizici koji se smatraju dovoljno malima da ih se može zanemariti. Takvi rizici se ne smatraju ranjivostima, ali u nekom slučaju to mogu postati, zbog čega ih se dodaje u izvješće. Uzevši to u obzir u Mobile Wallet aplikaciji došlo je do sljedeća dva srednja propusta:

- iOS aplikacija ne koristi zadan "AppTransportSecurity" dodjeljen od strane operacijskog sustava.
- iOS apliakcija dozvoljava uporabu moguće malicioznih prilagođenih tipkovnica

i tri niska:

- iOS aplikacija koristi uobičajan način zaštite podataka u mirovanju
- iOS aplikacija dozvoljava snimanje ekrana dok korisnik koristi aplikaciju i kešira zadnju sliku dok aplikacija ide u pozadinu.
- iOS aplikacija koristi obične UITexfield elemente čime potencijalno može pospremiti povjerljive podatke unutar keša tipkovnice. [30]

Činjenica da aplikacija ne koristi zadani "AppTransportSecurity" moguć je izvor problema pošto onemogućavanje istog dozvoljava komunikaciju sa serverima bez TLS certifikata i serverima koji imaju zastarjelu verziju TLS-a, što samu aplikaciju čini pogodnom za "Man in the Middle" napad i "ARP Poisoning". Činjenica da je trenutna prijetnja od strane organizacije označena kao srednja upućuje na dobru kontrolu podataka od strane servera. Više o analizi podatkovnog prometa iOS aplikacija moguće je pročitati u istoimenom poglavlju. Daljnja ranjivost je ne ograničavanje tipkovnice na sistemsku tipkovnicu uređaja već dozvoljavanje korištenja prilagođenih tipkovnica. Kada je riječ o sigurnijim aplikacijama koje provode transakcije i novčane

transakcije, poželjno je koristiti što manje vanjskih biblioteka te ograničiti korištenje prilagodljivih elemenata poput tipkovnice, kako ne bi došlo do mogućeg prikupljanja podataka kroz neku vrstu keyloggera ili zapisivanja svih unosa u poseban cache prilagođene tipkovnice te slanje istog preko mreže trećoj strani. Zanimljivo je da je poduzeće oba gornja slučaja klasificirala kao srednje opasna, iako oba mogu dovesti do relativno brzog gubitka povjerljivih informacija i prevare korisnika i to još u aplikaciji koja bi trebala sadržavati novčana sredstva. Nadalje korištenje uobičajene zaštite podataka u mirovanju znači da podaci ovisno o tome gdje su spremljeni vrlo vjerojatno nisu kriptirani ako korisnik koristi aplikaciju ili ima otključani ekran. S druge strane istu ranjivost moguće je klasificirati kao nisku, pošto da bi napadač uspio doći do podataka ipak se mora podosta toga posložiti. Napadač mora preoteti uređaj, te isti mora uspješno otključati, preuzeti aplikaciju, modificirati istu i tek onda ispisati željene podatke. Zbog čega takav tip napada možemo svrstati u teorijski i dati mu niski prioritet. Sljedeća niska prijetnja je omogućavanje snimanja ekrana i ostavljanje vidljivog posljednjeg ekrana iz aplikacije ako se ista nalazi u pozadini. Navedena ranjivost također postaje tek vektor napada ako korisnik dođe u posjed korisnikovog uređaja. Posljednja niska ranjivost je korištenje običnog unosnog polja unutar aplikacije koja radi s povjerljivim podacima. Također ukoliko bi napadač došao u posjed uređaja, iz cachea tipkovnice koji uglavnom sadrži utipkane riječi po kojima se AutoCorrect uči ponašati, moguće je ekstrahirati upisane podatke. [30]

8.3. AppSec Labs

Sigurnosna je kompanija iz Izraela koja se bavi penetracijskim testiranjem desktop, web i mobile aplikacija kao i treniranjem profesionalaca u području računalne sigurnosti. Neki od njihovih klijenata su: Intel, HP, AT‘I&’T, AVG i Motorola. Trenutno će se u ovom radu analizirati njihov javno dostupan izvještaj o njihovom načinu penetracijskog testiranja ZenGo iOS aplikacije.

Metodologija korištena tijekom testiranja je "Gray Box" testiranje. Kombinacija "White Box" i "Black Box" testiranja, odnosno testeri su svjesni nekih unutarnjih struktura i načina rada sustava. Testiranje je provedeno ručno i koristeći automatizirane alate, a od posebnih resursa testeri su imali mogućnost intervjuiranja zaposlenika i pristup izvornom kodu aplikacije i servera. Okvir unutar kojeg se obavljalo testiranje bila je konkretna verzija aplikacije i korespondirajućeg bekenda, a korisnik je bio vlastiti/ novi korisnik registriran od strane AppSec Labsa.Zanimljivo je primjetiti kako AppSec Labs unutar aplikacije nije pronašao nikakvih nedostataka ili ranjivosti, a testovi koji su bili provedeni prikazani su u sljedećoj tablici. [31]

Tablica 2: Provedeni testovi na ZenGo iOS aplikaciji

Kategorija	Ime testa
Sakupljanje informacija	Search engine discovery / reconnaissance Web application fingerprint Review Webpage Comments and Metadata for Information Leakage Application entry points Identification Execution paths mapping Web application framework fingerprinting Web application fingerprinting Application architecture mapping Information Disclosure by error codes SSL Weakness - SSL/TLS Testing (SSL Version, Algorithms, Key length, Digital Cert. Validity)
Testiranje konfiguracije i postavljanja servera	Application Configuration management weakness File extensions handling - sensitive information Old, Backup and Unreferenced Files - Sensitive Information Unauthorized Admin Interfaces access HTTP Methods enabled, XST permitted, HTTP Verb Http strict transport security RIA cross domain policy Role definitions enumeration Vulnerable user registration process Vulnerable account provisioning process Permissions of Guest/Low Permission Accounts Account suspension/resumption process
Testiranje autentifikacije	Credentials Transported over Unencrypted Channel User enumeration Account lockout Authentication bypass "Remember password" functionality Browser caching Weak password policy Weak password security mechanisms Weak password change or reset flow Race conditions Weak multiple factors authentication Weak CAPTCHA implementation Weaker authentication in alternative channel

Testiranje autorizacije	Directory traversal/file inclusion Authorization schema bypass
Upravljanje sesijama	Privilege escalation Insecure direct object references Session Management Testing Session management bypass Cookies are set without 'HTTP Only', 'Secure', and no time validity Session fixation Exposed session variables Cross site request forgery (CSRF) Logout management Session timeout Session puzzling
Validacija podataka	Reflected cross site scripting Stored cross site scripting HTTP verb tampering HTTP Parameter pollution / manipulation SQL injection LDAP injection ORM injection XML injection SSI injection Xpath Injection IMAP/SMTP injection Code injection Local/remote file inclusion Command injection Buffer overflow Heap overflow Stack overflow Format string manipulation Incubated vulnerabilities HTTP splitting/smuggling
Rukovanje greškama	Analysis of Error Codes Analysis of Stack Traces
Kriptografija	Weak SSL/TLS ciphers, insufficient transport layer protection Padding oracle Sensitive information sent via unencrypted channels
Testiranje poslovne logike	Business logic data validation Ability to Forge Requests Integrity checks Process timing Replay attack Circumvention of Work Flows Abuse of Functionality File upload vulnerabilities

Testiranje klijenta	DOM based Cross Site Scripting Javascript Execution Html/css injection Client side url redirect Client side resource manipulation
Testiranje nedostupnosti usluge	Cross origin resource sharing Cross site flashing Clickjacking / UI rendering Web sockets Web messaging Local storage / session storage sensitive information SQL Wildcard vulnerability Locking customer accounts Buffer overflows User specified object allocation User Input as a Loop Counter Writing User Provided Data to Disk Failure to Release Resources Storing too Much Data in Session
Testiranje web servisa	WS information gathering WSDL weakness Weak xml structure XML content-level WS HTTP GET parameters/REST WS Naughty SOAP attachments WS replay testing

(Izvor: Hakon, 2019)

Imena samih testova nisu prevedena, već su ostavljeni u originalu zbog lakšeg snalaženja kod pretraživanja pojedinog provedenog testa, ali i zbog velike količine prisutnih testova. Što samo pokazuje koliko je područje sigurnosti mobilnih uređaja ogromno i koliko je potrebno savladati da bi se osoba mogla prozvati sigurnosnim stručnjakom. Zanimljivo je usporediti testove i količinu testova koji su provedeni u prva dva slučaja analize tehnika penetracijskog testiranja u praksi i zadnji test, koji je proveden od strane jedne od najboljih svjetskih sigurnosnih poduzeća, što potvrđuje i klijentela s kojom surađuju. Ovaj rad ne može ni približno pokriti sve moguće slučajeve i rigorozne kriterije koje jedna financijska aplikacija mora proći u AppSec Labsu, ali to samo ukazuje na kolike sve raznolikosti treba biti spreman i o čemu je sve potrebno razmišljati kako bi se na tržište dovela sigurna aplikacija.

9. Zaključak

Rad je prikazao strukturu i sigurnosni model iOS operacijskog sustava pri čemu su razrađeni i pojašnjeni njegovi unutarnji mehanizmi i slikovito prikazana struktura i arhitektura sustava. Fascinantno je koliko je Apple uložio truda da već od najniže razine svojeg sustava, isti osigura u najboljoj mogućoj mjeri od najveće opasnosti svakog sustava,a to je neuki korisnik. Svoje je proizvode do te mjere predefinirao da prosječan korisnik nije u mogućnosti niti preuzeti neovlaštenu aplikaciju, a kamoli poremetiti unutarnju strukturu sustava. Međutim, kao i svaki sustav i iOS ima mane koje dolaze na vidjelo kada se netko želi bolje upoznati s njim. Tako se kreće dalje na osnovne tipove sigurnosnog testiranja iOS aplikacija. Pri tome se dotiče sigurnosnih propusta unutar samog operacijskog sustava, kao što su *jailbreak* pomoću kojeg je preuzeta kontrola nad promatranim aplikacijama i omogućena detaljnija analiza samih aplikacija, ali i datotečnog sustava koji sam po sebi nije u potpunosti dostupan krajnjem korisniku. Proučavanjem datotečnog sustava uočilo se da je vrlo jednostavno iz pojedinih aplikacija preuzeti veliku većinu podataka, zbog čega se već u naprijed ne preporuča spremanje bilo kakvih povjerljivih podataka na iOS uređaj. Također je potrebno sve podatke dodatno osigurati prilikom njihovog slanja mrežom i ograničiti komunikaciju između više aplikacija. Nadalje obrađena su dva najkorištenija alata za manipulaciju aktivnim aplikacijama i pokretanje vanjskog koda za vrijeme provođenja aplikacije. Frida i Objection kao alati za sigurnosnu analizu stvarno su istu podigli na drugu razinu, dajući tražene rezultate u kratkom vremenu uz maksimalnu jednostavnost korištenja. Također su oba alata kvalitetno dokumentirani. Nadalje je definiran pojam penetracijskog testiranja, a u praktičnom dijelu prikazane su neke od glavnih prijetnji vezanih uz iOS aplikacije, a to su: nesigurno skladištenje podataka, *SQL injection*, analiza podatkovnog prometa, *certificate pining*, *jailbreak* detekcija i zaobilaženje iste te je pojašnjeno kako ih prepoznati i izbjeći. Za kraj analizirana su tri svjetska sigurnosna poduzeća i njihovi sigurnosni izvještaji, kako bi se pokazalo kako se navedene tehnike iz rada uistinu i koriste u praksi, ali isto tako kako bi se ukazalo na opću širinu samog područja penetracijskog testiranja iOS aplikacija i sigurnosne prijetnje s kojima se svaka aplikacija svakodnevno treba nositi.

Sigurnosnih prijetnji je bilo i bit će ih. Na nama je da ih upoznamo i pokušamo smanjiti njihovu pojavu u našim sustavima. S time da svi moramo shvatiti da nikada neće nestati. Jedino što nam preostaje je diviti se ljudskoj kreativnosti i novim sigurnosnim propustima koji će se pojavljivati sa svakim novim sustavom.

Popis literature

[1] *Mobile Security Testing Guide*, verzija SNAPSHOT July 24, 2021, The OWASP Foundation, srpanj 2021.

[2] *Mobile Application Security Verification Standard*, verzija 1.3, The OWASP Foundation, 2021.

[3] Lucideus. (2021.). „Taint analysis," adresa: https://thecyberwire.com/glossary/taint-analysis. (preuzeto: 24.07.2021).

[4] „PiOS: Detecting Privacy Leaks in iOS Applications," *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, The Internet Society, 2011. adresa: https://www.ndss-symposium.org/ndss2011/pios-detecting-privacy-leaks-ios-applications-paper.

[5] Apple. (2021.). „Apple Platform Security," adresa: https://support.apple.com/guide/security/welcome/web. (preuzeto: 24.07.2021).

[6] S. Chauhan. (2017.). „iOS architecture," adresa: https://www.dotnettricks.com/learn/xamarin/understanding-xamarin-ios-build-native-ios-app. (preuzeto: 24.07.2021).

[7] Apple. (2018.). „Apple Application sandbox," adresa: https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html. (preuzeto: 24.07.2021).

[8] A. Bouchard. (2019.). „Understanding untethered, semi-untethered, semi-tethered, and tethered jailbreaks," adresa: https://www.idownloadblog.com/2019/11/21/types-of-jailbreaks/. (preuzeto: 30.07.2021).

[9] O. A. V. Ravnås. (2021.). „Frida, Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers," adresa: https://frida.re/. (preuzeto: 1.09.2021).

[10] L. J. r. (2017.). „Objection, mobile runtime exploration," adresa: https://sensepost.com/blog/2017/objection-mobile-runtime-exploration/. (preuzeto: 1.09.2021).

[11] SensePost. (2017.). „Objection," adresa: https://github.com/sensepost/objection/wiki. (preuzeto: 1.09.2021).

[12] OWASP. (2017.). „iGoat," adresa: https://github.com/OWASP/igoat. (preuzeto: 4.08.2021).

[13] Apple. (2020.). „kSecValueData," adresa: https://developer.apple.com/documentation/security/ksecvaluedata. (preuzeto: 14.08.2021).

[14] ——, (2021.). „Keychain data protection," adresa: https://support.apple.com/en-gb/guide/security/secb0694df1a/1/web/1. (preuzeto: 14.08.2021).

[15] ——, (2019.). „Requirements for trusted certificates in iOS 13 and macOS 10.15," adresa: https://support.apple.com/en-us/HT210176. (preuzeto: 1.09.2021).

[16] OpenSSL. (2018.). „X509 V3 certificate extension configuration format," adresa: https://www.openssl.org/docs/man1.1.0/man5/x509v3_config.html. (preuzeto: 1.09.2021).

[17] K. von Randow. (2021.). „Charles Proxy," adresa: https://www.charlesproxy.com/. (preuzeto: 1.09.2021).

[18] (2021.). „Certificate and Public Key Pinningy," adresa: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning. (preuzeto: 1.09.2021).

[19] C. Palmer. (2015.). „About Public Key Pinning," adresa: https://noncombatant.org/2015/05/01/about-http-public-key-pinning/. (preuzeto: 1.09.2021).

[20] P. Gianchandani. (2018.). „DVIA," adresa: https://github.com/prateek147/DVIA-v2. (preuzeto: 4.09.2021).

[21] N. Mooney. (2019.). „Jailbreak Detector Detector: An Analysis of Jailbreak Detection Methods and the Tools Used to Evade Them," adresa: https://duo.com/blog/jailbreak-detector-detector. (preuzeto: 6.09.2021).

[22] SpiderLabs. (2014.). „Jailbreak Detection Methods," adresa: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/jailbreak-detection-methods/. (preuzeto: 6.09.2021).

[23] A. Mustedanagic. (2019.). „Why Jailbreak Detection in iOS Apps Is Pointless," adresa: https://infinum.com/the-capsized-eight/jailbreaking-bad-why-jailbreak-detection-in-ios-apps-is-pointless. (preuzeto: 4.09.2021).

[24] B. Hugo. (2017.). „Objective-C vs Swift messages dispatch," adresa: https://blog.untitledkingdom.com/objective-c-vs-swift-messages-dispatch-9d5b7fd58327r. (preuzeto: 6.09.2021).

[25] B. King. (2016.). „Method Dispatch in Swift," adresa: https://www.rightpoint.com/rplabs/switch-method-dispatch-table. (preuzeto: 6.09.2021).

[26] G. Raskind. (2012.). „dyld: Dynamic Linking On OS X," adresa: https://www.mikeash.com/pyblog/friday-qa-2012-11-09-dyld-dynamic-linking-on-os-x.html. (preuzeto: 6.09.2021).

[27] R. S. Labs. (2021.). „About Us," adresa: https://rhinosecuritylabs.com/company/. (preuzeto: 8.09.2021).

[28] Techopedia. (2021.). „Stack Smashing," adresa: https://www.techopedia.com/definition/16157/stack-smashing. (preuzeto: 8.09.2021).

[29] *Mobile Application Penetration Test*, RhinoSecurityLabs, 2018.

[30] (2021.). „Mobile Wallet," adresa: https://concordium.com/wp-content/uploads/2021/05/Concordium-Mobile-Application-Security-Assessment-Report-Updated.pdf. (preuzeto: 8.09.2021).

[31] Y. Hakon. (2019.). „KZen iOS application PT," adresa: https://zengo.com/wp-content/uploads/2019/06/Kzen_External_Report_AppSecLabs_June2019.pdf. (preuzeto: 8.09.2021).

Popis slika

Popis tablica