VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

iOS operacinės sistemos saugumo galimybių tyrimas ir spragų išnaudojimas

Research of iOS operating system security and the abuse of its exploits

Bakalauro baigiamasis darbas

Atliko:	Mindaugas Jucius	(parašas)
Darbo vadovas:	lekt. dr. Vytautas Valaitis	(parašas)
Recenzentas:	lekt. Jonas Žagūnas	(parašas)

Vilnius 2017

Santrauka

Apple vystoma mobiliųjų įrenginių operacinė sistema iOS, kompanijos pristatoma kaip saugiausias produktas rinkoje. Pastaraisiais metais globaliu mastu plintantis ir valstybiniame lygmenyje vykdomas naudotojų informacijos kaupimas kviečia įsitikinti išmaniuosiuose įrenginiuose veikiančios programinės įrangos patikimumu.

Darbe, remiantis saugumo ekspertų atliktais eksperimentais ir atlikta taikomų saugumo priemonių analize, išryškinamos pažeidžiamos iOS operacinės sistemos vietos, pateikiami siūlymai saugumo gerinimui.

Operacinės sistemos saugumo spragų analizė sutelkta į trečiųjų šalių produktams taikomus veikimo apribojimus ir jų išvengimo būdus. Programėlėms, veikiančioms izoliuotuose konteineriuose, pasiekiamas tik ribotas kiekis sistemos išteklių ir naudotojo talpinamų duomenų. Darbe išskirti du būdai leidžiantys pasiekti privačius duomenis ir priversti iOS įrenginį atlikti potencialiai žalingus veiksmus.

Išmaniųjų įrenginių operacinių sistemų saugumo lyginimui pasitelktas atviro kodo produktas, egzistuojantis abiejose tiriamose platformose. Pateikti kodo pavyzdžiai, rodantys tokį pat funkcionalumą siūlančios aplikacijos realizacijos skirtumus, susijusius su prieiga prie tinklo. Multiplatforminiams produktams suteikiamų privilegijų kiekių skirtumų analizės rezultatais parodyti iOS privatumo valdymo mechanizmo trūkumai.

Raktiniai žodžiai: iOS, Android, mobiliųjų įrenginių operacinių sistemų saugumas, iOS spragos, iOS spragų išnaudojimas

Summary

iOS operating system, which is developed by Apple Inc., is claimed to be the most secure product in the market. In the light of worsening human rights conditions, mainly concerning data privacy breaches performed on governmental level, the need to be reassured about the security of software that runs in our pockets is strengthened.

This thesis is focused on researching on potential iOS security exploits and their abuse to access private information or perform malicious actions without user's consent.

The operating system's security exploit analysis is focused on security features imposed upon third-party products that run in iOS. Apps in iOS are each separately contained in a sandbox, with limited access to system resources and user data. Building upon the work of mobile security analysts, two ways of invoking private functionality and possibly abusing it, are presented in late chapters of the thesis.

The comparison of Android and iOS security features is performed mainly by using an open source product, which is available on both platforms. Code examples are given, representing implementation differences for the same functionality which uses sensitive accesses. Info about other cross-platform products are gathered from the corresponding app marketplaces, disclosing issues within iOS app run-time privileges handling.

Keywords: iOS, Android, mobile devices' operating systems' security, iOS security issues, iOS security exploits

TURINYS

Įvadas

Didėjant išmaniųjų mobiliųjų įrenginių naudojimui (prognozuojama, jog 2020-taisiais metais naudotojų skaičius pasieks 2.87 milijardo [Cha17]), prietaisuose veikiančių operacinių sistemų saugumas tampa ypatingai svarbus. Išmanieji įrenginiai naudojami kaupti ir dalintis privačia informacija - nuotraukomis, buvimo lokacija, vaizdo įrašais. Plečiantis bankinių paslaugų sektoriui, iš išmaniųjų įrenginių pasiekiamos banko sąskaitos, atliekami pirkimai. Tokios informacijos naudojimas ir laikymas įrenginyje, kurio programinė įranga potencialiai pažeidžiama yra reali naudotojų saugumo grėsmė. Tai patvirtina išviešinta informacija apie vyriausybinių organizacijų (pvz. Jungtinėse Amerikos Valstijose - *National Security Agency (NSA)*, Junginėje Karalystėje - *The Government Communications Headquarters (GCHQ)*) vykdomą piliečių sekimą naudojant programinės įrangos spragas [MD13]. Didžiausią rinkos dalį užimančių mobiliųjų operacinių sistemų iOS ir Android [App17c] siūlomi vartotojų privatumo apsaugos mechanizmai išbandomi ne vien pavienių atakuotųjų, bet ir valstybiniais pinigais funduojamų organizacijų.

2016-taisiais metais naujausioje iOS 9 operacinėje sistemoje buvo aptiktos trys saugumo spragos, leidusios nuotoliniu būdu perimti įrenginio kontrolę. Iki spragų išviešinimo ir ištaisymo, Jungtinių Arabų Emiratų vyriausybė, įsigijusi spragų išnaudojimo programinę įrangą iš trečiųjų šalių, jas naudojo sekti ir įkalinti politinius disidentus, nepritarusius šalyje vykusiems pokyčiams [Hel16].

Per 2015-taisiais Jungtinėse Amerikos Valstijose, San Bernardino mieste, įvykusį teroristinį išpuolį, buvo rastas vieno iš nukautų akto sukelėjų iPhone įrenginys. JAV federalinių tyrimų biuras *(angl. FBI)* išdavė teismo orderį, liepiantį Apple kompanijai sukurti programinę įrangą, leidžiančią iššifruoti įrenginyje esančią informaciją [McL16]. Kompanijai atsisakius paklusti, federalinių tyrimu biuras neatskleistam asmeniui sumokėjo 900 tūkst. JAV dolerių už įrenginio atrakinimą [Ing16].

Apple kompanija savo kuriamą mobiliųjų telefonų operacinę sistemą iOS pristato kaip saugiausią ir labiausiai į naudotojo privatumo išsaugojimą koncentruotą produktą rinkoje [App17b]. Apple vystomuose iPhone mobiliuosiuose telefonuose, išimtinai naudojančiuose tik iOS operacinę sistemą, pasitelkiami ne tik programinės, bet ir aparatinės įrangos saugumo mechanizmai. Už pagal nutylėjimą veikiantį įrenginio duomenų šifravimą atsakingas atskiras fizinis komponentas, operacinė sistema leidžia apriboti reklamų tiekėjų vykdomą statistikos kaupimą apie įrenginį. Yra galimybė nuotoliniu būdu ištrinti įrenginio duomenis.

Vis dėlto, globaliai prastėjanti žmogaus teisių laikymosi padėtis, verčia patikrinti Apple teiginį, jog iOS yra saugiausia operacinė sistema rinkoje. Tad darbo tema pasirinkta neatsitiktinai: norima išanalizuoti silpnąsias iOS vietas, pateikti galimų atakų pobūdžius ir palyginti sistemos saugumo būklę su daugiausiai naudotojų turinčiu mobiliųjų operacinių sistemų rinkos produktu, Google kuriama Android sistema.

Darbas skirstomas į dvi dalis. Pirmoji - technologinių sprendimų, paveikusių Apple kurtų operacinių sistemų vystymąsi, iOS atsiradimą ir sistemoje naudojamas saugumo technologijas, įvertinimas. Ši analizė atlikta pirmuose trijuose skyriuose. Pirmasis skyrius skirtas iOS pirmtakės Mac OS personalinių kompiuterių operacinės sistemos vystymosi istorijos tyrimui. Tai svarbu tuo,

jog technologijos, kurios pradėtos naudoti devintajame XX a. dešimtmetyje, vis dar tarnauja kaip pagrindas šiuolaikinėms Apple operacinėms sistemoms, tarp jų ir iOS. Antrajame skyriuje išdėstyti pagrindiniai sluoksniuotos architektūros *Mach* branduolio komponentai. Trečiajame - iOS sistemos apsaugos mechanizmų analizė, didžiausią dėmesį skiriant toms programinės įrangos dalims, kuriose dažniausiai randamos saugumo spragos.

Antroji dalis sudaryta iš dviejų skyrių. Ketvirtajame skyriuje suformuluojami iOS sistemos kompromitavimo būdai, remiantis pirmojo skyriaus saugumo priemonių analize ir saugumo ekspertų eksperimentais. Paskutiniame skyriuje lyginama iOS ir Android operacinių sistemų saugumo būklė. Lyginimas atliekamas atsižvelgiant į sistemose taikomas, trečiųjų šalių aplikacijoms suteikiamų veikimo privilegijų, politikas. Pasitelkiant atviro kodo aplikaciją *WordPress*, veikiančią abiejose platformose, kodo pavyzdžiais paremiami lyginimo rezultatai.

Suformuluotų iOS atakos vektorių informacija naudojama išskirti patobulinimus sistemos saugumui, kuriuos įgyvendinus, potencialiai sumažinama sistemos išnaudojimo rizika. Išskirti patobulinimai paremiami atliktu iOS sistemos saugumo būklės lyginimu su Android operacine sistema.

Darbo tikslas:

Pateikti rekomendacijas iOS operacinės sistemos patobulinimams, tiriant sistemoje realizuotas saugumo priemones ir lyginant su įrankiais naudojamais Android sistemoje.

Darbo uždaviniai:

1. Ištirti saugumo priemones naudojamas naujausioje iOS versijoje, pradedant nuo istorinių aplinkybių, lėmusių iOS susiformavimą ir technologijas, naudotas sistemos kūrimui.
2. Pateikti galimus atakos vektorius, kompromituojančius iOS sistemos saugumo priemones
3. Palyginti iOS ir Android operacinėse sistemose naudojamas saugumo priemones
4. Suformuluoti rekomendacijas iOS sistemos patobulinimams atsižvelgiant į atakų vektorių analizės ir sistemų palyginimo rezultatus

1. Apple operacinių sistemų istorija

Skyriuje tiriama Apple kompanijos kurtų operacinių sistemų vystymosi istorija, apžvelgiami pagrindiniai technologiniai komponentai. Šios darbo dalies svarba pasižymi tuo, jog nuo 9-ojo XX a. dešimtmečio vystomos *Mac OS* kūrimo metu naudotos technologijos pasitarnavo kaip pagrindas išmaniųjų įrenginių operacinei sistemai iOS.

Apple kompanija įkurta 1976-taisiais Steve Jobs, Steve Wozniak ir Ronald Wayne kaip personalinių kompiuterių kūrimo ir pardavimo kompanija [Bra01]. Pirmieji kompiuteriai su šiandienykštės operacinės sistemos *macOS* pirmtake pasirodė 1984-taisiais metais [Her05]. Tai buvo Macintosh serijos pradininkas - kompiuteris su *System Software* operacine sistema. Ši programinė įranga vystyta iki 1999-tųjų metų [Sei10] (1996-taisiais pervadinta į *Mac OS*), charakterizuojama kaip monolitinio tipo sistema. Nuo pirmosios versijos iki *System 4* revizijos, sistema neturėjo daugiaprogramio režimo. Sistema taip pat kritikuota dėl menkų įrankių atminties valdymui, prieigos teisių kontrolės ir atminties apsaugos nebuvimo, dažnų branduolio kritinių klaidų. Šių trūkumų pagrindinė priežastis - atgyvenusios technologijos, kurių pagrindu vystyta *Mac OS* [Her10].

1.1. Copland projektas

1994-taisiais metais Apple pradėjo vystyti projektą pavadinimu *Copland*. Dešimtį metų skaičiuojančią *Mac OS* turėjo pakeisti sistema su realizuota atminties apsauga, prevenciniu *angl. (preemptive)* daugiaprogramiu rėžimu, nauja vartotojo sąsaja, bet išlaikant suderinamumą su esama *Mac OS* programine įranga [Fra96].

Copland sistemos architektūra paremta naujai sukurtu mikrokerneliu, pavadintu *NuKernel*, kurio pagrindinės užduotys apėmė sistemos darbo pradžią ir atminties valdymą, kitus procesus paliekant specialioms programoms - *serveriams*. Nuo kernelio atskirtų procesų pavyzdžiai - failų sistema ir tinklo paketų siuntimas [Fra96].

Pagrindinė problema, su kuria susidūrė Apple inžinieriai - atminties trūkumas kompiuteriuose. Programinės įrangos suderinamumui išlaikyti vienas iš *Copland* sistemos modulių yra veikianti *System 7.5* sistema. Toks architektūrinis sprendimas reiškė dvi operacines sistemas atmintyje, tačiau tai leido naujojoje operacinėje sistemoje naudotis neatnaujintomis programomis [Die95].

Apie naująją operacinę sistemą pirmąkart viešai paskelbta 1995-tųjų kovą [Fra96]. Žadėtas išleidimas 1996-taisiais [Cra95] neįvyko dėl vidinės sumaišties Apple procese. Inžinieriai veržėsi dirbti prie naujosios sistemos, komandų vadovai to neleisdavo, teigdami, jog dabar vykdomų projektų įgyvendinimas svarbus *Copland* užbaigimui. Tokia problema spręsta perkeliant projektus *Copland* sistemą kaip naują funkcionalumą [Wid08]. Augant funkcijų kiekiui, testavimas darėsi vis labiau sudėtingas, Apple inžinieriai jau 1995-tųjų pradžioje teigė, jog 1996-tieji pernelyg optimistinė prognozė išleidimui, tikėtis išleidimo galima tik 1997-taisiais [Wid08].

1.2. NeXT įsigijimas

Naujasis Apple vadovas Gil Amelio pasamdė Ellen Hancock iš *National Semiconductor* puslaidininkių gamintojos techninės direktorės *(angl. Chief Technical Officer)* pozicijai [Bra01]. E.

Hancock priėmė sprendimą atšaukti *Copland* ir svarstė kaip naująją OS naudoti trečiųjų šalių produktą [Hor12].

1996-tųjų rugpjūtį oficialiai paskelbta apie *Copland* atšaukimą. Rinkos analitikai tikėjosi, jog sistemą naujuose Apple kompiuteriuose pakeis Sun korporacijos produktas *Solaris*, arba *Windows* [Hor12] [Wal05].

Vis dėlto dabar naudojamoms *macOS* ir iOS sistemoms pradžią davė *NeXT Computer Inc* produktai. *NeXT* kompaniją įkūrė Steve Jobs, po to, kai jį iš vadovo pozicijos 1985-taisiais atleido Apple akcininkų susirinkimas [Wal05]. *NeXT* buvo įsteigta įgyvendinti inovatoriškus produktus akademinėms ir verslo reikmėms. Vienas iš *NeXT* produktų yra *Mach* kernelio pagrindu [HC11] sukurta operacinė sistema *NeXTSTEP*, kuri pasitarnavo kaip pagrindas naujajai Apple operacinei sistemai *OS X* išleistai 2001-taisiais [Wal05].

NeXTSTEP kompanijos sukurtos funkcijos, pvz.: *Interface Builder*, skirtas sąsajų kūrimui, *Driver Kit(I/O kit)* tvarkyklių naudojimui, tarp-procesinio bendravimo įrankiai, su fundamentaliais patobulinimais, *macOS* sistemoje egzistuoja iki šiol [HC11]. Kai kurios *NeXTSTEP* bibliotekos parašytos *Objective-C* kalba, iki šiol naudojamos *macOS* ir iOS programų kūrimui, karkaso *Foundation* klasės turi prefiksą *NS*, reiškiantį *NeXTSTEP*[HC11]. *Objective-C* kalba yra naudojama platformai skirtų aplikacijų kūrimui.

1.3. iOS atsiradimas

2007-taisiais išleistame pirmajame *iPhone* telefone veikė *iPhone OS* (vėliau pervadinta į iOS, nes tapo naudojama ne tik telefonuose - planšetiniuose kompiuteriuose, išmaniuosiuose laikrodžiuose ir t.t.) sistema, kurios branduolys yra išvestas iš *OS X (macOS)* [HC11]. Abiejų sistemų pagrindas, kurį sudaro kernelis ir primityvi operacinė sistema, yra *Darwin*. *Darwin* yra Apple sukurta, atviro kodo sistema. *Darwin* turi įprastinius *Unix* šeimos sistemų įrankius, tačiau neturi vartotojo sąsajos, tad nepritaikyta įprastiniam naudojimui. *Darwin* distribucija veikia dėka *XNU* kernelio, kuris paremtas *Mach* kerneliu ir *FreeBDS* operacine sistema.

1.4. iOS ypatybės

iOS yra operacinė sistema, pritaikyta įrenginiams su liečiamaisiais ekranais. Vartotojo sąsaja sukurta dėka *Cocoa Touch* karkaso, kurio pirmtakas yra *macOS* karkasas *Cocoa*.

Pirmojoje iOS versijoje trečiųjų šalių produktai buvo veikė tik kaip web aplikacijos. Programėlių kūrimo paketas *(SDK)* išleistas 2008-taisiais [HC11], atvėrė galimybę ne Apple produktų veikimui iPhone įrenginiuose. Tačiau visos programėlės prieš jų viešo platinimo pradžia *AppStore* platformoje, turi būti patvirtintos Apple. Įkeliami produktai tikrinami nuo pavojingo kodo, privačių programavimo interfeisų *(angl. Application Programming Interface)* naudojimo, užšifruojami, tam, kad užtikrinti, jog įrenginyje būtų užtikrintas nemodifikuotos programos veikimas[HC11].

Toks įkėlimo procesas vis dar sulaukia kritikos dėl per ilgo tikrinimo periodo (2016-aisiais tikrinimo laikotarpis sumažėjo nuo savaitės iki 2-jų darbo dienų), tačiau tai leido kompanijai apsaugoti platformą nuo kenkėjiškų, standartų neatitinkančių, programų platinimo [HC11].

iOS sistemoje *root* privilegijos nėra įprastai prieinamos. Prieiga gaunama tik išnaudojus saugumo spragas sistemoje, toks procesas vadinamas *jailbreak* [HC11].

2. Darwin ir XNU kernelis

Darwin yra, atviro kodo, Apple sukurta, sistema, kuri pasitarnavo kaip pagrindas kitoms sukurtoms operacinėms sistemoms. *Darwin* pagrindas - *XNU* kernelis. Kadangi *XNU* kodas yra viešai prieinamas, šio kernelio kodo analizavimas yra pagrindinis būdas trečiųjų šalių saugumo ekspertams ieškoti klaidų kodo logikoje, vedančių į privilegijų eskalaciją [Ess11].

XNU hibridinis kernelis skirstomas į tris pagrindines dalis:

- *Mach* kernelis - žemiausio lygio sluoksnis *XNU* architektūroje. Apple naudoja *Mach* 3.0 versiją, sukurtą Carnegie Mellon universitete. *Mach* kurtas kaip mikrokernelis, suteikiantis procesoriaus valdymo, užduočių skirstymo *(angl. task scheduling)* ir tarp-procesinio bendravimo įrankius *angl. IPC* [HC11]. Dėl „sluoksniuotos“ architektūros skirtumai tarp iOS ir *macOS* šiame abstrakcijos lygmenyje yra nežymūs [Ess11]. *XNU* kernelyje failų sistema ir ryšio moduliai dalinasi atminties vieta *(angl. memory spaces)* su *Mach* kerneliu. Apple teigia, jog tai leidžia išvengti tarp-procesinių būsenos atstatymų *angl. (context switching)*, kas teigiamai veikia sistemos našumą [HC11].
- *BSD* sluoksnis - užtikrina *POSIX* suderinamumą. *BSD* posistemė atsakinga už *Unix* procesų modelį, saugumo strategijas (failų prieinamumo apribojimą vartotojams), tinklo protokolus, failų sistemas naudojamas *macOS* - *HFS/HFS+*, kriptografijos karkasą, daugiagijiškumo apsaugas *(angl. locking primitives)* [Wat08]. *XNU* sistemoje naudojamas *BSD* kodas yra paimtas iš *FreeBSD* projekto kernelio [Wat08]. *FreeBSD* kodas Apple inžinierių yra modifikuojamas *macOS* pritaikymui. Į kernelį pridėtomis funkcijomis Apple dalinasi su atviro kodo bendruomene, pvz. *Grand Central Dispatch* daugiagijiškumo karkasas *FreeBSD* kernelyje realizuotas 2009-taisiais [Wat09].
- *I/O Kit* karkasas - parašytas modifikuota (nėra išimčių *(angl. exceptions)*, daugybinio paveldėjimo *(angl. multiple inheritance)*, ir kt.) C++ kalba. Skirtas įrenginių tvarkyklių kūrimui ir veikimo pritaikymui *XNU* kerneliui [Sin03]. Objektinio tipo karkasas pateikia įrenginių klases, kurių elgsena yra perpanaudojama naudojant paveldėjimą. *I/O Kit* palaiko daugiagijiškumą, automatinį įrenginių konfigūravimą *(angl. plug-and-play)* [Sin03]. *I/O Kit* leidžia rašyti tvarkykles veikiančias vartotojo procese *(angl. user-space)* [Sin03]. Aukštą privilegijų lygį turinti tvarkyklė *(angl. kernel-space)* (tinklo adapteriams, vaizdo plokštėms, USB įrenginiams, OS virtualizacijos programoms), nesėkmės atveju neatstatomai sutrikdo sistemos veikimą (sukelia kernelio *paniką*), tad veikimas žemu lygmeniu prisideda prie *XNU* kernelio stabilumo. Kiti privalumai rašant tokio pobūdžio tvarkykles - prieinamos derinimo programos *(angl. debugger)*, kadangi tik vartotojo procesų atmintis yra sukeičiama *angl. (swappable)* [CRK05] (retai prieinama informacija perkeliama iš RAM atminties į standųjį diską, atlaisvinant vietą naujai informacijai), didelius kompiuterio atminties kiekius naudojanti tvarkyklė, įrenginio nenaudojimo atveju, gali būti perkelta, taip atlaisvinant RAM atmintį.

3. Saugumo priemonės iOS sistemoje

Skyriuje atliekama programinės įrangos saugumo priemonių analizė. Analizė atliekama dėmesį sutelkiant į pagrindinius iOS saugumo komponentus, kuriuose dažniausiai išnaudojamos saugumo spragos. Vėlesnėse darbo darbo dalyse šio skyriaus rezultatai pasitelkiami kaip pagalbinė informacija sistemos atakų vektorių formulavimui.

iOS operacinės sistemos saugumas realizuotas taip, jog apimtų visus pagrindinius programinės ir fizinės įrangos komponentus. Tai reiškia, jog ir funkcijos, su kuriomis sąveikauja iOS vartotojai yra „apvilktos“ atitinkamomis saugumo priemonėmis, pvz. aplikacijos ir duomenys yra šifruojami. iOS sistemos architektūra užtikrina, jog gili saugumo priemonių integracija nesukeltų naudojimosi vėlinimo. Operacinės sistemos saugumo modulių analizė pradedama nuo žemiausio lygio modulių - saugios pasileidimo procesų grandinės *(angl. secure boot chain)*, sisteminės programinės įrangos autorizacijos *(angl. System Software Authorization)* [App17b].

3.1. Saugi pasileidimo procesų grandinė

Šis modulis apima sistemos pasileidimo *(angl. boot-up)* procesą. Kiekvienas sistemos darbo pradžioje atliekamas žingsnis naudoja kriptografiškai pasirašytus komponentus, taip užtikrinant vykdomo kodo integralumą. Sistemos pasileidimo komponentai yra šie: branduolys, branduolio plėtiniai, ryšio programinė aparatinė įranga *(angl. baseband firmware)* [App17b]. Šio saugumo proceso tikslas apsaugoti žemiausio lygio programinę įrangą nuo neleistino modifikavimo.

Įjungus iOS operacinę sistemą naudojantį įrenginį, pagrindinė plokštė vykdo kodą iš *read-only* atminties vadinamos *Boot ROM*. Šis nekintamas kodas yra įrašomas gaminant įrenginio plokštę, tad jo kilme neišreikštinai pasitikima *(angl. implicitly trusted)* [App17b]. *Boot ROM* kodas šifruojamas Apple Root CA viešuoju raktu, tam kad būtų įsitikinta *iBoot* paleidyklės (sekantis žingsnis paleidimo proesų grandinėje) *(angl. bootloader)* integralumu. Paleidyklei užbaigus krovimo darbus, paleidžiamas veikti iOS branduolys.

Jei bent vienas proceso žingsnių nesugeba įvykdyti savo užduoties arba patvirtinti savo leidžiamo kodo integralumo, įrenginio įjungimas yra sustabdomas. Įrenginio ekrane matomas pranešimas prisijungti prie *iTunes* programos, kuri atlieka programinės įrangos atnaujinimą ar taisymą.

Įrenginiuose, kurie turi prieigą prie mobiliojo ryšio, tinklo posistemė *(angl. baseband subsystem)* taip pat vykdo panašų saugaus pasileidimo procesą verifikuojantį naudojamus komponentus. Atskiras procesas vykdomas ir įrenginiuose su *Secure Enclave* koprocesoriais.

3.2. Sisteminės programinės įrangos autorizacija

Reguliariai išleidžiami programinės įrangos atnaujinimai ištaiso plintančias saugumo spragas. Atnaujinimai yra suteikiami visiems palaikomiems įrenginiams vienu metu. Programinės įrangos atnaujinimai gali būti įrašomi per *iTunes* aplikaciją arba parsisiunčiami pačiame atnaujinamame įrenginyje.

Aprašytasis pasileidimo procesas padeda užtikrinti, jog iOS įrenginiuose veikia tik Apple pasirašytas kodas. Tam, kad užkirsti kelią ankstesnės, potencialiai nesaugios, iOS operacinės sistemos

versijos įsirašymui įrenginyje, pasitelkiami žingsniai apibendrintai vadinami *System Software Authorization* [App17b]. Jei būtų įmanoma grįžti į senesnę operacinės sistemos versija, tokia praktika suteiktų lengvą būdą žinomų saugumo spragų išnaudojimui.

iOS operacinės sistemos atnaujinimai instaliuojami per iTunes arba belaidžiu būdu *(angl. over the air)* įrenginyje. Vykstant atnaujinimui įrenginys prisijungia prie Apple instaliacijos autorizacijos serverio ir kiekvienam sistemos komponentui (paleidyklei, branduoliui, operacinei sistemai) siunčia užšifruotą atnaujinimui reikalingą informaciją, atsitiktinai sugeneruotą reikšmę apsaugančia nuo užklausų siuntimo pakartojimo *(angl. anti-replay value (nonce))* ir unikalų įrenginio plokštės identifikatorių *(angl. Exclusive Chip ID (ECID))* [App17b].

Autorizacijos serveris tikrina siunčiamą įrenginio informaciją su iOS versijomis į kurias galimas atnaujinimas. Jei randamas atitikimas, įrenginio ECID identifikatorius prie atnaujinimo informacijos ir užšifruojamas. Įrenginys iš serverio gauna „personalizuotą“ atnaujinimą.

Įrašius atnaujinimą, įrenginiui kraunant naująją operacinės sistemos versiją pasileidimo procesų grandinė validuoja jog kodo parašas yra suteiktas Apple tikrindamas ar įrenginio ECID identifikatorius sutampa su esančiu atsiųstame atnaujinime.

Šie žingsniai užtikrina, jog atnaujinimas skirtas konkrečiam įrenginiui ir užkerta kelią iOS versijų kopijavimui iš vieno įrenginio į kitą. Kriptografinė *nonce* reikšmė užkerta kelią sistemos atakuotojams modifikuoti serverio suteiktus atnaujinimus.

3.3. Aplikacijų saugumas

iOS operacinė sistemos saugumo architektūra užtikrina, jog naudotojų parsisiunčiamos aplikacijos yra užšifruotos, nemodifikuotos ir jos veikia savo konteinerio *(angl. sandbox)* ribose.

3.3.1. Programėlių kodo pasirašymas

Nuo veikimo pradžios iOS branduolys kontroliuoja kokiems naudotojo procesams ir aplikacijoms leidžiama veikti. Tam, kad užtikrinti, jog parsisiųsti produktai yra sukurti patikimų šaltinių, iOS sistemoje yra privaloma, jog vykdomas kodas būtų pasirašytas Apple išduodamu sertifikatu. Aplikacijos, kurios įrašomos kartu su operacine sistema (pvz. interneto naršyklė Safari arba el. pašto programėlė Mail), yra pasirašytos pačios Apple [App17b]. Trečiųjų šalių sukurti produktai pasirašomi kūrėjams Apple suteikiamu sertifikatu. Privalomas kodo pasirašymo mechanizmas išplečia vykdomo pasitikėjimo zoną nuo operacinės sistemos iki programėlių, užkirsdamas galimybę trečiųjų šalių produktams vykdyti neverifikuotą kodą ir resursus.

iOS programėlių kūrėjams, kurie nori platinti savo kurtus produktus, reikalinga turėti paskyrą *Apple Developer Program* portale. Prieš suteikiant kūrimo sertifikatą yra patvirtinama fizinio ar juridinio asmens tapatybė. Tai reiškia, jog kiekviena aplikacija App Store platformoje yra sukurta identifikuojamo asmens. Kenksmingų produktų plitimo atveju tai padeda atrasti atsakingas organizacijas ar asmenis. App Store programėlių parduotuvėje esantys produktai prieš viešo platinimo pradžią yra patikrinti tinkamumo proceso *(angl. vetting process)*. Taip užtikrinama platinamų programų kokybė ir atitikimas aprašymui.

iOS programėlių kūrimo metu leidžiama į projektą įterpti *(angl. embed)* įvairų funkcionalumą suteikiančius trečiųjų šalių karkasus [HC11]. Tam, kad apsaugoti iOS ir kitas veikiančias aplikacijas nuo kodo įkrovimo į atmintį veikimo metu, sistema, visoms į aplikaciją įterptoms dinaminėms bibliotekoms (karkasams) atlieka kodo parašų validaciją [App17b]. Ši verifikacija įgalinama naudojant komandos identifikatorių, kuris gaunamas iš Apple suteikto sertifikato. Komandos identifikatorius yra 10-ties simbolių ilgumo skaičių ir raidžių eilutė, pvz. *1A2B3C4D5F*. Trečiosios šalies sukurta programa gali naudoti bet kurią viešai iOS platformoje prieinamą biblioteką arba biblioteką su tokiu pačiu komandos identifikatoriumi [HC11]. Kadangi sisteminiai vykdomieji failai neturi komandos identifikatorių, juose naudojamos bibliotekos, kurios yra operacinės sistemos dalis.

iOS kodo pasirašymo mechanizmas apsaugo sistemą nuo potencialiai kenksmingų neverifikuotų programėlių instaliavimo ir kodo vykdymo. Vykdomųjų atminties sektorių *(angl. executable memory pages)* įkrovimo metu yra vykdomi kodo parašų patikrinimai, užtikrinantys, jog programėlė nebuvo modifikuota nuo jos instaliavimo ar atnaujinimo laiko.

3.3.2. Veikiančiųjų procesų apsauga

Sistemai įsitikinus, jog programėlė yra iš patikimo šaltinio ir jos veikimą galima pradėti, pradedamos taikyti saugumo priemonės, skirtos iOS ar kitų aplikacijų kompromitavimo išvengimui. iOS operacinėje sistemoje kiekviena trečiosios šalies sukurta aplikacija veikia atskiruose konteineriuose *(angl. sandbox)*, apribojančiuose prieigą prie kitų aplikacijų sukurtų failų ir neleidžiančių atlikti pakeitimų sistemoje [App17b]. Kiekviena aplikacija turi unikalią, savo failams, skirtą direktoriją, kuri, atsitiktiniu būdu, yra priskiriama instaliacijos metu. Jei aplikacija nori pasiekti duomenis esančius „už“ jos konteinerio, tam pasitelkiami iOS suteikti servisai.

Naudotojo įrašytoms programėlėms taip pat nėra prieigos prie sisteminių failų ir resursų. Didžioji dalis iOS procesų, taip pat ir trečiųjų šalių aplikacijos, veikia kaip neprivilegijuotas *mobile* naudotojas [App17b; HC11]. Operacinės sistemos particija failų sistemoje yra skaitymo režime *(angl. read-only)*. iOS sistemos aplikacijų programavimo sąsajos *(angl. application programming interfaces)* veikiančioms aplikacijoms nesuteikia galimybės eskaluoti savo privilegijų ar modifikuoti sistemą.

Trečiųjų šalių aplikacijų prieiga prie naudotojo informacijos apibrėžiama naudojant projekte paskelbtomis teisėmis *(angl. entitlements)*. Teisė šiame kontekste yra rakto - reikšmės pora *(angl. key-value pairs)* aplikacijos *Info.plist* faile, kuris yra pasirašomas kartu su aplikacijos kodu, tad negali būti keičiamas po programėlės išleidimo. Sisteminiai procesai naudoja teisių paskelbimą prieigai prie resursų, kurie reikalautų veikimo *root* teisėmis. Tai padeda išvengti potencialių privilegijų eskalacijų, veikiant kompromituotai sisteminei aplikacijai ar foniniam procesui *(angl. daemon process)*. Atlikti darbą fone aplikacijoms leidžiama tik per specialiai tam skirtas programavimo sąsajas [App17b], tai padeda išlaikyti pastovų sistemos našumą ir baterijos energijos eikvojimo lygį.

Atminties adresu erdvės išdėstymo randomizacija *(angl. address space layout randomization (ASRL))* apsaugo nuo atminties iškraipymo klaidų išnaudojimo. Sisteminės aplikacijos naudoja *ASRL* atsitiktinių atminties regionų priskyrimui. Atsitiktinis atminties adresų priskyrimas vykdo-

mam kodui ir sisteminėms bibliotekoms, sumažina atakų prieš iOS grėsmę. Pavyzdžiui *return-to-libc* tipo ataka manipuliuoja atminties adresais steke, priversdama sistemą įvykdyti kenksmingą kodą. Įvykdyti ataką tampa ypatingai sunku nežinant galimų paleisti *(angl. executable)* atminties segmentų adresų, tai užtikrinama adresus randomizuojant. Xcode integruota programų kūrimo aplinka kuriamus produktus automatiškai kompiliuoja su *ASRL* palaikymu [App17b].

Papildoma apsauga iOS sistemoje suteikiama ARM architektūroje esančia *Execute Never (XN)* funkcija, kuri pažymi atminties segmentus kaip nevykdomus *(angl. non-executable)*. Atminties vietos, į kurias galima rašyti kodą ir jį įvykdyti naudojamos tik ypatingomis sąlygomis. Norint pasiekti tokią atminties vietą kernelis tikrina ar egzistuoja tik Apple prieinama dinaminio kodo paleidimo teisė. Net ir aptikus šią teisę gali būti įvykdytas tik vienas *mmap* funkcijos iškvietimas, grąžinantis galimą modifikuoti atminties segmentą, kuriam ASRL suteikia atsitiktinį adresą. iOS naršyklė Safari šį funkcionalumą naudoja JavaScript realaus laiko *(angl. Just In Time)* kompiliatoriui [App17b].

3.3.3. Aplikacijų plėtiniai

iOS trečiųjų šalių aplikacijoms suteikia galimybę dalintis funkcionalumu per plėtinius [App17b]. Plėtiniai yra specialaus tipo pasirašyti vykdomieji failai *(angl. executable binaries)*, kurie pridedami į juos suteikiančią aplikaciją. Sistema plėtinius aptinka instaliavimo metu ir juos suteikia kitoms aplikacijoms.

Sistemos vieta, kuri palaiko plėtinius vadinama plėtimo tašku *extension point*. Kiekvienas taškas suteikia skirtingas programavimo sąsajas ir pritaiko tai informacijai atitinkamus apribojimus. Operacinė sistema automatiškai inicijuoja plėtinių veikimo pradžią ir gyvavimo laikotarpį. Teisės *(entitlements)* naudojamos apriboti plėtinio pritaikymą sisteminiams komponentams ir trečiųjų šalių programoms, pvz. *Today* valdiklis *(angl. widget)* matomas tik pranešimų centro *(angl. Notification center)* modulyje, dalinimosi plėtinys matomas tik aplikacijai iškvietus atitinkamą sisteminį dialogą. iOS sistemoje plėtimo taškai yra šie - valdikliai, dalinimosi dialogas, nuotraukų modifikavimas, manipuliavimas dokumentais ir klaviatūros.

Plėtiniai veikia atskirose nuo aplikacijų adresų erdvėse. Komunikacijai tarp plėtinio ir aplikacijos, kuri jį aktyvavo, naudojami tarp-procesiniai informacijos dalinimosi mechanizmai, kuriuos valdo sisteminis karkasas. Plėtiniai veikia izoliuotai nuo kitų plėtinių, tėvinių aplikacijų ir nuo aplikacijų kurios tuos plėtinius naudoja. Aplikacija negali modifikuoti plėtinio adreso erdvės, tai taip pat taikytina ir plėtiniui. Plėtiniai veikia konteineriuose kaip ir trečiųjų šalių programėlės, kurie yra atskirti nuo tėvinio produkto. Tai reiškia, jog jei programėlei, kuri turi plėtinių, suteikta prieiga prie vartotojo kontaktų knygelės informacijos, ši privilegija bus pasiekiama ir jos plėtiniams. Jei aplikacija turinti prieigą prie kontaktų iškviečia kitos programos plėtinį, iškviestasis plėtinys prieigos prie kontaktų neturi.

3.3.4. Aplikacijų grupės

Aplikacijos ir plėtiniai, platinami vieno kūrėjo gali dalintis informacija įrenginyje, jei jiems yra sukurta aplikacijų grupė *(angl. App Group)*. Aplikacijos grupės sukūrimas yra produktų vys-

tytojo atsakomybė, atliekama *Apple Developer* portale, į ją įdedant norimas aplikacijas ir plėtinius. Vienoje aplikacijų grupėje esančioms programoms yra prieinama:

- Bendra direktorija aplikacijų duomenims, kuri įrenginyje išlieka tol, kol yra instaliuota bent viena aplikacija iš grupės.
- Bendri sisteminiai nustatymai (lokacijos prieiga ir panašiai)
- Bendra *Keychain* informacija (slaptažodžiai, aplikacijos sukurta šifruota informacija)

Yra garantuojama, jog *Apple Developer* portale sukurtos aplikacijų grupės identifikatorius yra unikalus visoje iOS ekosistemoje [App17b].

4. Galimi iOS platformos saugumo kompromitavimo būdai

Trečiųjų šalių sukurtos programos iOS sistemoje turi „lygias teises“, instaliavimo metu suteiktų veikimo teisių *(angl. run-time privileges)* atžvilgiu. Aplikacijos iOS sistemoje veikia kaip *mobile* vartotojas, kuris skirtingai nei aukščiausią prieigą valdantis *root* naudotojas turi apribotą prieigą prie sistemos duomenų ir kitų resursų. Tačiau net ir veikdamos tokiomis sąlygomis trečiųjų šalių aplikacijoms yra leidžiama prieiga prie tam tikrų įrenginio naudotojo duomenų. Potencialiai, tai galima išnaudoti duomenų nutekinimui [EKK+11]. Kadangi Apple nesuteikia trečiųjų šalių programuotojams prieinamų duomenų ir privilegijų sąrašo, patys aplikacijų kūrėjai ir saugumo ekspertai analizuoja iOS nubrėžtas ribas [EKJ+13; EKK+11; T W13]. Tam tikslui kuriamos koncepcijos išbandymo *(angl. proof-of-concept)* aplikacijos, kurių kodas įvykdo prieigą prie privačios informacijos. Tokios aplikacijos pateikiamos iTunes AppStore peržiūros procesui, kaip įprastinės programos [EKJ+13; T W13]. Šie eksperimentai padeda suprasti kriterijus, kuriais Apple priima arba atmeta aplikacijas iš savo programėlių parduotuvės. Jei aplikacija priimama, daroma išvada, jog atitinkamą informaciją galima naudoti trečiosioms šalims.

Iki šiol įvykdyti tiriamieji darbai leidžia teigti [EKJ+13; EKK+11; T W13], jog prieš iOS sistemą paprasčiausiai įvykdomas atakos tipas yra privačių aplikacijų programavimo sąsajų *(angl. application programming interface)* naudojimas trečiųjų šalių kurtose programose. Privačios aplikacijų programavimo sąsajos iOS egzistuoja ir privačiuose, ir viešuose karkasuose. Neviešinamų metodų naudojimas trečiųjų šalių aplikacijose suteikia papildomas privilegijas, leidžiančias naudotis tik sisteminiam kodui prieinamu funkcionalumu, pvz. trumpųjų SMS žinučių siuntimu. Apple uždraudė tokią praktiką, todėl programėlės tinkamumo tikrinimo *(angl. vetting)* metu, aptikus privataus metodo kvietimą, pateiktas produktas yra atmetamas.

Suradus būdą, kuriuo naudojantis pavyktų išvengti privačių metodų kvietimo aptikimo tinkamumo tikrinimo metu, tai leistų rašyti ir vykdyti kodą, kuris turėtų priėjimą prie įprastai nepasiekiamo funkcionalumo ir jį išnaudoti potencialiai kenksmingiems tikslams. Aptikimo išvengimas reiškia, jog tokie produktai atsidurtų oficialioje Apple programėlių parduotuvėje AppStore, kuri yra pasiekiama visiems iOS naudotojams. Viena naujausių viešai nuskambėjusių plačiu mastu įvykdytų atakų prieš iOS vartotojų privatumą yra Uber kompanijos aplikacijoje naudotas privatus iOS metodas, leidžiantis pasiekti unikalų įrenginio identifikavimo kodą [Isa17]. Šio metodo naudojimas leido kompanijai Uber sužinoti ar aplikacija prieš tai buvo instaliuota įrenginyje, kiek kartų buvo ištrinta ir t.t. Tokios informacijos kaupimas nusižengia Apple privatumo gairėms.

Šiame poskyryje pateikti du būdai kaip dinamiškai kreipiantis į privačius metodus, įmanoma išvengti aptikimo tikrinimo metu. Dinaminis kodo vykdymo mechanizmas padengia pirmąjį iš dviejų atakos vektoriaus žingsnių. Antrasis žingsnis yra aptikti naudingus privačius metodus, nereikalaujančius privilegijų eskalacijos, t.y. kurie yra leidžiami įvykdyti procesams veikiantiems *mobile* naudotojo terpėje (kai kurie Apple aplikacijų procesai dėl saugumo taip pat vykdomi *mobile* privilegijomis, pvz. *MobileSafari* naršyklė). Tokių metodų išnaudojimas leido tyrėjams realizuoti žalingas atakas prieš iOS sistemą [EKJ+13; T W13], įrenginiuose kurie neturi *root* privilegijų (neįvykdytas ankščiau minėtas *jailbreak* procesas).

Privataus metodo išnaudojimo proceso realizavimas susideda iš dviejų žingsnių, kurie yra

susiję vienas su kitu, tuo atžvilgiu, jog norimas naudoti privačios programavimo sąsajos metodas apsprendžia dinaminio kvietimo būdą. Toliau pateikti du skirtingus dinaminio krovimo metodus naudojančios kenksmingo kodo vykdymo atakos: trumpųjų SMS žinučių siuntimo ir jėgos metodo *(angl. brute-force)* naudojimas PIN kodo sužinojimui. Naudojantis šiais pavyzdžiais galima realizuoti kitas panašaus pobūdžio atakas.

4.1. iOS sistemos atakos naudojantis dinamiškai įkeltais karkasais

Kuriant iOS aplikaciją kuri išnaudoja privačius metodus, įprastiniai žingsniai, kurių imamasi, jei produkto nebus norima įkelti į AppStore platformą, yra tokie:

1. Naudojant statinės analizės įrankius pasiekti dabartines iOS programinės įrangos kūrimo paketo *(angl. SDK)* antraštinius *(angl. header)* failus
2. Sukurti statinę nuorodą į atitinkamą karkasą Xcode integruotoje programavimo aplinkoje [App16]
3. Importuoti gautuosius karkaso antraštinius failus su metodu antraštėmis aplikacijos kodo failuose, kuriuose norima pasiekti metodus.

SMS žinučių siuntimo funkcionalumas yra apribotas tik sisteminėms programėlėms, trečiųjų šalių kūrėjai oficialiai neturi prieigos prie tam tikslui naudojamų karkasų [EKJ+13]. iOS platformos įrenginiuose trumpųjų SMS žinučių siuntimą įgyvendina *CoreTelephony.framework* karkasas.

Karkase esantis *CTMessageCenter.h* antraštinis failas apibrėžia metodus, skirtus žinučių siuntimui. Importavus šį antraštinį failą atitinkamame aplikacijos kodo faile, SMS žinučių siuntimo privatus metodas iškviečiamas vykdant šį kodą:

```
1 CTMessageCenter *sharedMessageCenter = [CTMessageCenter sharedMessageCenter];
2 [sharedMessageCenter sendSMSWithText:@"SMS testas" serviceCenter:nil
     toAddress:@"+123456789"];
```

1 pav. SMS žinučių siuntimo kodas naudojant privačius metodus

Programai vykdant kodą nurodytą 1-jame paveiksle, *CTMessageCenter* klasėje apibrėžtas klasinis metodas *(angl. class method) sharedMessageCenter* (Objective-C neturi statinių metodų) grąžina *singleton* šabloną įgyvendinantį objektą. Grąžinamas objektas yra *CTMessageCenter* klasės egzempliorius, iOS 5 naudojamas SMS žinučių siuntimo funkcionalumui [EKJ+13]. Toks kodas trečiųjų šalių aplikacijose, gali būti naudojamas siųsti žinutes į apmokestinamus numerius. Šiuo būdu išsiųstos žinutes neatsiranda pagrindinėje iOS žinučių gavimo ir siuntimo aplikacijoje [EKJ+13], tad naudotojas apie tokią aplikacijos veiklą sužinotų tik gavęs ryšio sąskaitą.

Nors AppStore platformos tikrinimo procesui pateikiamas tik vykdomasis aplikacijos failas *(angl. executable binary)*, toks privačių metodų kvietimo būdas yra lengvai aptinkamas tinkamumo tikrinimo metu [EKJ+13]. Tam pasitelkiamas UNIX įrankis *grep*, naudojamas raktažodžių paieškai failuose. Funkcijos pavadinimas matomas vykdomojo failo *methname* segmente, talpinančiame

aplikacijoje kviestinų metodų sąrašą. Taip pat karkaso vardas yra matomas aplikacijos importuotų simbolių sąraše. Nors šiame pavyzdyje naudojamas *CoreTelephony.framework* karkasas yra viešas, naudotas antraštinis failas nėra prieinamas įprastiniame SDK pakete. Tad tokio failo importavimas aplikacijos kode aptinkamas statinės analizės metu, kurį vykdo Apple. Norint išvengti aptikimo tinkamumo tikrinimo metu, aplikacija negali importuoti privačių antraštinių failų ir turėti nuorodų į privačius karkasus.

Aptikimo išvengimas pasiekiamas karkasą, klases ir metodus, įkraunant dinamiškai - veikimo metu. Tam galima pasitelkti *Objective-C* dinamiškumą išnaudojančius įrankius [T W13]. Kodas, naudojamas SMS žinučių siuntimui, pasitelkiant dinamiškai įkrautą karkasą:

```
1 NSBundle *coreTelephonyBundle = [NSBundle
2 bundleWithPath:@"/System/Library/Frameworks/CoreTelephony.framework"];
3 [coreTelephonyBundle load];
4 Class messageCenterClass = NSClassFromString(@"CTMessageCenter");
5 id sharedMessageCenter = [coreTelephonyBundle
     performSelector:NSSelectorFromString(@"sharedMessageCenter")];
```

2 pav. Veikimo metu įvykstančio karkaso įkrovimo kodas

Pirmosios dvi kodo eilutes iš 2-jo paveikslo dinamiškai įkrauna *CoreTelephony* karkasą, nenaudojant statinių nuorodų *(angl. linking)* aplikacijos projekte. Kelias failų sistemoje į karkasą yra vienodas visiems iOS įrenginiams, viešai prieinami karkasai yra */System/Library/Frameworks/* aplankale, o privatūs - */System/Library/PrivateFrameworks/*. Privatūs karkasai taip pat gali būti dinamiškai įkrauti naudojant tą patį *NSBundle* metodą. Egzistuojantys tyrimai [EKJ+13], parodė, jog Apple aplikacijų apribojimo mechanizmas *(angl. sandbox)*, netikrina ar parametrai, kuriuos gauna metodas *[NSBundle load]*, nenurodo kelio į karkasus */System/Library/* aplankale.

Trečioje kodo pavyzdžio eilutėje naudojamas metodas *NSClassFromString* atmintyje randa klasę, pagal parametre paduotą simbolių eilutę (Java kalboje šio metodo atitikmuo yra *Class.forName()* metodas). Ketvirtojoje eilutėje įkrautam karkaso kintamajam siunčiama *sharedMessageCenter* žinutė (Objective-C kalboje metodų kvietimas vadinamas „žinutės siuntimu“), kuri grąžina *CTMessageCenter* egzempliorių. Įvykdžius šiuos žingsnius lieka iškviesti metodą, kuris siunčia trumpąsias SMS žinutes.

SMS žinučių metodo kvietimui naudojamas *NSInvocation* objektas:

```
1 NSString *smsText = @"dinaminis SMS siuntimo testas";
2 NSString *addressText = @"+123456789";
3 SEL smsMethodSelector = @selector(sendSMSWithText:serviceCenter:toAddress:);
4 NSMethodSignature *methodSignature = [messageCenterClass
     instanceMethodSignatureForSelector:smsMethodSelector];
```

```
5  NSInvocation *invocation=[NSInvocation
       invocationWithMethodSignature:methodSignature];
6  [invocation setTarget:sharedMessageCenter];
7  [invocation setSelector:smsMethodSelector];
8  [invocation setArgument:&smsText atIndex: 2];
9  [invocation setArgument:&addressText atIndex: 4];
10 [invocation invoke];
```

2 pav. Metodo iškvietimo pavyzdys naudojant *NSInvocation* objektą

NSInvocation pasitelktas dėl dviejų priežasčių:

1. SMS žinutės siuntimo metodas priima tris argumentus, iš kurių du yra privalomi (kitas būdas siųsti žinutę naudojant *[NSObject performSelector:withObject:afterDelay:]* priima daugiausiai vieną argumentą)
2. Naudojant šį būdą yra apsunkinamas privačių metodų kvietimo aptikimas. Metodo aprašas *(angl. signature)* nėra simbolių eilutė - metodo aprašui sukurti naudojamas *SEL* objektas. Kadangi *SEL* objektai nefigūruoja aplikacijos *binary* failo text segmente, statinės analizės metu nėra įmanoma aptikti, jog kviečiamas privatus metodas [EKK+11; T W13].

Pagal Apple *Cocoa* karkaso dokumentaciją, *NSInvocation* objektui sukurti negalima naudoti įprastinio *NSObject* klasės konstruktoriaus [App17a]. Objektui sukurti naudojamas statinis metodas *invocationWithMethodSignature:*, priimantis metodo aprašą reprezentuojančios klasės *NSMethodSignature* egzempliorių. Sukūrus *NSInvocation* objektą, 6-9 eilutėse nurodomas metodo kvietimo žinutės gavėjas (šiuo atveju tai yra *CTMessageCenter* objektas), kviečiamo metodo aprašas, kuris laikomas *SEL* objekte ir metodo argumentai. Pirmosios dvi argumentų pozicijos užimtos kiekvienoje Objective-C kalboje siunčiamoje žinutėje pagal nutylėjimą perduodamais argumentais - *self* ir *_cmd* (atitinkamai reiškiančiais kviečiančios klasės egzempliorių ir selektoriaus, atitinkančio siunčiamą žinutę, objektą), tad kviečiamo metodo argumentai numeruojami nuo trečiosios pozicijos.

Paskutinė priemonė, apsunkinanti privačių metodų kvietimo aptikimą tinkamumo tikrinimo metu yra kodo simbolių eilučių užmaskavimas *(angl. obfuscation)*. Primityvi simbolių eilučių maskavimo technika gali būti įgyvendinta imantis tokių žingsnių:

1. Sukuriama simbolių eilutės konstanta. Jos reikšmė - visi lotyniškos abėcėlės simboliai (mažosios ir didžiosios raidės), skaičiai ir dažniausiai naudojami specialieji ženklai.
2. Simbolių eilutės kodo rašymo metu generuojamos pasirenkant atitinkamus simbolius iš konstantos.

Taip užtikrinama, jog simbolių eilutės (apart visus simbolius turinčios konstantos) nebus aptinkamos *text* segmente. Vis dėlto, maskavimo realizavimas nėra būtinas, nes remiantis įvykdytais tyrimais [EKJ+13], tinkamumo tikrinimo proceso metu nėra tikrinami visi vykdomojo failo *text* segmentai.

4.2. iOS sistemos atakos išnaudojant privačias C kalbos funkcijas

Informacija apie privačius aplikacijų kūrimo interfeisus, jų metodus ir priimamus parametrus Cocoa ir Cocoa Touch karkasuose gaunama iš vykdymo metu *(angl. run-time)* naudojamų antraštinių klasės failų [Ser17a]. Tam naudojami dinaminiai įrankiai (pvz. RuntimeBrowser [Ser17b]), kurie veikimo metu iš atminties nuskaito įkeltų karkasų informaciją. Pavyzdys, kaip tokia informacija išnaudojama atakų prieš iOS sistemą įgyvendinimui, pateikta poskyryje „iOS sistemos atakos naudojantis dinamiškai įkeltais karkasais“.

Tyrėjai atrado, jog Objective-C privačios klasės ir metodai nesudaro visų privačiųjų programinių sąsajų, kurias galima išnaudoti trečiųjų šalių programose [EKJ+13]. Egzistuoja C kalba parašytos funkcijos, kurios nėra matomos Objective-C antraštiniuose failuose. Tačiau vėlgi, norint išnaudoti tokias funkcijas tenka įkelti atitinkamą karkasą, tinkamai iškviesti norimą funkciją. Žemiau pateikiamas kodo iškarpa iš *Applied Cryptography and Network Security* konferencijoje J. Han ir S. M. Kywe atliktos prezentacijos, parodančios, kaip naudojantis privačiomis C funkcijomis, įmanoma sužinoti įrenginio PIN kodą.

```
1 void *mobileKeyBag =
      dlopen("/System/Library/PrivateFrameworks/MobileKeyBag.framework/MobileKeyBag",
      1);
2 int (*f)(id, id, id) = dlsym(mobileKeyBag, "MKBKeyBagChangeSystemSecret");
3 //...
4 int r = f(oldpwd, newpwd, pubdict);
5 //...
```

3 pav. Įrenginio PIN kodo sužinojimas naudojant privačias C kalbos funkcijas iOS programų kūrimo pakete

Pirmojoje eilutėje panaudota funkcija *dlopen()* į atmintį įkrauna privatų karkasą *MobileKeyBag* ir grąžina nenurodyto tipo nuorodą *(angl. opaque pointer)* į įkrautąjį karkasą. Naudojant šią nuorodą funkcijoje *dlsym()* gaunamas atminties adresas, kuriame yra funkcijos *MKBKeyBagChangeSystemSecret* realizacijos pradžia. Gautasis adresas priskiriamas funkcijos nuorodos tipo kintamajam, priimančiam tris parametrus, tam, jog vėliau galėtų būti naudojamas kaip funkcija. Tai matoma trečioje eilutėje, kur funkcijos nuorodos kintamasis *f* kviečiamas su trimis parametrais.

Didžioji darbo dalis, kai norima išnaudoti privačias C funkcijas, yra tinkamų parametrų padavimas atrastoms funkcijoms ir jų grąžinamų reikšmių interpretavimas. Dažniausiai karkasas turimas tik atmintyje, tad veikimo principų supratimui reikia analizuoti asemblerio instrukcijas. Net jei išsiaiškinami teisingi parametrai, gali paaiškėti, jog funkcijos vykdymas neįmanomas dėl iOS programėlių apribojimo mechanizmo *(angl. sandboxing)*.

Šiuo atveju išsiaiškinta, jog *MKBKeyBagChangeSystemSecret* funkcija skirta įrenginio slaptažodžio pakeitimui ir jos vykdymo, net ir žemomis privilegijomis, neriboja iOS sistema [EKJ+13]. Pirmieji du parametrai (abu jie yra *NSData* * tipo, naudojamo dvejetainei informacijai perduoti),

kuriuos priima funkcija, reprezentuoja senojo ir naujojo slaptažodžio reikšmes. Trečiasis - *NSDictionary* tipo, skirtas laikyti slaptažodžio „klaviatūros tipą“. Tokiai informacijai pasiekti pasitelkta kita privati C programavimo sąsajos fukncija *MKBKeyBagCopySytemSecretBlob* iš *MobileKeyBag* karkaso. Šios funkcijos grąžinama reikšmė naudojama kaip trečiasis *MKBKeyBagChangeSystemSecret* funkcijos parametras.

Įvykdžius šiuos žingsnius galima sėkmingai iškviesti slaptažodžio keitimo funkciją. Kadangi naudojama žemo lygio C kalba realizuota funkcija nesėkmingi bandymai neiššaukia „klaidingo slaptažodžio“ žinutės, taigi bandymai yra neriboti. Tai leidžia vykdyti jėgos metodo ataką prieš iOS įrenginį. iOS įrenginiuose dažniausiai naudojami 4 arba 6 skaitmenų PIN kodai [EKJ+13]. Šiuo atveju ataka bandyta su įrenginiu, kuris apsaugotas keturių skaitmenų kodu, tad įmanomi 10^4 skirtingi slaptažodžių variantai. Eksperimento vykdymo metu iPhone 5 įrenginio PIN kodo sužinojimui vidutiniškai reikia 18.2 minutės, per sekundę išbandomi 9.2 skirtingi kodai [EKJ+13].

Atakos greitį galima optimizuoti pirmiausiai tikrinant dažniausiai pasitaikančius PIN kodus, pvz. skaičių kombinacijas, kurios yra taisyklingos gimimo dienos ir mėnesio kombinacijos. Jei PIN kodas yra sudarytas iš vadovaujantis tokia logika, prieiga prie iPhone įrenginio gaunama per 40 sekundžių [EKJ+13]. Ši ataka tampa nebeaktuali, jei slaptažodžiui naudojamas didelis skaičius simbolių, nes stipriai išauga atakos vykdymo laikas. Naujausiose iOS versijose mažiausias skaitmenų kiekis PIN kodui yra 6 simboliai [Ser17a].

4.3. Atakų prieš iOS sistemą išnaudojimo reikšmė

Ankstesniuose poskyriuose aprašytų dviejų tipų, SMS žinučių siuntimo ir PIN kodo sužinojimo, atakų įgyvendinimo idėja gali būti panaudota kuriant naujus sistemos išnaudojimo būdus. Pasitelkiant dinaminius Objective-C veikimo įrankius įmanoma apeiti Apple tinkamumo tikrinimo procesą, visų pirma randant privačius metodus, kurių vykdymas neapribotas iOS apsaugos mechanizmų.

Programavimo sąsajos skirtos SMS ir el. laiškų siuntimui iOS 7 versijoje perdarytos nuo pagrindų. Naujosios karkasų versijos ištaisė daugelį sistemos išnaudojimo spragų, kurios buvo paviešintos saugumo ekspertų [App15]. Vis dėlto, kasmet išleidžiamos iOS versijos papildomos nauju funkcionalumu, kurį įgyvendina tik vidiniuose Apple testuose išbandytas kodas. Natūralu, jog su kiekviena versija didėja saugumo ekspertų neištirto kodo kiekis. Toks kodas potencialiai gali pasitarnauti naujoms atakoms prieš iOS sistemą. Norint sustiprinti iOS apsaugą prieš tokio tipo atakas, yra reikalinga peržiūrėti tinkamumo tikrinimo procesą, į jį įtraukiant apsaugą nuo dinamiškai vykdomo kodo, taip pat išplėsti iOS programėlių apribojimo *(angl. sandboxing)* mechanizmą.

5. Android and iOS operacinų sistemų saugumo priemonių analizė

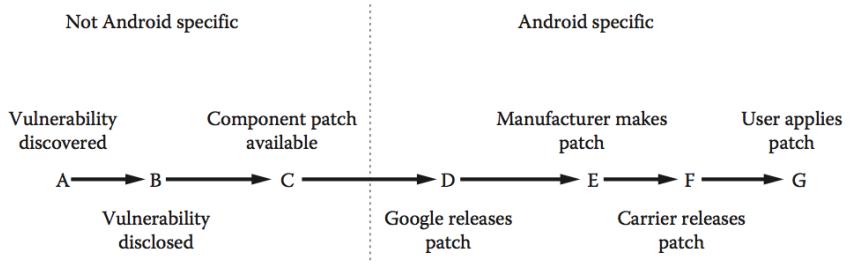
Dviejų didžiausių [App17c] mobiliųjų įrenginių operacinių sistemų, iOS ir Android, lyginimas retai sulaukia vieningo vertinimo. Tai yra teisinga ir saugumo priemonių analizei: yra teigiama, jog Android sistema pranašesnė, dėl aplikacijoms suteikiamų teisių sąrašo matomumo instaliavimo metu ir atviro kodo komponentų naudojimo sistemoje [For16]. Vis dėlto kiti saugumo ekspertai išreiškia nuomonę, jog iOS saugumas yra pažangesnis dėl Apple naujų aplikacijų priėmimo proceso į AppStore platformą ir dėl Apple monopolio iOS sistemai, kas reiškia greitesnius saugumo spragų ištaisymus ir pateikimus visiems palaikomiems įrenginiams [Hof15]. Retkarčiais pasigirsta nuomonių, jog abi sistemos turi vienodus saugumo lygmenis, tačiau įgyvendintus skirtingomis priemonėmis [She17]. Tokia nuomonių įvairovė yra tinkamas veiksnys darkart pažvelgti į abiejų platformų siūlomą funkcionalumą ir pateikti saugumo priemonių analizę.

Šis skyrius pradedamas nuo trumpos Android saugumo iššūkių apžvalgos, vėliau pateikiant bendrus platformų saugumo priemonių sąlyčio taškus. Toliau suformuluojamas platformų saugumo priemonių lyginimo karkasas, pasitarnaujantis vertinant abiejose platformose išleistų aplikacijų naudojamas veikimo privilegijas ir jų naudojimo būdus.

5.1. Saugumo iššūkiai Android sistemoje

Android sistemos padėtis yra unikali - tai yra daugiausiai vartotojų turinti platforma [App17c]. Android pagrindas atviro kodo kernelis Linux, sistema platinama įvairių kompanijų įrenginiuose, kurios modifikuoja programinę įrangą pagal savo vizijas. Toks platinimo modelis sukelia vieną rimčiausių Android trūkumų - ilga atnaujinimų pateikimo trukmė naudotojams.

5.1.1. Android saugumo spragų taisymo ciklas



4 pav. Android saugumo spragų ištaisymo ciklas. Nuo spragos atradimo iki naudotojų įrenginių pasiekimo. [VVC11]

Viršuje esantis paveikslas vaizduoja Android saugumo atnaujinimo ciklą, nuo pažeidžiamos sistemos vietos atradimo iki naujosios versijos platinimo. Pirmieji trys ciklo žingsniai (A-C) yra įprastiniai bet kokiam programinės įrangos produktui (tarp jų ir iOS sistemai):

A. Atrandamas pažeidžiamumas;
B. Pažeidžiamumas išviešinamas;
C. Sukuriama pažeisto komponento pataisa.

Tačiau dėl Android platinimo specifikos reikalingi papildomi žingsniai:
D. Google išleidžia pataisą gamintojams;
E. Gamintojai pritaiko pataisą savo palaikomai Android versijai;
F. Ryšio tiekėjai išleidžia gautąją pataisą. Šis žingsnis figuruoja jei įrenginys yra pritaikytas tik tam tikram ryšio tiekėjui, pvz. Verizon;
G. Naudotojas atnaujina savo sistemą.

Remiantis atliktais eksperimentais nuo spragos atradimo Android sistemoje, iki naudotojui pasiekiamo atnaujinimo praeina bent dviejų mėnesių laikotarpis [VVC11]. Toks atnaujinimų vėlinimas sukelia padidintą pažeidžiamumo riziką: atradus spragą Linux branduolyje arba WebKit tinklapių vaizdavimo variklyje (ir kituose atviro kodo Android sistemos komponentuose), ji greit ištaisoma atviro kodo bendruomenės entuziastų [VVC11]. Tačiau dėl ilgo proceso, kol atnaujinimas pasiekia vartotojus, jų naudojamos sistemos yra pažeidžiamos viešai žinomų sistemos silpnybių. Taip pat, dėl Linux branduolio populiarumo ir pritaikymo platumo, žemo lygio saugumo spragų ieškojimui ir išnaudojimui Android sistemoje, nereikia išmokti naujų technologijų. Tai nėra teisinga iOS atveju, kurioje naudojamos nišinės technologijos - Objective-C kalba, Darwin operacinė sistema.

Ši Android sistemos problema gali būti išspręsta tik artimesniu bendradarbiavimu tarp Google ir įrenginių gamintojų.

5.1.2. Android aplikacijų perdarymas ir platinimas

Java yra oficiali Android programėlių kūrimo kalba. Dėl virtualiosios Java mašinos generuojamo nuo platformos nepriklausomo kodo, Android programoms yra lengviau taikyti apgrąžos inžinerijos procesą *(angl. reverse engineering)* nei iOS aplikacijoms, kurios žemo lygio kodas *(angl. assembly code)* yra specifinis vienai platformai [Nol12]. Dėl apgrąžos proceso prieinamumo Android aplikacijose, supaprastėja trečiųjų šalių produktų modifikavimas įdedant žalingą kodą [HHW+12; ZZJ+12]. Šios problemos sprendimo būdai siūlyti statinės analizės [CGC12; LLW+12] ir dinaminiai modifikuoto kodo aptikimo įrankiai [ZZD+12], tačiau žalingos Android aplikacijos vis dar yra pagrindinė platformos saugumo grėsmė. Svarbiausias to veiksnys yra tai, jog platforma neuždraudžia instaliuoti modifikuotų aplikacijų platinamų ne Google programėlių parduotuvėje *Play Store*.

5.2. iOS ir Android platformu saugumo palyginimas

1 lentelė. iOS ir Android mobiliųjų įrenginių operacinių sistemų saugumo ypatybių palyginimas

Saugumo ypatybė	*Android*	*iOS*
Programėlių privilegijų prašymo pranešimai	Yra	Dalinai
Tinkamumo tikrinimo procesas aplikacijoms	Dalinis	Yra
Programėlių kodo pasirašymas *(angl. digital signing)*	Yra	Yra
Vykdomojo kodo failo šifravimas *(angl. binary encryption)*	Nuo 4.1 versijos (2011 m.)	Yra
Programėlių apribojimo mechanizmas *(angl. sandboxing)*	Yra	Yra
Naudotojo duomenų šifravimas	Yra	Yra
Žalos kontrolė	Yra	Yra
Atminties adresų erdvės išdėstymo randomizacija *(angl. address space layout randomization)*	Nuo 4.0 versijos (2011 m.)	Nuo 4.3 versijos (2011 m.)

- *Naujų privilegijų aplikacijoms prašymo pranešimai*: Android ekosistemos aplikacijos turi išreikštinai apibrėžti norimas gauti veikimo privilegijas. Aplikacijos instaliavimo metu, naudotojui rodomas privilegijų sąrašas, kurias ši aplikacija gautų, jei būtų įrašyta įrenginyje. Šiame žingsnyje naudotojas gali pasirinkti atsisakyti įrašyti programėlę. Android sistema turi apie 130 skirtingų privilegijų tipų [Goo17]. iOS sistemoje aplikacijos įrašymo metu gauna numatytąjį privilegijų rinkinį, kurį, veikimo metu norint prieiti prie tam tikrų sistemos resursų, gali praplėsti. Iki iOS 5 versijos egzistavo tik dvi išreikštinio vartotojo sutikimo reikalaujančios sistemos funkcijos - įrenginio lokacijos informacija ir pranešimų *(angl. push notification)* siuntimas. Nuo iOS 6 iki iOS 10 versijų pridėtos šios aplikacijoms prieinamos privilegijos ir prieiga prie resursų, reikalaujančios išreikštinio vartotojo leidimo naudotis:
  1. Adresų knygelės prieiga
  2. Kalendoriaus ir priminimų prieiga
  3. Nuotraukų prieiga
  4. Bluetooth failų dalinimasis
  5. Garso įrašymas
  6. Kamera
  7. Sveikatos informacija (valdoma karkaso *HealthKit*)
  8. Daiktų interneto galimybės (valdomos karkaso *HomeKit*)

  Net ir praplėtus iOS sistemoje suteikiamų privilegijų spektrą, neprilygstama Android aplikacijoms suteikiamam kontrolės lygiui.
- *Aplikacijų tinkamumo tikrinimo procesas*: Kiekviena aplikacija, kurią norima platinti iOS platformoje prieš tai patikrinama Apple kompanijos tikrinimo procese *(angl. vetting process)*. Ieškoma žalingo kodo, privatumo politikos ir kūrimo gairių pažeidimų. Žingsniai, kuriuos atlieka Apple, nėra viešai žinomi. Yra pasitaikę atvejų, kai į *AppStore* programėlių platforma pateko žalinga aplikacija, sėkmingai praleista tikrinimo proceso [Sor10].

  Android kūrėjai naudoja įrankį pavadinimu *Bouncer* [Loc12], skirtą automatinei naujų aplikacijų ir jų atnaujinimų analizei. Šis įrankis skanuoja produktus jau esančių programėlių

platinimo platformoje *Play Store*, tad kūrėjams nereikia laukti platinimo pradžios patvirtinimo. *Bouncer* įrankiui aptikus žalingą aplikaciją, produkto platinimas nutraukiamas.

- *Pasirašymas ir šifravimas*: iOS ir Android aplikacijos yra pasirašomos skaitmeniniu sertifikato sukurtu parašu. Tačiau egzistuoja esminis skirtumas - sertifikato kilmė. iOS aplikacijos pasirašomos pačios Apple sugeneruotu sertifikatu, o Android - aplikacijos kūrėjų. Android platformoje pagrindinis parašo panaudojimas - resursų dalinimasis tarp aplikacijų pasirašytų tuo pačiu privačiu raktu (aplikacijos priskiriamos vienodam proceso identifikatoriui).
  iOS aplikacijos yra šifruojamos, tam, kad apsaugoti nuo neautorizuoto platinimo ir modifikavimo. Prieš pradedant veikti programai, kiekviena iš *AppStore* atsiųsta aplikacija, įrenginio atmintyje yra dekoduojama, patikrinant jos atitikimą raktui, su kuriuo buvo pasirašyta. Jei rasta neatitikimų - veikimas nutraukiamas. Nuo Android 4.1 versijos naudotojo įsigyti produktai iš *Play Store* yra šifruojami specifiniu įrenginio raktu.
- *Kitos ypatybes*: Skiriasi platformose naudojami aplikacijų atskyrimo mechanizmai:
  - iOS aplikacijos egzistuoja savo konteineryje *(angl. sandboxing)*. Veikdamos žemu sistemos modifikavimo lygmeniu, joms yra apribojamas išorinių resursų pasiekimas, plečiamas įrankiais aprašytais trečiajame skyriuje.
  - Android naudoja *UNIX UIDs* suteikiamą funkcionalumą, aplikacijų atskirčiai užtikrinti. Aplikacijos turi galimybę prašyti *root* naudotojo prieigos.

  Abi platformos suteikia vartotojo duomenų šifravimo funkcionalumą ir atminties išvalymo opciją įrenginio praradimo atveju. Taip pat iOS ir Android kūrėjai, nuotoliniu būdu gali iš naudotojo įrenginio ištrinti aplikaciją, kuri atlieka žalingą veiklą.
  Nuo Android 4.0 ir iOS 4.3 versijų, platformose veikia atminties adresų erdvės išdėstymo randomizacija. Ši apsaugos priemonė dinamiškai išdėlioja sistemos atmintyje esančią informaciją, operacinės sistemos komponentus. Taip yra apsunkinamas spragų išnaudojimo realizavimas, nes nėra žinoma konkrečios informacijos vieta atmintyje.

Ši aukšto lygio ypatybių apžvalga leidžia teigti, jog platformose naudojamos panašios saugumo priemonės, tačiau su dviem esminiais skirtumais:

1. Android platformoje naudojamas privilegijų mechanizmas sprendimą dėl aplikacijai suteikiamų teisių atlikti leidžia naudotojui. Tačiau nėra žinoma ar vartotojas perskaito ir supranta tai, kas išdėstyta aplikacijos instaliavimo lange [FHE+12].
2. iOS sistemoje tinkamumo tikrinimo procesas suteikia tam tikrą papildomą saugumo lygmenį prieš žalingus produktus. Kaip minėta anksčiau *AppStore* platformoje yra plitusios kenksmingos programėlės, bylojančios apie proceso trūkumus.

Kaip minėta ankstesniame poskyryje, Android aplikacijos rašomos Java ir veikia ART *(angl. Android Runtime)* technologijos pagalba, kuri aplikacijos *bytecode* failo turinį perdaro į aparatinės įrangos instrukcijas [Fru14]. iOS aplikacijos įgyvendinamos naudojant Objective-C ir (arba) Swift programavimo kalbas, kurių kompiliavimo produktai naudojami ARM architektūros aparatinėje įrangoje [Sea01]. Tokie technologijų skirtumai reiškia platformų architektūrinius ir įgyvendinimo neatitikimus, iškeldami objektyvios platformų saugumo ypatybių analizės svarbą. Vienas iš būdų,

kuriuo naudojantis galima analizuoti iOS ir Android saugumo ypatybės - lyginti abiejose platformose veikiančias trečiųjų šalių programėles [HYG+13], siūlančiose tokį patį funkcionalumą (pvz. *Twitter* socialinio tinklo pagrindinė programėlė).

Pagrindinis daugiaplatforminių *(angl. cross-platform)* aplikacijų lyginimo aspektas - naudojamų programavimo sąsajų kiekis, kurių pagalba prieinama ar kontroliuojama naudotojo privatumui svarbi informacija: kamera, Bluetooth ryšys ir pan. Tokios aplikacijų programavimo sąsajos vadinamos SS-API *(angl. Security-Sensitive Application Programming Interface)* [HYG+13]. Pirmasis žingsnis - rasti tas aplikacijoms suteikiamas privilegijas, kurios egzistuoja abiejose platformose.

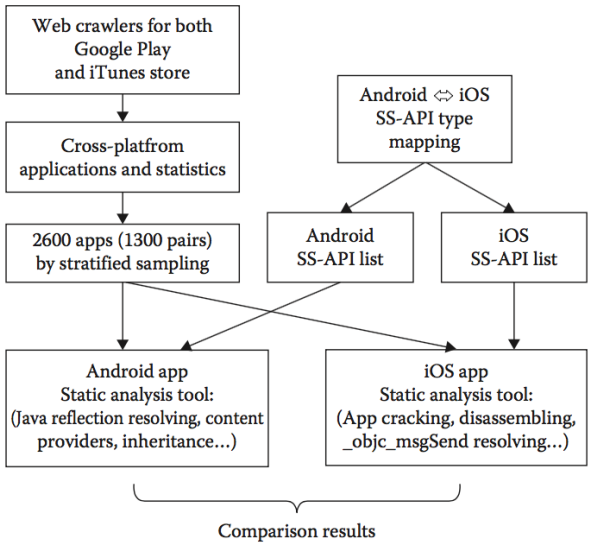
5.2.1. Android ir iOS platformose palaikomos aplikacijų privilegijos

2 lentelė. Saugumo privilegijos galimos suteikti trečiųjų šalių aplikacijoms, palaikomos abiejose (iOS ir Android) mobiliosiose operacinėse sistemose [HYG+13]

SS-API tipas	Reikšmė
ACCESS_LOCATION	Suteikia prieigą prie įrenginio buvimo vietos
ACCESS_NETWORK_INFO	Leidžia pasiekti tinklo informaciją
BATTERY_STATS	Leidžia rinkti įkrovos statistiką
BLUETOOTH	Leidžia jungtis prie Bluetooth įrenginių
BLUETOOTH_ADMIN	Leidžia atrasti ir susieti *(angl. pair)* Bluetooth įrenginius
CALL_PHONE	Leidžia inicijuoti skambutį
CAMERA	Leidžia prieigą prie įrenginio kameros
CHANGE_WIFI_MULTICAST_STATE	Leidžia aplikacijoms naudoti *Wi-Fi multicast* funkcionalumą
FLASHLIGHT	Leidžia įrenginio blykstės naudojimą
INTERNET	Leidžia atidaryti tinklo kanalus *(angl. network sockets)*
READ_CALENDAR	Leidžia prieigą prie naudotojo kalendoriaus duomenų
READ_CONTACTS	Leidžia prieigą prie naudotojo adresų knygelės duomenų
RECORD_AUDIO	Leidžia aplikacijai įrašyti garsą
VIBRATE	Suteikia prieigą prie vibravimo įrenginio
WAKE_LOCK	Leidžia išjungti automatinio įrenginio užsirakinimo ir (arba) ekrano užtamsinimo funkcijas
WRITE_CALENDAR	Leidžia į naudotojo kalendorių įrašyti duomenis
WRITE_CONTACTS	Leidžia į naudotojo adresų knygelę įrašyti duomenis

2-joje lentelėje esančios privilegijos aprėpia prieigą prie visų įprastinių resursų ir įrenginio funkcionalumo - naudotojo duomenys (kontaktai, kalendorius), Bluetooth ryšys, tinklo būsena, kamera ir t.t.

5.3. Operaciniu sistemų saugumo lyginimo metodika



5 pav. Aplikacijų SS-API interfeisų lyginimo metodikos žingsniai [HYG+13]

Analizė atliekama pasitelkus informaciją iš 2-osios lentelės ir 5-jame paveiksle pavaizduotą lyginimo metodiką [HYG+13]. Privačių, vartotojo informaciją naudojančių, aplikacijų programavimo sąsajų skirtumų lyginimas operacinėse sistemose, turi du pagrindinius žingsnius:

1. Surasti aplikacijas, turinčias atitikmenis abiejose operacinėse sistemose. Šį žingsnį atlieka *web-crawlers* renkantys informaciją iš milijonus produktų turinčių aplikacijų parduotuvių *iTunes App Store* ir *Google Play*. Surinkus informaciją apie platformų programėles, duomenys išfiltruojami pagal norimus raktažodžius. Randamos multiplatforminės aplikacijos.
2. Statinės analizės atitikimas. Iš platinimo platformų atsisiųstų aplikacijų vykdomiesiems failams atliekama statinė analizė naudojantis atitinkamos sistemos įrankiais. Android sistemoje *Dalvik bytecode* rinkiniams, iOS - *Objective-C* kompiliavimo produktams. Galiausiai lyginama rasta informacija apie SS-API naudojimo kiekį.

5.4. Lyginimo rezultatai ir analizė

Naudojamų SS-API lyginimas atliktas 1300-tams porų aplikacijų [HYG+13]. Rezultatai rodo, jog iš lygintų aplikacijų, 73% iOS platformos produktų naudoja daugiau SS-API prieigos, nei jų atitikmenys Android platformoje. Dažniausiai šie papildomi SS-API naudojami prieigai prie įrenginio identifikacinės informacijos, kameros, vartotojo kontaktų, kalendoriaus.

Įdomu pažvelgti į naudojamų programavimo sąsajų koreliacijas multiplatforminėse aplikacijose. Rezultatai rodo, 1300-tuose lygintų Android aplikacijų SS-API skaičius yra 4582. Vidutiniškai viena Android aplikacija naudoja 3.5 aplikacijų programavimo sąsajos, kuri suteikia prieiga prie privačios informacijos [HYG+13]. Tuo tarpu 1300-ai lygintų iOS aplikacijų SS-API išvis naudoja 7739 kartus, kas vienam produktui reiškia 5.9 saugumo privilegijas, teoriškai suteikiančias prieigą prie privačios informacijos. 948-ios (73%) iOS aplikacijos naudoja daugiau SS-API nei jų

atitikmenys Android platformoje [HYG+13].

Kai kurių iš 2-ojoje lentelėje pateiktų saugumo privilegijų naudojimas yra beveik vienodas abiejose platformose. Prieiga prie interneto (*INTERNET* saugumo privilegija) naudojama 1247-iose Android ir 1253-jose iOS aplikacijose. Tačiau kameros prieiga (*CAMERA* saugumo privilegija) naudojama tik 172-jose Android aplikacijose ir 601-oje iOS aplikacijoje [HYG+13].

5.4.1. Skirtumai aplikacijų įgyvendinime

iOS ir Android saugumo privilegijų naudojimo skirtumai paaiškinami dvejomis ypatybėmis:

1. *Implementacijų skirtumai atitinkamose multiplatforminėse aplikacijose*. WordPress aplikacijoje tinklo informacijos programavimo sąsaja (ACCESS_NETWORK_INFO saugumo privilegija) naudojama tik iOS versijoje. iOS aplikacijoje naudojamas *Reachability* karkasas tikrina tinklo prieigą, kviečiant ACCESS_NETWORK_INFO saugumo privilegiją naudojančius programavimo sąsajos metodus (kodas paimtas iš *WordPress* aplikacijos repozitorijos - `https://github.com/wordpress-mobile/WordPress-iOS`):

```
1
2  @property (nonatomic, strong, readwrite) Reachability *internetReachability;
3
4  - (void)setupReachability
5  {
6      // Setup Reachability
7      self.internetReachability = [Reachability
           reachabilityForInternetConnection];
8
9      __weak __typeof(self) weakSelf = self;
10
11     void (^internetReachabilityBlock)(Reachability *) = ^(Reachability
           *reach) {
12    //...
13         weakSelf.connectionAvailable = reach.isReachable;
14     };
15     //...
16     self.connectionAvailable = [self.internetReachability isReachable];
17 }
```

6 pav. *Reachability* karkaso naudojimas interneto prieigos tikrinimui atviro kodo *WordPress* aplikacijoje

Septintoje eilutėje `[Reachability reachabilityForInternetConnection]` metodo grąžinamas objekto laukas `isReachable` nusako ar tinklas yra pasiekiamas. *WordPress* iOS aplikacijoje tinklo prieigos informacija naudojama norint atlikti veiksmus susijusius su informacijos perdavimu į serverius (pvz. naujo įrašo į tinklaraštį pridėjimo metu), jei tinklas

nepasiekiamas (`isReachable` laukas grąžino neigiamą reikšmę), naudotojui rodomas klaidos pranešimas. Android aplikacijoje bandoma siųsti duomenis ir tada tikrinama funkcijos grąžinama reikšmė, atspindinti siuntimo proceso baigtį [HYG+13].

Implementacijos skirtumai yra vienas iš pavyzdžių lemiančių naudojamų SS-API skaičiaus skirtumus multiplatforminėse aplikacijose.

2. *Dėl instaliavimo metu matomų aplikacijai suteikiamų privilegijų trečiųjų šalių Android kūrėjai stengiasi išvengti SS-API naudojimo*. Šis reiškinys aiškinamas pasitelkiant atviro kodo *WordPress* aplikaciją [HYG+13]. Lyginant su Android atitikmeniu iOS WordPress aplikacija papildomai naudoja READ_DEVICE_ID programavimo sąsajos metodus:

```
1  - (void)runStats {
2    NSString *deviceModel = [[[UIDevice currentDevice] platformString]
         stringByUrlEncoding];
3    NSString *deviceuuid = [[UIDevice currentDevice] uniqueIdentifier];
4    NSDictionary *info = [[NSBundle mainBundle] infoDictionary];
5    NSString *appversion = [[info objectForKey:@"CFBundleVersion"]
         stringByUrlEncoding];
6    NSLocale *locale = [NSLocale currentLocale];
7    NSString *language = [[locale objectForKey: NSLocaleIdentifier]
         stringByUrlEncoding];
8    NSString *osversion = [[[UIDevice currentDevice] systemVersion]
         stringByUrlEncoding];
9    int num_blogs = [Blog countWithContext:[self managedObjectContext]];
10   NSString *numblogs = [[NSString stringWithFormat:@"\%d", num_blogs]
         stringByUrlEncoding];
11
12   //.. siuntimas
13 }
```

7 pav. Įrenginio identifikacinės informacijos naudojimas *WordPress* iOS aplikacijoje

WordPress iOS aplikacija nuskaito įrenginio unikalų indentifikacinį kodą (trečia eilutė), programėlės versiją (penkta eilutė), operacinės sistemos versiją (aštunta eilutė) ir t.t. Ši informacija siunčiama į `http://api.wordpress.org/iphoneapp/update-check/1.0/`, tam, kad patikrinti, ar yra programėlės atnaujinimų. Android aplikacijoje taip pat tikrinama ar programėlę reikia atnaujinti - `wpAndroid` klasėje `uploadStats` metodas surenka panašų informacijos rinkinį ir jį siunčia į *WordPress* serverius, gaudamas atsakymą ar programėlė yra senesnės versijos. Tačiau tarp iOS ir Android funkcionalumo įgyvendinimo egzistuoja vienas pagrindinis skirtumas - Android platformoje išsiverčiama be prieigos prie unikalių įrenginio identifikatorių.

Pateiktame iOS kode matoma, jog įrenginys identifikuojamas kviečiant `UIDevice` klasės metodą `uniqueIdentifier`, grąžinantis įrenginiui unikalią simbolių eilutę. Android versijoje

unikalią simbolių eilutę atitinka atsitiktinai sugeneruota UUID reikšmė. Ši reikšmė įrašoma į Android aplikacijoje naudojama SQLite duomenų bazę ir yra naudojama kaskart ieškant programėlės atnaujinimų.

Priežastis, kodėl iOS ir Android aplikacijų kūrėjai pasirenka skirtingus būdus įgyvendinti tam pačiam funkcionalumui multiplatforminiuose produktuose, išlieka ta pati. Android aplikacijų instaliavimo metu naudotojai mato sąrašą privilegijų (tarp jų ir SS-API atinkančias prieigas), kurios bus suteiktos programai, ją įrašius. iOS programėlių įrašymo procesas nesuteikia tokios informacijos [HYG+13].

Tai lėmė ir iki iOS 6 versijos multiplatforminėse aplikacijose naudotų READ_CONTACTS ir READ_CALENDAR programavimo sąsajų kiekių skirtumą. Visose iOS versijose iki iOS 6 naudotojui nebuvo rodomas pranešimas, kai programėlė naudodavo informaciją iš kalendoriaus, kontaktų, nuotraukų ar priminimų [HYG+13].

Rezultatai ir išvados

Darbe atskleidžiami iOS operacinės sistemos saugumo trūkumai ir pateikiamos rekomendacijos spragų ištaisymui. Pirmuose trijuose skyriuose atlikta iOS operacinės sistemos technologijų, architektūrinių sprendimų ir saugumo ypatybių analizė. Šis iOS sistemą sudarančių komponentų tyrimas yra tolimesniuose skyriuose išskirtų sistemos kompromitavimo būdų pagrindas. Sistemoje naudojamų technologijų ypatybės (*Objective-C* kalbos dinamiškumas, aplikacijų trečiųjų šalių kodo naudojimas), remiantis saugumo ekspertų atliktais eksperimentais, susietos su galimais išnaudojimo būdais, pateikti jų kodo pavyzdžiai. Sekantis darbo etapas - iOS sistemos saugumo lyginimas su Android operacine sistema. Pasirinkta lyginimo metodika atsižvelgianti į sistemose veikiančių aplikacijų naudojamų privilegijų kiekį. Kaip skirtumų įrodymo pavyzdys pasitelkta abiejose platformose platinama atviro kodo aplikacija *WordPress*. Galiausiai išskirti trys galimi iOS operacinės sistemos saugumo patobulinimai, suformuluoti remiantis atlikta saugumo spragų išnaudojimo ir lyginimo analize.

Rezultatai

1. Ištirtos iOS saugumo priemonės, didžiausią svarbą suteikiant sprendimams, kurie apsprendžia sistemos naudojimo ypatybes
2. Remiantis praeityje aptiktomis ir viešai prieinamomis saugumo spragomis pateikti galimi sistemos saugumo spragų išnaudojimo būdai
3. Išanalizuotos Android ir iOS operacinių sistemų saugumo priemonės, naudojant saugumo ekspertų sukurta lyginimo metodiką, paremta sistemose aplikacijoms suteikiamomis privilegijomis
4. Naudojantis analizės rezultatais suformuluoti galimi iOS sistemos saugumo stiprinimo būdai

Išvados

1. iOS operacinėje sistemoje naudojamos įvairaus pobūdžio saugumo priemonės. Saugumo karkaso apimami moduliai prasideda nuo žemiausio lygmens - sistemos leidyklės *(angl. bootloader)*, kurio įkraunamas kodas šifruojamas, iki operacinės sistemos vietų, su kuriomis sąveikauja pats naudotojas - programėlių, jų plėtinių. Tačiau aukščiausio lygio iOS operacinės sistemos saugumo būklę apsprendžia du pagrindiniai mechanizmai - programėlių kodo pasirašymas ir veikiančiųjų procesų apsauga.
    - *Programėlių kodo pasirašymas.* Darbo metu išsiaiškinta, jog kodo pasirašymas naudojamas kaip pagrindinis barjeras, užkertantis kelią kenksmingų aplikacijų plitimui iOS platformoje. Programėlės pasirašomos Apple išduodamu sertifikatu, kuris gaunamas tik įrodžius kūrėjo tapatybę, taip apsisaugant nuo anonimiško produktų platinimo *iTunes AppStore* aplikacijų parduotuvėje. Skaitmeninis kodo parašas naudojamas tikrinti leidžiamo kodo integralumui - bandant paleisti programėlę, kurios kodas buvo modifikuotas, iOS sistema aptinka parašo pokyčius ir užbaigia proceso darbą.
    - *Veikiančiųjų procesų apsauga.* Kodo pasirašymo mechanizmo pradėtą sistemos apsaugos darbą tęsia vykdomų procesų apsaugos modulis. iOS operacinėje sistemoje visos

aplikacijos veikia atskiruose konteineriuose *(angl. sandbox)*, panaikinančiuose prieigą prie kitų aplikacijų duomenų ir kontroliuojančiuose sisteminių resursų naudojimą.
Aplikacijos iOS sistemoje veikia kaip neprivilegijuotas *mobile* naudotojas, sisteminiai failai yra skaitymo rėžime *(angl. read-only mode)*, kuris užtikrinamas aparatinės ARM architektūros funkcionalumu, pažyminčiu atminties sektorius kaip nevykdomus *(angl. non-executable)*. Aplikacijose prieinamose viešuose iOS kūrimo paketo priemonėse nėra galimybės eskaluoti proceso privilegijas (veikti kaip *root* naudotojui) ir modifikuoti sistemos veikimą.
Dėl veikimo *mobile* naudotojo teisėmis aplikacijos negali apeiti kodo pasirašymo mechanizmo.

2. Lengviausiai prieinamas iOS operacinės sistemos saugumo kompromitavimo būdas - privačių aplikacijų programavimo sąsajų išnaudojimas. Privačios aplikacijų programavimo sąsajos iOS egzistuoja ir privačiuose, ir viešuose iOS kūrimo paketo karkasuose. Neviešinamų metodų naudojimas trečiųjų šalių aplikacijose suteikia papildomas privilegijas, leidžiančias naudotis tik sisteminiam kodui prieinamu funkcionalumu. Apple uždraudė tokią praktiką: programėlės tinkamumo tikrinimo *(angl. vetting)* metu, aptikus privataus metodo kvietimą, pateiktas produktas yra atmetamas. Tad didžiausias atakos vektoriaus įgyvendinimo iššūkis - atmetimo tikrinimo proceso metu išvengimas. Remiantis saugumo ekspertų atliktais darbais, išskirtos dvi praktikos, potencialiai leidžiančios išvengti atmetimo:
   - *Atakos naudojant dinamiškai įkrautais karkasais.* iOS kūrimo paketas suskirstytas į karkasus. Vieni iš jų dalinai prieinami trečiųjų šalių kūrėjams (vieši karkasai), dalis tik Apple inžinieriams (privatūs karkasai). Dinaminis karkaso įkėlimas reikalingas, tam, jog tikrinimo proceso metu vykdomojo failo TEXT segmentuose nebūtų aptinkami antraštiniai failai aprašantys privačius metodus, kurie gaunami atlikus paketo analizę, ir nereiktų pasitelkti statinių nuorodų į privačius karkasus naudojimo.
     Išnaudojant *Objective-C* kalbos dinamiškumą įmanoma veikimo metu įkrauti karkasą. Tam naudojamas iOS kūrimo pakete viešai prieinamas objektas *NSBundle* ir jo metodai. Įkrautajam karkasui siunčiamos žinutės (kviečiami metodai), leidžiantys pasiekti privatų funkcionalumą.
   - *Atakos naudojančios privačias C kalbos funkcijas.* iOS kūrimo paketo statinės analizės metu gautų karkasų antraštinių failų turinys neatspindi viso karkasų funkcionalumo. Saugumo ekspertai, analizuodami karkasų asemblerio instrukcijas, aptiko pakete naudojamas C kalbos funkcijas, atskleidžiančias žemo lygio sistemos funkcionalumą. Parodyta kaip vykdant C kalba parašytą kodą išvengiama operacinės sistemos apsaugos mechanizmų. C kalbos funkcijų išnaudojimas yra daugiau darbo reikalaujanti ataka nei dinaminis karkasų įkrovimas. Kadangi nematomos net metodų antraštės, reikia rasti tinkamus parametrus metodams, interpretuoti jų grąžinamas reikšmes, išvengti aplikacijos konteinerio apribojimų.
3. Android ir iOS sistemų saugumo būklė lyginta per aplikacijoms suteikiamų privilegijų kiekius. Remtasi saugumo analitikų atliktais darbais, kurių metu analizuota *iTunes AppStore*

platforma ir jos atitikmuo Android ekosistemoje. Palyginta daugiau nei 1300-ai abiejose platformose egzistuojančių programėlių atitkmenų. Programėlės lygintos pagal prieigą prie vartotojo informacijos prieigą turinčių programavimo sąsajų *(angl. SS-API)* naudojimo kiekį. 73% procentuose iOS platformos produktų naudojama daugiau *SS-API* sąsajų nei Android atitikmenyse. Vidutiniškai vienai iOS aplikacijai tenka po 5.9 *SS-API* panaudojimus, Android - 3.9.

Atlikta atviro kodo aplikacijos *WordPress* analizė leidžia teigti, jog naudojamų sąsajų kiekį veikia ir realizacijos detalių skirtumai. *WordPress* iOS aplikacijoje prieš atliekant veiksmus, kuriems reikalingas interneto ryšys yra tikrinama prieiga prie tinklo, Android aplikacijoje tai nėra daroma. Taip pat Android sistemoje instaliavimo metu matomos visos aplikacijai sistemos suteiktos teisės, tad programų kūrėjai stengiasi išvengti nepagrįsto privilegijų naudojimo.

4. Prieš iOS ekosistemą įvykdytos atakos [EKJ+13; T W13], įrodo, jog dabar Apple taikomi apsaugos mechanizmai ir tinkamumo tikrinimo procesas turi silpnybių. Tam, kad sumažinti naujų sistemos spragų išnaudojimo atvejų kiekį, turi būti įgyvendinti saugumo mechanizmų patobulinimai.
    - *Tinkamumo tikrinimo proceso gerinimas.* Statinė analizė atliekama dabartinio tinkamumo proceso metu aptinka privačių metodų kvietimą, kurie naudojami iš statiškai įkelto karkaso. Tačiau statinė analizė pilnai neužtikrina aplikacijos atliekamų veiksmų legitimumo [T W13]. Programos gali naudoti lokacija paremtus saugiklius, kurie išjungia potencialiai kenksmingą funkcionalumą esant tam tikroje vietoje [Isa17]. Taip pat naudojama praktika kuri vykdo kenksmingą kodą tik naudotojui atlikus tam tikrą veiksmų seką (paspaudus mygtukus, praleidus tam tikrą laiko kiekį aplikacijoje). Tokių produktų patekimo į programėlių platinimo platformą *AppStore* uždraudimui gali būti pasitelktos atsitiktinio testavimo metodologijos. Naudojant tokį testavimo būdą aplikacija išbandoma atliekant nenumatytas veiksmų sekas, padengiančius visas galimas atlikti sekas. Tokio testavimo atlikimo užimama trukmė tiesiogiai susijusi su aplikacijos suteikiamu funkcionalumo kiekiu, tad ši metodologija galėtų būti taikoma atsitiktiniams tikrinamiems produktams.
    - *Dinaminis parametrų tikrinimas* Paprasčiausias būdas apsisaugoti nuo pateiktų atakos vektorių yra uždrausti naudoti dinaminės įkrovos metodus *[NSBundle load]* ir *dlopen()* aplikacijų kode. Vis dėlto šis kūrimo paketas naudojamas ir nekenksmingiems tikslams - bibliotekų įkrovai, aplikacijos duomenų pasiekimui, tad toks siūlymas problemą išspręstų panaikindamas svarbų sistemos funkcionalumą.

      Kadangi yra žinoma, jog šie metodai naudojami privačių karkasų naudojimui, tikrinant paduodamų parametrų reikšmes ir aptikus tam tikras sistemos lokacijas, sustabdant metodų įvykdymą, būtų išvengta privačių karkasų įkrovos. iOS sistemoje privatūs ir vieši karkasai yra talpinami */System/Library/* aplankale. Parametre aptikus tokią reikšmę metodo įvykdymas turėtų būti sustabdomas, jei kviečiantysis procesas turi netinkamas veikimo privilegijų teises.

Tačiau atmintyje įkrautiems karkasams įmanoma pritaikyti apgrąžos inžinerijos procesą. Tam pasitelkiama įrenginio atminties analizė, ieškant asemblerio instrukcijų priklausančių karkasui, kuris vykdo reikiamą privatų funkcionalumą. Taip išvengiama aprašyto dinaminio krovimo metodų parametrų reikšmių metodų tikrinimo, nes karkaso kodas pasiekiamas tiesiogiai.

Pasiūlytas prametrų reikšmių tikrinimo sprendimas visiškai neišsprendžia dinaminio naudojimo problemos, tačiau ją žymiai apsunkina.

- *Tarp-procesinės komunikacijos verifikacija.* Antrasis siūlomas aplikacijų konteinerio patobulinimas - dinaminis *SS-API* sąsajas kviečiančiųjų procesų privilegijų tikrinimas. Pavyzdžiui, žemomis privilegijomis veikiantys procesai neturėtų gauti prieigos prie 4.2 poskyryje „iOS sistemos atakos išnaudojant privačias kalbos funkcijas“ minimos privačios sąsajos *MKBKeyBagChangeSystemSecret* metodų. Tik procesams veikiantiems aukštu privilegijų lygi turi būti leidžiama vykdyti privačius metodus. Tačiau, tiesioginis tokio tipo metodų kvietimo uždraudimas problemos neišspręstų. Saugumo analitikų atliktos analizės metu [EKJ+13; T W13] atrasta, jog privačios programavimo sąsajos naudoja *Mach* branduolio suteikiamus tarp-procesinės komunikacijos metodus *(angl. inter-process communication (IPC))* informacijos perdavimui sisteminiams servisams, kurie įgyvendina reikiamą funkcionalumą. Minėtoji *MKBKeyBagChangeSystemSecret* sąsaja komunikacijai su sisteminiu servisu naudoja *mach_msg_send()* metodą, parametruose perduodant reikiamą informaciją. Tai reiškia, jog vietoj tiesioginio privačių sąsajų metodų kvietimo trečiųjų šalių aplikacijos gali pasitelkti *IPC* metodus, jog pasiektų tokį pat funkcionalumą.

  Vienas iš galimų apsisaugojimo būdų yra pačiam žinutės gavėjui (šiuo atveju sisteminiam servisui), patikrinti ar siuntėjas (pvz. *MKBKeyBagChangeSystemSecret* sąsajos metodus kviečianti aplikacija) turi teisę naudoti gavėjo suteikiamą funkcionalumą. Sistema turėtų tikrinti kviečiamo metodo įvykdymui reikalingas privilegijas ir jas sulyginti su siuntėjo privilegijomis. Jei siuntėjas kviečia aukštesniu privilegijų lygiu veikiantį procesą, darbas nutraukiamas.

  Įgyvendinus šį sprendimą, sistemos apsauga nuo privačių sąsajų naudojimo nukeliama į žemesnį lygmenį nei dinaminio parametrų tikrinimo metodo atveju. Sukeliami papildomi sunkumai įgyvendinant sistemos atakas: nepakanka turėti apgrąžos proceso metu gauto karkasų asemblerio kodo, tektų modifikuoti *Mach* kernelyje esančių tarp-procesinės komunikacijos metodų veikimą, kas nėra įmanoma, neturint *root* teisių prieigos įrenginyje.

Literatūra

[App15] Apple. About the security content of iOS 7. https://support.apple.com/en-us/HT202816. tikrinta 2017-04-30. 2015.

[App16] Apple. *Apple Developer: Xcode, Apple's integrated development environment for creating apps for Mac and iOS*. Apple, 2016.

[App17a] Apple. API Reference: NSInvocation class. https://developer.apple.com/reference/foundation/nsinvocation. tikrinta 2017-04-30. 2017.

[App17b] Apple. iOS Security. https://www.apple.com/business/docs/iOS_Security_Guide.pdf. tikrinta 2017-05-10. 2017.

[App17c] Net Applications. Mobile/Tablet Operating System Market Share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1. tikrinta 2017-05-01. 2017.

[Bra01] Ann Brashares. *Steve Jobs Thinks Different*. Twenty-First Century Books, 2001.

[CGC12] Jonathan Crussell, Clint Gibler ir Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. Springer, Berlin, Heidelberg, 2012-09. URL: https://link.springer.com/chapter/10.1007/978-3-642-33167-1_3.

[Cha17] Dave Chaffey. Mobile Marketing Statistics compilation. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/. tikrinta 2017-05-10. 2017.

[Cra95] Don Crabb. Copland, Where Are Thee? *Mactech.com*, 1995.

[CRK05] Jonathan Corbet, Alessandro Rubini ir Greg Kroah-Hartman. *Linux Device Drivers: Third Edition*. O'Reilly Media, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2005.

[Die95] Tim Dierks. Copland: The Mac OS Moves Into the Future. *Mactech.com*, 1995.

[EKJ+13] Mohammad Etemad, Alptekin Küpçü, Michael Jacobson, Michael Locasto, Payman Mohassel ir Reihaneh Safavi-Naini. *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings. Lecture Notes in Computer Science 7954 Security and Cryptology*. Springer-Verlag Berlin Heidelberg, 2013, p. 272–289.

[EKK+11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the network and distributed system security symposium*. San Diego, CA, 2011.

[Ess11] Stefan Esser. Exploiting the iOS Kernel, 2011.

[FHE+12] A. P. Felt, E. Ha, S. Egelman, A. Haneya, E. Chin ir D. Wagner. Android permissions: User attention, comprehension, and behavior. *In proceedings of the symposium on usable privacy and security (soups)*. ACM, New York, 2012.

[For16] Conner Forrest. The state of mobile device security: Android vs. iOS. http://www.zdnet.com/article/the-state-of-mobile-device-security-android-vs-ios/. July 11, 2016.

[Fra96] Tony Francis. *Mac OS 8 Revealed*. Addison-Wesley Developers Press, 1996. 18 - 32 p.

[Fru14] Andrei Frumusanu. A Closer Look at Android RunTime (ART) in Android L. *Anandtech*, July 1, 2014. URL: http://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/.

[Goo17] Google. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission.html. tikrinta 2017-05-03. 2017.

[HC11] Henry Halvorsen ir Doug Clarke. *OS X and iOS Kernel Programming*. Apress, 2011. 15-38 p.

[Hel16] Michael Heller. Pegasus iOS exploit uses three zero days to attack high-value targets. *Techtarget: search security*, 2016.

[Her05] Andy Hertzfeld. *Revolution in the Valley*. O'Reilly Media, Inc., Sebastopol, CA, 2005.

[Her10] Andy Hertzfeld. The Original Macintosh: Mea Culpa. *Folklore.org*, 2010.

[HHW+12] S. Hanna, L. Huang, E. Wu, C. Chen S. Li ir D. Song. Juxtapp: A scalable system for detecting code reuse among Android applications. *Proceedings of the ninth conference on detection of intrusions and malware and vulnerability assessment (dimva'12)*. Springer-Verlag, Berlin, 2012, p. 62–81.

[HYG+13] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng. Comparing mobile privacy protection through cross-platform applications. In *In proceedings of the 20th annual network & distributed system security symposium*. San Diego, CA, 2013.

[Hof15] Chris Hoffman. Why iPhones Are More Secure Than Android Phones. https://www.howtogeek.com/224096/why-iphones-are-more-secure-than-android-phones/. tikrinta 2017-05-01. 2015.

[Hor12] Tom Hormby. The Rise and Fall of Apple's Gil Amelio. *Lowendmac*, 2012.

[Ing16] Nathan Ingraham. Senator confirms FBI paid $900,000 to unlock San Bernardino iPhone. *Engadget.com*, 2016.

[Isa17] Mike Isaac. Uber's C.E.O. Plays With Fire. *The new york times*, 2017.

[LLW+12] L. Lu, Z. Li, Z. Wu, W. Lee ir G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. *In proceedings of the 2012 acm conference on computer and communications security (ccs'12)*. ACM, New York, 2012, p. 229–240.

[Loc12] H. Lockheimer. Android and security. Google Mobile Blog, February 2, 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html. tikrinta 2017-05-03. 2012.

[McL16] Jenna McLaughlin. New Court Filing Reveals Apple Faces 12 Other Requests to Break Into Locked iPhones. *Theintercept.com*, 2016.

[MD13] E. Macaskill and G. Dance. NSA FILES: Decoded. *Theguardian.com*, 2013.

[Nol12] G. Nolan. *Decompiling Android*. Apress, New York, 2012.

[Sea01] D. Seal. *ARM Architecture Reference Manual*. Pearson Education, 2001.

[Sei10] Chris Seibold. Mac OS 9 Release. *Apple Matters*, 2010.

[Ser17a] N. Seriot. iOS Runtime headers. `https://github.com/nst/iOS-Runtime-Headers`. tikrinta 2017-04-30. 2017.

[Ser17b] N. Seriot. Objective-C runtime browser, for Mac OS X and iOS. `https://github.com/nst/RuntimeBrowser/`. tikrinta 2017-04-30. 2017.

[She17] Satish Shetty. Enterprise Android Vs iOS: Which is More Secure? `http://www.darkreading.com/mobile/enterprise-android-vs-ios-which-is-more-secure/a/d-id/1328068`. tikrinta 2017-05-01. 2017.

[Sin03] Amit Singh. *Mac OS X Internals: A Systems Approach*. Springer, 2003.

[Sor10] C. Sorrel. Apple approves, pulls flashlight app with hidden tethering mode. `http://www.wired.com/gadgetlab/2010/07/apple-approvespulls-flashlight-app-with-hidden-tethering-mode`. tikrinta 2017-05-03. 2010.

[T W13] T. Wang and K. Lu and L. Lu and S. Chung and W. Lee. Jekyll on ios: when benign apps become evil. In *Proceedings of the 22nd usenix security symposium (usenix security)*. USENIX Association, Berkeley, CA, 2013, pp. 559–572.

[VVC11] Timothy Vidas, Daniel Votipka ir Nicolas Christin. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. *Proceedings of the 5th usenix conference on offensive technologies*. WOOT'11. USENIX Association, San Francisco, CA, 2011, p. 1–10.

[Wal05] Isaacson Walter. *Steve Jobs*. Simon ir Schuster, 2005. 227 psl.

[Wat08] Robert Watson. Re: freebsd-advocacy Digest, Vol 248, Issue 1. `https://lists.freebsd.org/pipermail/freebsd-advocacy/2008-August/003674.html`. August 2, 2008.

[Wat09] Robert Watson. Grand Central Dispatch - FreeBSD port. `https://www.freebsd.org/news/status/report-2009-04-2009-09.html`. April, 2009.

[Wid08] Jake Widman. Lessons Learned: IT's Biggest Project Failures. *Computerworld*, 2008.

[ZZD+12] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han ir W. Zou. SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications. *In proceedings of the second acm workshop on security and privacy in smartphones and mobile devices (spsm'12)*. ACM, New York, 2012, p. 93–104.

[ZZJ+12] W. Zhou, Y. Zhou, X. Jiang ir P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. *In proceedings of the second acm conference on data and application security and privacy (codaspy'12)*. ACM, San Francisco, CA, 2012, p. 317–326.