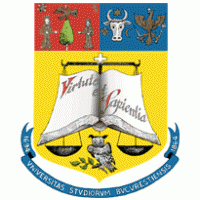
UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

****

**LUCRARE DE LICENȚĂ**

Studiu în exploatarea unui bug

|  |  |
| --- | --- |
| Profesor îndrumător,  *Lector Dr. Paul Irofti* | Student,  *Călin Marian Gligore* |

**București**

**2020**

**Rezumat**

În această lucrare vom discuta despre securitatea sistemului de operare iOS, exploatarea bug-urilor software și despre mitgațiile software și hardware aduse de Apple. Bug-ul pe care îl vom exploata, cunoscut ca checkm8, este un bug în software-ul scris în ROM-ul tuturor iPhone-urilor pană la iPhone 8. Deși un exploit la acest nivel ar trebui să fie suficient de puternic, vom afla că modelul de securitate Apple, compus din mitigații și chiar și faptul că majoritatea codului lor sursă nu este public, complică foarte mult munca unui atacator. De exemplu, pentru a explora codul aplicațiilor va trebui să folosim unelte de inginerie inversă, precum Ghidra sau IDA. Scopul nostru final va fi să oferim unei aplicații ce rulează în userland drept de root, să o scoatem din sandbox și să îi oferim posibilitatea de scriere și citire în memoria kernel-ului și a celorlalte aplicații.

**Cuprins**

[1. Introducere 4](#_gjdgxs)

[2. Preliminarii 5](#_30j0zll)

[2.1. Securitatea iOS 5](#_1fob9te)

[2.2. Protocolul USB (pentru transferuri control) 7](#_3znysh7)

[2.3. Protocolul DFU 8](#_2et92p0)

[3. Bug-ul 9](#_tyjcwt)

[3.1. Construirea exploit-ului 11](#_3dy6vkm)

[3.2. Construirea payload-ului 18](#_1t3h5sf)

[3.3.1 Modificarea iBoot-ului 20](#_4d34og8)

[3.3.2 Modificarea kernel-ului 24](#_2s8eyo1)

[3.4. Aplicație proof-of-concept 31](#_17dp8vu)

[4. Concluzii 34](#_3rdcrjn)

[5. Bibliografie 36](#_26in1rg)

# 1. Introducere

Domeniile în care se încadrează lucrarea sunt cel al ingineriei inverse și cel al securității software. În această lucrare vom discuta despre o parte din măsurile de protecție din dispozitivele noastre mobile. Am ales această temă deoarece sunt interesat de ingineria inversă și de libertatea consumatorului de a-și folosi bunurile.

În prezent, direcția în care Apple și Google se duc este una anti-consumator, AppStore fiind singurul loc din care se pot instala aplicații,  iar în unele țări precum China, multe aplicații importante sunt direct cenzurate de aceste companii. Mai mult, dezvoltatorii trebuie să își distribuie aplicațiile doar prin intermediul acestor platforme și sunt obligați să plătească 30% din profit companiilor Apple sau Google. Toate aceste limitări sunt impuse chiar de software-ul menit să protejeze consumatorul. Spre exemplu, pe iOS toate aplicațiile trebuie să fie semnate de Apple. Consumatorii pot instala pe telefon aplicații scrise de ei, dar acestea expiră într-o săptămână și nu pot avea mai mult de trei simultan pe telefon. Fiecare aplicație trebuie să declare drepturile speciale cu care vrea să execute (entitlements); o mare parte dintre acestea nu pot fi folosite de dezvoltatori independenți, iar unele nici de aplicațiile distribuite de AppStore. De exemplu, pentru a scrie o aplicație VPN ai nevoie de un certificat în valoare de 100 de dolari care trebuie cumpărat de la Apple.

Prin urmare, scopul acestui proiect este acela de a oferi posibilitatea unei aplicații scrise de un dezvoltator independent să evite toate aceste limitări și chiar să poată face lucruri care nu sunt deloc posibile în sandabox-ul impus de Apple, precum să vadă ce procese rulează sau să acceseze toate fișierele de pe telefon. Va fi practic un “Run as Administrator” din Windows sau “sudo” din Linux.

În capitolul 2 voi prezenta noțiunile cu care trebuie să fim familiari înainte de a exploata bug-ul ales și vom discuta despre securitatea iOS, de la bootrom până la kernel-ul lui, Darwin, un kernel UNIX-like bazat pe BSD și Mach.

În capitolul 3 vom construi pe baza bug-ului un exploit ce ne va permite să rulăm aplicații în userland ce pot accesa memoria kernel-ului și a altor aplicații.

# 2. Preliminarii

## 2.1. Securitatea iOS

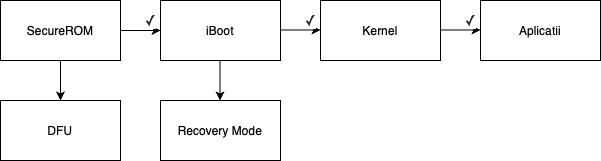
În iOS securitatea se bazează pe faptul că toate aplicațiile sunt verificate și semnate și rulează într-un sandbox care le permite să acceseze doar datele din container-ul lor. Cu ajutorul drepturilor (entitlements) pe care o aplicație le poate cere, aceasta poate face lucruri precum accesarea contactelor, a pozelor sau să mapeze memorie ca RWX pentru cod JIT (Just-In-Time compiled). Multe dintre aceste drepturi, considerate de Apple a fi periculoase sunt păstrate doar pentru aplicațiile lor, precum “com.apple.security.cd.allow-jit”, Safari fiind singura aplicație semnată cu un astfel de drept (desigur acest lucru face ca browserele competiției să fie mai lente). Aplicația și drepturile cu care aceasta are voie să ruleze sunt semnate împreună, deci o aplicație nu are voie să își schimbe lista cu drepturi după ce aceasta a fost deja semnată de Apple. Semnăturile sunt verificate în userland de către AMFId (AppleMobileFileIntegrity daemon), nu de către kernel direct. Acesta rulează în background și verifică că toate aplicațiile lansate au fost semnate de Apple și că nu realizează acțiuni nepermise de lista lor de drepturi. AMFId limitează și accesul la anumite syscall-uri (partea de BSD) și mach traps (practic syscall-uri din partea de Mach a kernel-ului).

În concluzie, în userland toate aplicațiile trebuie să treacă peste o serie de verificări conduse de kernel. Dacă dorim o aplicație care să execute mai mult decât îi este permis va trebui să trecem de acestea. Îl putem oare modifica? Printre primele lucruri pe care kernel-ul le face este să își mapeze zona de TEXT ca read-execute. Instrucțiunile care ar putea schimba asta trebuie executate în kernel mode și se află într-o pagină nemapată, deci din care nu se poate executa. Mai mult, telefoanele apărute după iPhone 7 vin cu un coprocesor supranumit KTRR (Kernel Test Readonly Region) care se asigură că și în cazul în care reușim să mapăm o pagină ca executabilă, nu vom putea da fetch instrucțiunilor din ea. Deci, tot ce putem modifica în kernel a rămas zona de date (neconstante) care este mapată ca read-write. Desigur, pentru a ajunge acolo este nevoie să exploatăm un bug în codul scris de inginerii de la Apple. Bug-ul pe care îl vom exploata se află mai jos decât kernel-ul (într-o bucată de cod mai privilegiată), de unde vom putea modifica kernel-ul fără astfel de limitări.

Deci, să coborâm la următorul nivel: iBoot. iBoot-ul este un bootloader care are ca scop căutarea kernel-ului pe disc, verificarea și încărcarea lui. După ce acesta a trecut toate verificările, sare în entry point-ul kernel-ului. Observăm că la fiecare pas componenta nou încărcată este verificată să fie semnată de Apple. Chiar și dacă am modifica fișierul în care se află kernel-ul sau iBoot-ul, nu vom putea boota.

Mai jos decât iBoot se află bootrom-ul, denumit de Apple SecureROM. El este însărcinat cu găsirea, verificarea și încărcarea iBoot-ului. Aici se află bug-ul despre care urmează să vorbim. Ce este important la această bucată de cod este faptul că este arsă în chip: aceasta nu mai poate fi modificată (să primeasca update-uri) din momentul în care chip-ul a ieșit din fabrică. La toate celelalte niveluri, codul poate primi update-uri și primește astfel de modificări de fiecare dată când un bug este descoperit (sau se adaugă un nou feature sistemului de operare). Aproape fiecare versiune de iOS rezolvă astfel de bug-uri, de aceea este important să avem instalată mereu cea mai recentă versiune care ne este disponibilă: astfel de bug-uri pot fi exploatate și de persoane rău intenționate, nu doar local, de către noi. În cazul SecureROM-ului lucrurile stau altfel: un device va rămâne vulnerabil la bug-urile din acest nivel pentru totdeauna.

Acest model, în care fiecare componentă încarcă și verifică următoarea component, se numește chain of trust. Un bug într-una din verigi compromite lanțul de la ea încolo. SecureROM-ul, care este un iBoot cu multe feature-uri dezactivate la compilare, este cea mai trusted componentă din lanț. Ea încarcă iBoot-ul, mai complex decât SecureROM-ul, adăugând feature-uri precum Recovery Mode, configurând RAM-ul și, mai ales, desfășurând diferite operații complexe precum parsare și decomprimare de fișiere. Într-un cod complex șansele să apară bug-uri în scrierea lui cresc, deci se dovedește necesară posibilitatea de a updata o astfel de componentă. De aceea, deși iBoot-ul putea fi pus direct în ROM, a fost pus a doua verigă în chain of trust-ul nostru. iBoot parsează fișiere complexe precum “deviceTree”-ul unde se află argumente pentru următoarea componentă, kernel-ul. Dacă nu poate să găsească pe disc kernel-ul sau nu îl poate verifica, atunci intră în Recovery Mode de unde așteaptă încărcarea unui nou firmware. Kernel-ul inițializează datele care vor deveni constante, activează KTRR și pornește launchd, un daemon care se ocupă de lansarea celorlalte procese din userland.



*Figura 2.1 Chain of trust*

Diagrama precedentă arată chain of trust-ul care stă la baza securității iOS ce are ca scop final rularea de aplicații semnate doar de Apple și doar în sandbox, astfel orice cod ar semna Apple, acesta tot nu poate produce daune prea mari (într-un sistem perfect, lipsit de bug-uri). Putem observa că acest lanț are și un sistem failsafe, prin care device-ul se poate întoarce la o stare normală după introducerea de erori: DFU și Recovery Mode.

DFU (Device Firmware Update) este un mod prin care un dispozitiv se poate întoarce la o stare verificată criptografic, orice s-ar intampla. Dacă posesorul device-ului crede că a fost victima unui atac, el poate introduce telefonul în DFU de unde știe că singurul cod care rulează este cel scris și verificat de Apple. Pentru a intra în DFU, trebuie să efectuăm o serie de apăsări de butoane. Deci se poate intra doar având acces fizic la device. În DFU se află bug-ul despre care vom discuta. Mai exact, în implementarea protocolului USB de care se folosește pentru uploadarea firmware-ului.

## 2.2. Protocolul USB (pentru transferuri control)

În acest capitol vom vorbi despre transferurile control (control transfers) care stau la baza DFU. Un astfel de transfer este compus din trei stagii: etapa de setup, etapa de transfer date și etapa de status.

În etapa de setup, gazda trimite un token setup urmat de un pachet care conține informații despre transferul ce urmează să aibă loc (setup packet). El conține informații despre direcția transferului de date, request-ul pe care-l face și alte valori interpretate în funcție de request-ul care are loc.

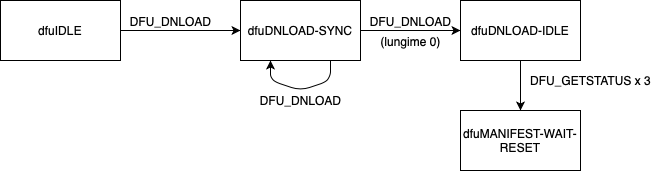
Etapa de transfer de date este compusă din mai multe tranzacții asemănătoare celor de la etapa de setup. Gazda trimite un token IN sau OUT, în funcție de direcția transferului, după care primește sau trimite un pachet de date. Acest lucru se repetă până când toate datele, a căror lungime a fost declarată în setup, au fost transferate.

Etapa de status se află la finalul transferului. Ea are ca scop verificarea statusului transferului pentru a ști dacă este nevoie de repetarea acestuia sau dacă s-a efectuat cu succes. În cazul unui transfer de tip IN, în care gazda a primit date de la device, gazda trimite un token OUT și un pachet de lungime 0, marcând astfel finalizarea transferului. În cazul unui transfer de tip OUT, gazda trimite un pachet IN și așteaptă primirea de la device a unui pachet de lungime 0. În locul pachetelor de lungime 0 se pot elibera pachete NAK sau STALL. Pachetele NAK indică faptul că request-ul trebuie repetat, iar pachetele STALL indică faptul că în procesarea request-ului s-a produs o eroare.

## 2.3. Protocolul DFU

Protocolul DFU poate fi văzut ca un automat finit [1]. Acesta începe în starea dfuIDLE. După ce device-ul primește un transfer control cu request-ul DFU\_DNLOAD, acesta trece în starea dfuDNLOAD-SYNC. Cât timp gazda mai are date de trimis, o să continue să îi trimită request-uri de tip DFU\_DNLOAD. După ce a terminat, gazda trimite un ultim request DFU\_DNLOAD, de data asta urmat de date de lungime 0.

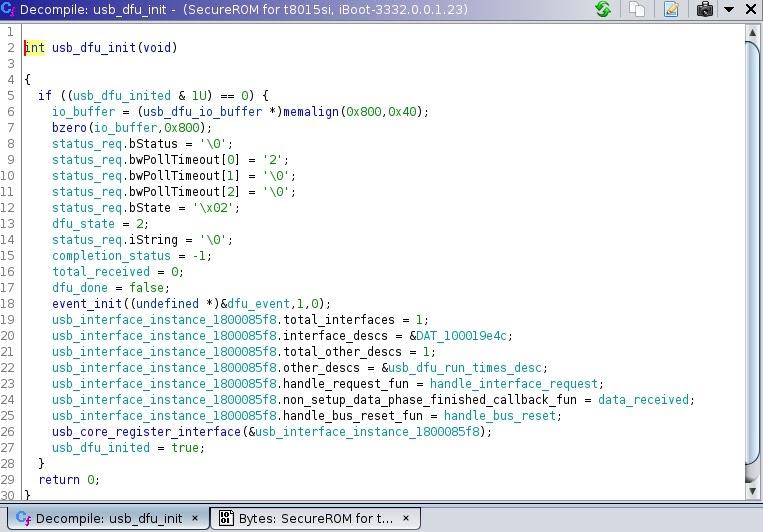
În acest moment, dacă gazda trimite un request de tip GET\_STATUS, automatul va intra în starea dfuMANIFEST-SYNC. După alte două request-uri GET\_STATUS, automatul va intra în starea dfuMANIFEST-WAIT-RESET. În acest moment, dacă dispozitivul primește un bus reset, va începe parsarea imaginii primite.



*Figura 2.2 Automat simplificat ilustrând modul de operare al DFU over USB în cazul Apple*

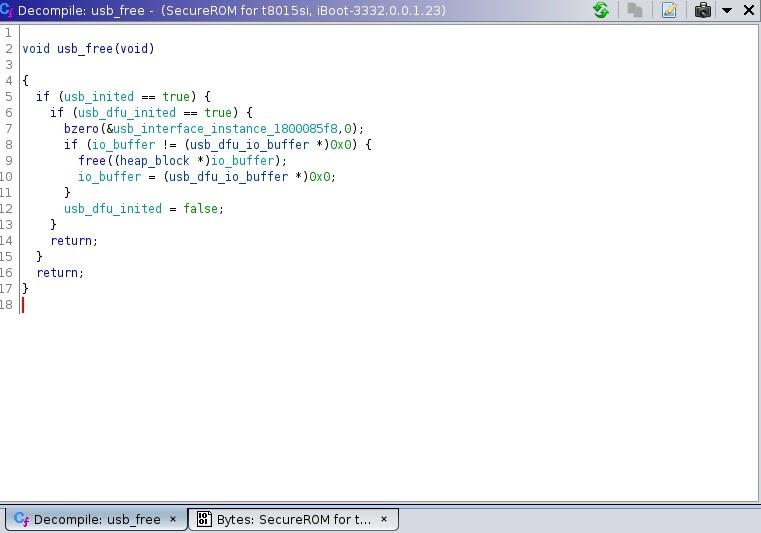
# 3. Bug-ul

Când un iPhone intră în DFU mode, acesta pornește codul principal (usb\_core). El alocă diferite stringuri folosite de descriptorii usb, utilizați de gazdă pentru a recunoaște dispozitivul și a afla diferite informații despre el. După aceea, pornește interfața DFU. Ea alocă o zonă de memorie de 0x800 bytes și ține un pointer către această zonă într-o variabilă globală (aici numită io\_buffer). Tot într-o variabilă globală ține minte și numărul total de bytes primit, pe care în funcția de inițializare îl va seta la 0. Codul sursă pentru SecureROM nu este public, însă putem decompila și analiza un dump al imaginii cu aplicații precum IDA sau Ghidra. În următoarea imagine putem vedea codul care inițializează interfața DFU chemat de usb\_core. Am redenumit variabilele și am recreat structurile pentru a face codul mai ușor de înțeles și mai aproape de codul sursă.



*Figura 3.1 Codul pentru inițializarea interfeței DFU*

Setup-urile transferurilor control emise către interfața DFU sunt tratate de către ea. Însă etapa de date este tratată de către core. Pentru asta, interfața DFU întoarce către core pointer-ul către zona alocată în init, pe care usb\_core îl va memora în altă variabilă globală (data\_phase\_buffer). Deci, dacă emitem un request DFU\_DNLOAD, pointer-ul către zona alocată se va afla în două variabile globale. Dacă toate datele au fost trimise, atunci usb\_core resetează la NULL data\_phase\_buffer în care ținea minte pointer-ul către zona alocată și notifică interfața DFU că transferul s-a terminat. După aceasta, gazda poate emite un usb reset pentru a începe procesarea imaginii. În acest moment, usb\_core resetează mai multe variabile globale, mai puțin data\_phase\_buffer, și notifică interfața DFU că un usb reset a fost emis. Aceasta va dezaloca zona de memorie și va reseta io\_buffer la NULL. Dispozitivul va verifica imaginea descărcată, iar dacă acest lucru nu se întâmplă cu succes, atunci dispozitivul pornește din nou usb core-ul de unde așteaptă încărcarea unei noi imagini.



*Figura 3.2 Codul apelat la închiderea interfeței DFU*

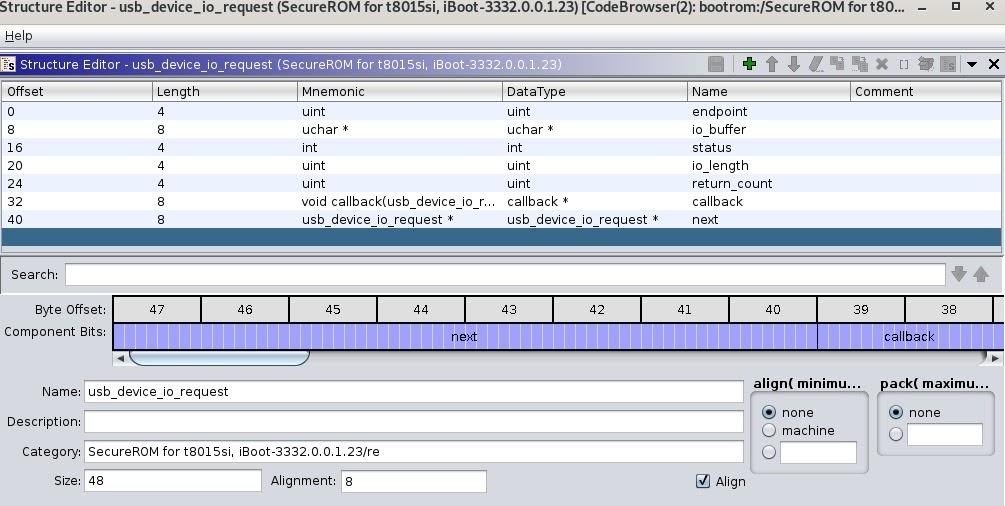
Prin urmare, dacă emitem un usb reset înainte să terminăm transferul de date, interfața DFU va aloca o nouă zonă de memorie, dar usb core-ul va avea încă un pointer la vechea zonă de memorie: asta pentru că această variabilă este setată doar la venirea unui setup de tip DFU\_DNLOAD și resetată doar dacă transferul s-a terminat. Avem, astfel, un bug de tip Use-After-Free. Acest lucru ne deschide posibilitatea de a scrie în heap peste alte variabile. Urmează să aflăm cum putem folosi acest lucru pentru a modifica kernel-ul astfel încât să ne permită rularea de aplicații cu drepturi de root și fără limitele impuse de sandbox. Bug-ul acesta a fost descoperit independent de mai multe persoane și, fiind un bug rar, foarte puternic și care afectează multe dispozitive încă menținute și chiar vândute de către Apple, a primit și un nume: “checkm8” (șah mat) [2][3].

## 3.1. Construirea exploit-ului

Scopul nostru e să ajungem în punctul în care kernel-ul a fost încărcat în memorie pentru a-l putea modifica. Pentru asta, este necesar să să trecem de SecureROM și iBoot.

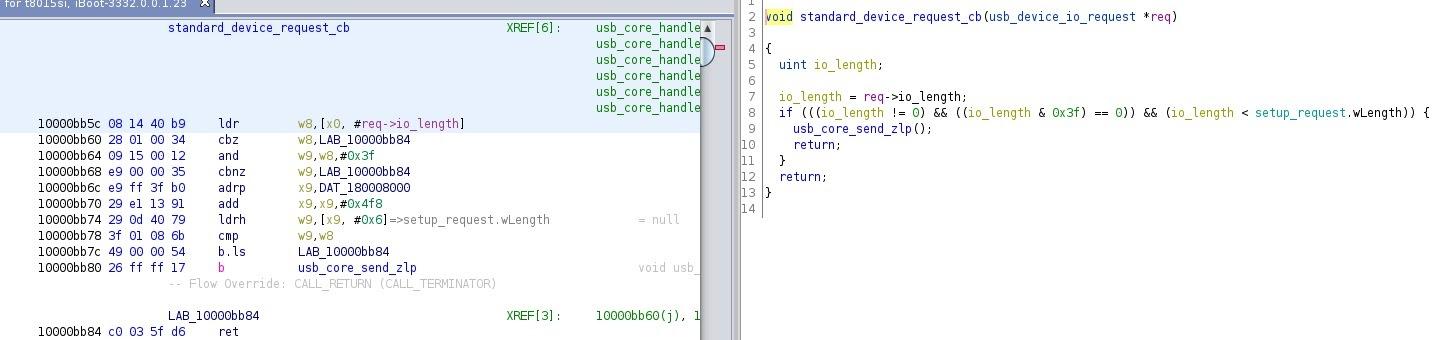
Pentru a utiliza Use-After-Free-ul trebuie doar să emitem un setup DFU\_DNLOAD, să emitem un setup DFU\_ABORT, să resetăm conexiunea și să continuăm transferul de date. Datele transmise vor fi puse în heap la zona pointată de variabila globală data\_phase\_buffer. Avem însă o problema: SecureROM-ul fiind o aplicație nu foarte complexă, alocațiile sunt mereu aceleași. Prin urmare, zona care se alocă pentru io\_buffer va fi mereu aceeași. Avem nevoie, deci, de un alt bug: un memory leak. Trebuie să găsim în cod un mod de a aloca memorie care nu va fi dezalocată. Astfel, noile alocări pentru io\_buffer vor fi diferite. Este necesar acest lucru pentru a putea scrie în heap peste alte date alocate acolo.

Fiecare request către usb are asociată această structură:



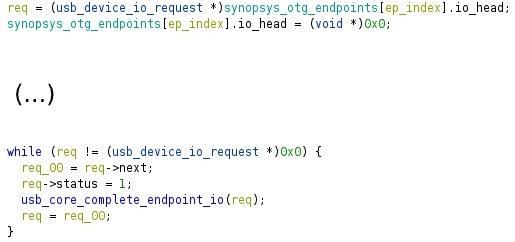
*Figura 3.3 Structura usb\_device\_io\_request*

Când un request s-a finalizat, usb\_core cheamă funcția din variabila “callback”. Aici, de obicei, se află o funcție care determină dacă este nevoie să trimită gazdei un pachet de lungime 0 pentru a marca finalizarea tranzacției. Callback-ul acesta arată astfel:

****

*Figura 3.4 Funcția de callback*

Funcția usb\_core\_send\_zlp creează un nou request și îl pune într-o variabilă globală “io\_head”, în care sunt ținute toate request-urile.

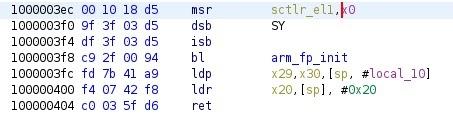
Request-urile mai pot fi completate și dacă se emite un bus reset. În acel moment, callback-ul tuturor request-urilor este apelat. Putem observa aici codul care se ocupă de asta:****

*Figura 3.5 Bucla care finalizează toate request-urile*

Toate request-urile, din acest moment, puse în io\_head nu vor mai putea fi finalizate. Prin urmare, din momentul în care bus-ul se resetează, toate requesturile care îndeplineau condițiile pentru a trimite un pachet de lungime 0 vor lăsa în heap câte o structură de 48 de biți care nu va mai fi eliberată.

În concluzie, acum putem scrie în heap peste alte date. Rămâne să ne alegem “victima” peste care să scriem. O structură de usb\_device\_io\_request este un candidat foarte bun: ținem minte că aceasta are un câmp “callback”. Putem scrie peste ea o altă adresă de la care să execute. Adresele de la care putem executa trebuie să îndeplinească însă o condiție: să se afle într-o zonă mapată ca executabilă. Asta pentru că MMU (Memory Management Unit) este deja pornită de mult timp. Bitul 0 din registrul SCTLR spune platformei dacă MMU este activată. Înainte ca ScerureROM să pornească usb-core-ul pentru descărcarea unei imagini, el inițializează platforma: pornește watchdog-ul, inițializează tabelele de translație la adresa 0x18000C000, scrie această adresă în registrul TTBR0 și scrie în regstrul SCTLR valoarea 0x8100D. Asta activează o serie de feature-uri ale procesorului printre care se numără MMU și protecția WXN. Deci, trebuie să găsim bucăți de cod care să facă ce vrem noi și să fie urmate de o instrucțiune “return”. Aceste bucați de cod poartă denumirea de gadget-uri. Este foarte complicat să scriem un cod prea complex în acest mod. Așa că ne rezumăm la a găsi o serie de instrucțiuni care să ne permită să mapăm zona în care se încarcă firmware-ul primit prin usb, ca RWX și apoi să sară acolo. Pentru a realiza acest lucru avem nevoie de următoarele gadget-uri:

* Un gadget cu care să scriem în registrul SCTLR - avem nevoie de acesta pentru a reseta bitul 19 la 0. El se mai numește și WXN (Write-Execute-Never). Acest registru indică dacă se pot executa instrucțiuni din paginile mapate ca write chiar dacă au fost mapate și ca executabile [4]. Cât timp bit-ul 19 este setat și MMU este pornită, paginile WX sunt văzute doar ca writeable. Avem un astfel de gadget la adresa 0x1000003EC:



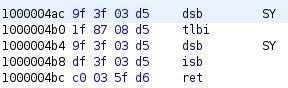
*Figura 3.6 write\_sctlr(arg)*

* Un gadget cu care să scriem în registrul TTBR. În acest registru se află un pointer către tabelele de translație. Trebuie să îl modificăm pentru a putea remapa temporar SRAM-ul ca RW(X). Nu putem scrie direct peste intrările din această tabelă deoarece acestea nu sunt mapate în memorie. Avem un astfel de gadget la adresa 0x10000045C:

https://lh4.googleusercontent.com/oHwyQ8swK2M9M9jMLlN6RqMs5JGv70cvOFuBEM3A6_olBy8NV9WgmROF02XJV6uA4d-fgQkMU33s0zmTlWK4yIwlDn4hFpnPnqqMNDfbIVogzSJZxw5JPIviwj3nx1qAiCyah39G

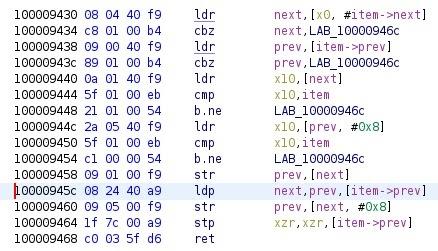
*Figura 3.7 write\_ttbr0(arg)*

* Un gadget cu care să invalidăm intrările din tabela de translație. Acesta ne asigură că modificările pe care le vom face tabelei de translație vor fi vizibile procesorului. Avem un astfel de gadget la adresa 0x1000004AC:

****

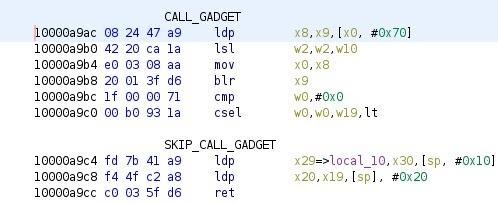
*Figura 3.8 tlbi()*

* Un gadget cu care să putem scrie la orice adresă dorim. Avem un astfel de cod într-una din funcțiile care lucrează cu liste - în funcția list\_delete de la adresa 0x10000945C:



*Figura 3.9 Gadget Write What Where*

Observăm că nu putem pune pur și simplu aceste adrese în câmpul callback: toate aceste gadget-uri, mai puțin invalidarea tabelelor de translație, au nevoie de un argument. axi0mX [5] a găsit în cod un astfel de gadget la adresa 0x10000A9AC:

****

*Figura 3.10*

Funcția de callback primește ca argument un pointer către request-ul curent. Deci registrul X0 va avea această valoare. Observăm că acest gadget încarcă, în X0 de la adresa request-ului + 0x70, primul argument, iar de la adresă + 0x78, adresa funcției pe care o va apela. Cu ajutorul acestei bucăți de cod foarte importante vom putea apela adrese arbitrare, cu un argument arbitrar. Trebuie doar să avem grijă cum construim acest callback-chain în memorie. La adresa 0x10000A9C4 avem și un gadget pe care îl putem folosi să ne întoarcem la funcția care a apelat funcția care a apelat callback-ul, astfel sărind peste apelarea funcției free. Acesta este un alt lucru foarte important deoarece structurile pe care le vom construi noi în memorie nu au fost scrise într-o zona alocată de memorie. Un free al unei astfel de adrese ar panica si reseta SecureROM-ul.

Acum putem construi un callback-chain, cu ultimul callback adresa payload-ului pe care dorim să-l executăm și valoarea câmpului next egală cu NULL.  Pentru comunicarea cu device-ul vom folosi librăria libusb. Vom împărți exploit-ul în mai multe etape.

1. Crearea unei găuri în heap cu ajutorul leak-urilor de memorie. O astfel de etapă se află în multe exploit-uri ce au de-a face cu bug-uri în managementul zonelor alocate în heap și a primit și un nume: “heap feng shui” [6]. După aceea emitem un bus reset ce va chema funcțiile din câmpul “callback” al fiecărui request. Ținem minte că acest câmp conține adresa funcției “standard\_device\_request\_cb” care alocă, în anumite condiții, un request pentru a trimite un pachet către gazdă de lungime 0. Condițiile care trebuie respectate pentru a aloca această structură sunt:
   1. Lungimea datelor trimise/primite este diferită de 0.
   2. Lungimea datelor se împarte exact la 0x40 (dimensiunea maximă a unui pachet).
   3. Lungimea datelor trimise/primite pâna în acest punct este strict mai mică decât lungimea datelor ultimului request care este ținută într-o variabilă globală (Figura 3.4).

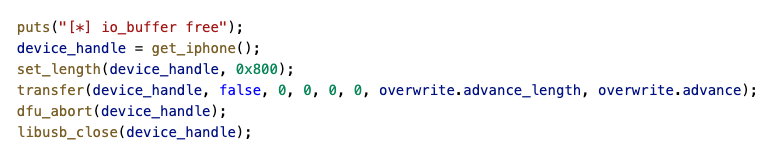
Avem următorul cod:



*Figura 3.11*

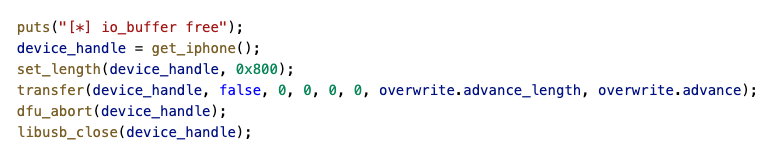
Funcția “stall\_data\_and\_leak” a fost creată cu scopul de a forma o coadă de request-uri. Ea trimite un setup dispozitivului prin care îi cere date, după care întrerupe finalizarea transferului. Telefonul așteaptă primirea token-ului IN, pe care nu îl va mai primi. În acest moment este incapabil să finalizeze transferurile care vor urma, poate doar să proceseze alte pachete setup. Formând o coadă de request-uri, lungimea ultimului request poate fi diferită de lungimea celorlalte. Putem deci să condiționăm leak-urile de memorie. Funcția “stall\_data\_and\_leak” va îndeplini condițiile pentru a lăsa în heap o structură de request care nu va fi eliberată. Funcția “dont\_leak” nu îndeplinește aceste condiții. La final, trimitem un bus reset telefonului. Acest lucru face SecureROM-ul să trateze, în sfârșit, lista de request-uri care s-a format. Rezultatul este o gaură în heap de mărimea (aproximativ) heap\_hole \* sizeof(usb\_device\_io\_request). Acest lucru este suficient să mute o parte din alocări la următoarea inițializare a usb-ului (alocările în SecureROM întorc cea mai mică zonă liberă care satisface dimensiunea cerută).

1. Eliberarea buffer-ului io\_buffer după ce am trimis și întrerupt un transfer DFU\_DNLOAD. Tot aici vom avansa și scrierea în heap, înainte de free, pentru a minimiza daunele provocate heap-ului în următorul stadiu. Device-ul va reinițializa usb-core-ul și va realoca descriptorii, task-urile și celelalte obiecte de care are nevoie. Funcția “set\_length” inițiază un transfer DFU\_DNLOAD cu lungimea specificată, pe care îl vom întrerupe folosind funcția “libusb\_cancel\_transfer”. Avansăm numărul de bytes transmis folosind un transfer cu un setup invalid. În acest fel suntem siguri că buffer-ul în care se vor scrie datele este tot io\_buffer. Trimitem un request DFU\_ABORT care va închide complet stack-ul usb. SecureROM-ul se va întoarce în bucla infintă din care așteaptă încărcarea unei imagini, va detecta faptul că DFU a fost închis cu o eroare (acesta este scopul transferului DFU\_ABORT), așa că stack-ul usb va fi repornit pentru a descărca o nouă imagine.



*Figura 3.12*

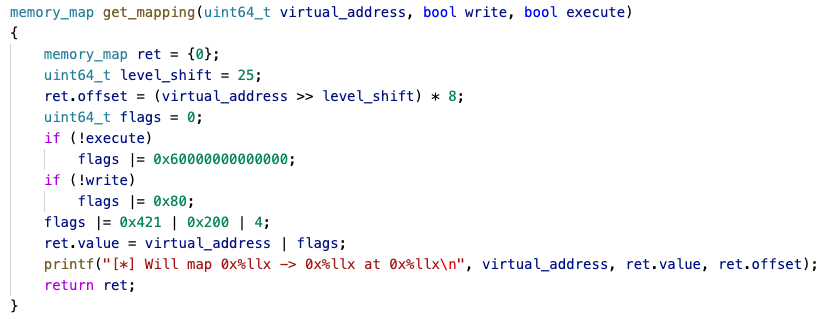
1. Suprascrierea în heap a câmpurilor “callback” și “next” ale unei structuri proaspăt alocate. Am scris funcția “stall\_only” cu scopul de a începe iar o coadă de request-uri; de data asta, pentru a ne asigura că scriem peste câmpurile “callback” și “next” ale unor request-uri vii, care vor fi tratate la primirea unui bus reset. Transferăm datele care vor suprascrie câmpul “callback” cu 0 și câmpul “next” care va pointa acum către adresa INSECURE\_MEMORY + 0x8. Acolo vom încărca un lanț de request-uri construite de noi.



*Figura 3.13*

1. Încărcarea payload-ului. Vom descărca device-ului un buffer cu structuri usb\_device\_io\_request construite de noi. Fiecare câmp “callback” și “next” va pointa către adrese special alese de noi.
2. Inițierea unui bus reset pentru a începe execuția callback-urilor construite de noi și a payload-ului.

Lanțul nostru de callback-uri are ca scop, printre altele, remaparea SRAM-ului și SecureROM-ului ca RWX. Asta pentru că din SRAM vom executa codul nostru și va trebui și să modificăm codul bootrom-ului. Pentru a ști la ce offset în tabela de translație trebuie să scriem și pentru a ști ce trebuie să scriem putem citi documentația ARM pentru arhitectura AARCH64. Putem în schimb să ne uităm la decompilarea codului care se ocupă de inițializarea MMU-ului. Aflăm că ce ne trebuie nouă se reduce la următorul cod:



*Figura 3.14 Emularea intrărilor din tabela de translație*

## 3.2. Construirea payload-ului

Tot ce am scris până acum se află în INSECURE\_MEMORY. În această zonă de memorie va fi urcat și iBoot-ul. Prin urmare, payload-ul nostru trebuie să copieze codul pe care vrem să îl executăm și în următorul stadiu în altă zonă de memorie. Putem muta payload-ul nostru în SecureROM, acolo avem destul loc. Singura problemă: dacă acesta rămâne mapat ca RX, iBoot-ul intră în panică și resetează telefonul. Mai avem o variantă: la adresa 0x1801E4000 avem o pagina în care nu se scrie niciodată. Aceasta este numită HEAP\_GUARD și rolul ei este să ajute în detectarea bug-urilor de tip heap-overflow, pagina nefiind mapată, încercarea de citire sau scriere ar panica iBoot-ul. Ea este scrisă cu 0 la intrarea în execuție a iBoot-ului. Și aici avem o nouă problemă: va trebui să preluăm execuția iBoot-ului în momente cheie pentru a ne putea continua execuția:

* Va trebui să oprim scriere cu 0 în această pagină: aici se află codul nostru.
* Va trebui să mapăm această zonă în memorie ca executabilă.
* Va trebui să modificăm kernel-ul înainte de a se trece cu execuția în el.

Cum facem toate acestea?

Vom împarți payload-ul nostru în trei părți: callback chain, bootrom\_shellcode și boot\_companion. bootrom\_shellcode va fi un program mic și va avea ca scop copierea boot\_companion-ului în HEAP\_GUARD și trecerea la următorul stadiu: iBoot-ul. Pentru a trece la următorul stadiu trebuie să reparăm heap-ul prin scrierea unor structuri de tip heap\_block [7] care sunt folosite ca metadată de heap. Însă asta nu e suficient pentru a ieși din SecureROM. Dacă intrăm în DFU mode, singura modalitate prin care putem continua execuția este să uploadăm o imagine bună prin usb din care să booteze. De aceea trebuie și să modificăm codul SecureROM-ului pentru a permite căutarea iBoot-ului de pe disc, din care mai apoi să booteze. SecureROM și iBoot, înainte să transfere execuția următorului stadiu, sar într-o bucată de cod numită boot\_trampoline. Aceasta are ca scop pregătirea pentru intrarea în următorul stadiu, resetând majoritatea regiștrilor. Amândouă trambulinele încep cu aceleași trei instrucțiuni. Le putem căuta în memorie pentru a ști unde se află. Avem nevoie de acest lucru pentru a putea modifica codul să sară mai întâi în boot\_companion înainte de a preda execuția următorului stadiu. Astfel vom putea aplica modificările de care avem nevoie.

Pe scurt, bootrom\_shellcode trebuie să se asigure că vom continua execuția în iBoot de unde vom putea modifica kernel-ul.

Payload-ul va consta dintr-un buffer care va avea la început lanțul nostru de request-uri. Callback-urile vor apela gadget-urile prezentate mai înainte. Ultimul callback va pointa către adresa INSECURE\_MEMORY + 0x300. Acolo se va afla bootrom\_shellcode. Scopul lui va fi să copieze boot\_companion de la adresa INSECURE\_MEMORY + 0x800 la adresa 0x1801E4000, pagina de HEAP\_GURAD.

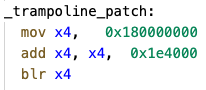


*Figura 3.15 Lanțul fake de request-uri*

### 3.3.1 Modificarea iBoot-ului

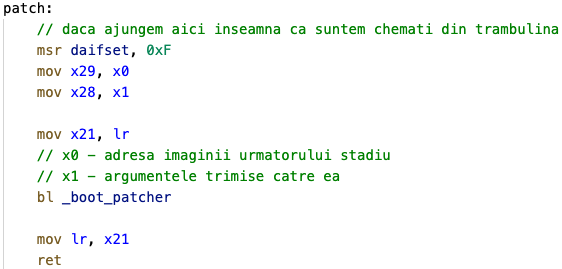
După cum am menționat, singurul loc în care putem pune codul nostru este o pagină în SRAM de 0x4000 bytes. Dacă dorim să compilăm chiar și un cod mic vom observa că executabilul rezultat are dimensiuni de ordinul kilo sau chiar megabytes-ilor. Codul nostru este ținut într-un format care oferă date despre adresa la care trebuie să fie încărcat, simboluri pentru debug, etc. Nu avem nevoie de toate acestea. Așa că vom scrie un program care să extragă dintr-un fișier Mach-O doar zona de TEXT. Mach-O este formatul fișierelor compilate pe Mac. El începe cu un header în care se află informații precum numărul de segmente din program și offset-ul la care se află acestea în fișier. Codul pentru acest program se află în fișierul “macho\_text.c” din proiect și îl vom folosi pentru a construi boot\_companion-ul și bootrom\_shellcode-ul.

Trambulina SecureROM-ului este ultima funcție chemată înainte de a preda execuția iBoot-ului. Am modificat primele trei instrucțiuni ale acesteia să cheme codul nostru punând în registrul X4 adresa noastră și sărind la ea.



*Figura 3.16*

Codul verifică în ce registru se află adresa noastră. Dacă se află în X4 atunci știe că trebuie să modifice următorul nivel. Trambulina primește ca argument în X0 adresa imaginii următorului nivel, iar în X1 argumentele pentru acesta. Deci știm de la ce adresă trebuie să începem căutarea instrucțiunilor pe care vrem să le modificăm. Începem mai întâi prin executarea celor trei instrucțiuni pe care le-am suprascris și salvând valoarea registrului LR. Apoi, chemăm funcția boot\_patcher. Această funcție cheamă la rândul ei iBoot\_patcher sau kernel\_patcher în funcție de stadiul la care ne aflăm.

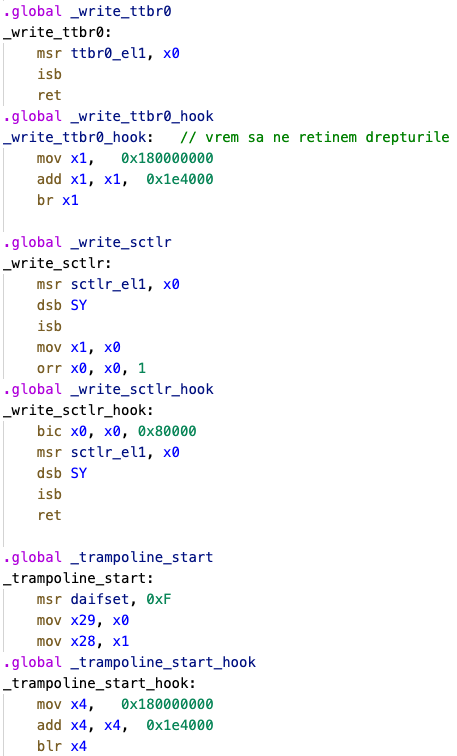
****

*Figura 3.17*

Pentru iBoot trebuie să preluăm execuția în câteva puncte cheie:

* Când se scrie adresa tabelelor de translație în registrul TTBR0: trebuie să ne mapăm pagina din care executăm ca executabilă înainte să se activeze MMU. Scrierea în TTBR0 nu activează MMU, dar dacă vrem să mapam pagina noastră manual, cum am facut în SecureROM, trebuie să stim adresa tabelelor de translație.
* Când se scrie în registrul SCTLR: dacă vrem să dezactivăm protecția WXN (Write-Execute-Never) aici putem face asta.
* Când se execută trambulina iBoot-ului: după cum am zis, execuția fiecărui stadiu se termină cu o trambulină care are ca scop ștergerea stadiului curent și transferul execuției către următorul. Dacă vrem să modificăm kernel-ul, aici o putem face.
* Când se șterge HEAP\_GUARD-ul: aici se află codul nostru și dorim să îl păstrăm în memorie.

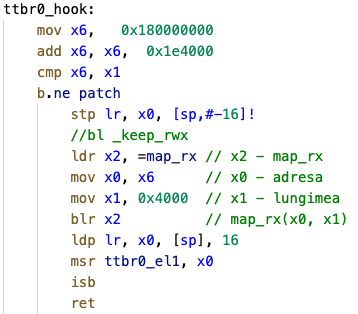
Pentru a modifica aceste locuri putem să căutăm cu Ghidra sau IDA adresa la care se află. Dar această abordare nu este rezistentă la update-uri, locul lor putându-se schimba. În plus, aceste instrucțiuni și regiștrii folosiți nu se schimbă prea des la fiecare update. Așa că vom scrie un patch-finder, al cărui cod se află în fișierul “utils.c” din proiect. Pe instrucțiunile pe care le căutăm și pe cele cu care le vom înlocui le vom pune în boot\_companion pentru a putea fi compilate direct de compilator: nu este nevoie să vedem cum sunt ele encodate. În imaginea următoare putem vedea instrucțiunile normale și instrucțiunile pe care le vom pune în locul lor:

****

*Figura 3.18 Patch-urile pe care le vom aplica iBoot-ului*

Codul care înlocuiește scrisul în SCTLR este executat tot acolo. Codul pentru scris în TTBR0 și codul de la începutul trambulinei au fost înlocuite cu salturi la adresa 0x1801E4000, pagina HEAP\_GUARD, locul unde se află codul nostru.

boot\_companion verifică în ce registru se află adresa 0x1801E4000 ca să știe ce trebuie să execute. Dacă aceasta se află în registrul X1, atunci el știe că trebuie să mapeze pagina noastră ca RX (read-execute) și să atribuie registrului TTBR0 valoarea care trebuia scrisă.

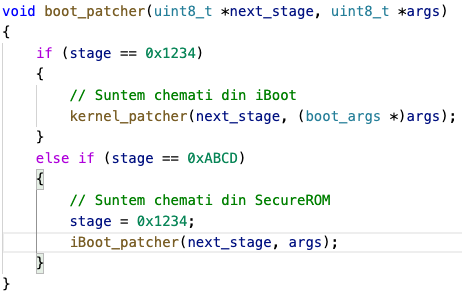
****

*Figura 3.19*

Putem să facem asta manual cum am făcut în cazul SecureROM. Doar că lucrurile sunt mai complicate de această dată. Acum se mapează mult mai multe zone și alocarea de tabele pentru niveluri mai joase, în cazul paginilor mai mici, nu mai este atât de ușor de înțeles. Putem folosi în schimb chiar funcțiile pe care iBoot le folosește pentru a inițializa MMU. Una dintre probleme este că nu există în cod posibilitatea de a mapa RWX. Însă, cât timp nu mai scriem în variabilele globale din boot\_companion, totul ar trebui să fie în regulă. Cunoaștem că zona de text a iBoot-ului trebuie mapată ca executabilă. Ne uităm care din funcțiile de mapare este chemată cu prim argument adresa ei. Știm că aceea trebuie să fie funcția care mapează o zonă ca RX. O chemăm cu primul argument, adresa noastră, și al doilea, lungimea zonei. După ce am terminat asta, scriem în TTBR0 adresa tabelei de translație.

Hook-ul, în cazul scrierii în SCTLR, este executat in-place. El reseteaza la 0 bit-ul 19. Acest pas nu mai este necesar din moment ce nu mai avem pagini mapate și ca write și ca execute, cum am avut la stadiul anterior, dar îl păstrăm în caz că îi vom găsi o întrebuințare.

Trambulina a fost modificată, și în acest stadiu, să ne cheme pe noi înainte de a preda execuția următorului stadiu, kernel-ul. De aici se apelează, ca în cazul SecureROM-ului, funcția boot\_patcher care știe că este rândul kernel-ului să fie modificat:

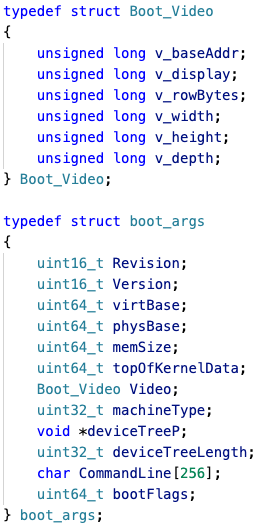
****

*Figura 3.20*

În acest moment imaginea kernel-ului este RW (read-write). Avem în X0 adresa lui și în X1 adresa către o structură de tip “boot\_args”.

### 3.3.2 Modificarea kernel-ului

Înainte să începem patch-urile dorim să verificăm dacă tot codul scris până în acest moment funcționează. Cel mai rapid mod de a face asta este să activăm “verbose boot”-ul. iBoot nu trimite prea multe argument kernel-ului. Printre cele puține pe care i le trimite totuși, clar nu se regăsește “-v”. Pentru a pasa noi acest argument, trebuie să modificăm structura boot\_args. Definiția acestei structuri este publicată de Apple ca open-source [8].

****

*Figura 3.21 Structurile cu argumentele pentru kernel*

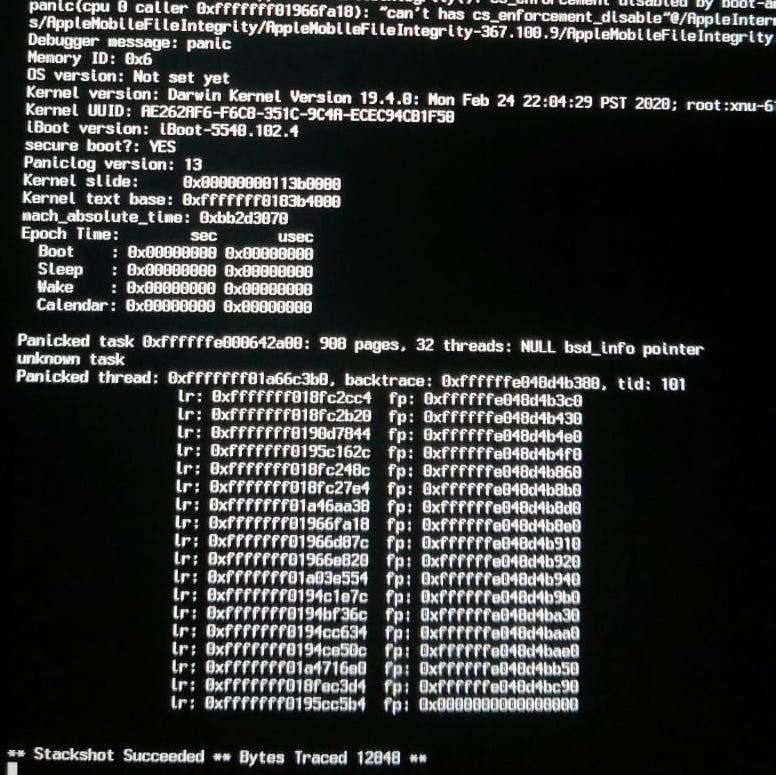
În variabila CommandLine trebuie să adăugăm “-v” la final și trebuie să resetăm la 0 variabila v\_display. Știm asta pentru că în iBoot există cod care tratează cazul în care argumentul “-v” este pasat kernel-ului, chiar dacă acest lucru nu se întamplă niciodată pe device-urile de release.

Succes! Telefonul booteaza verbose.

****

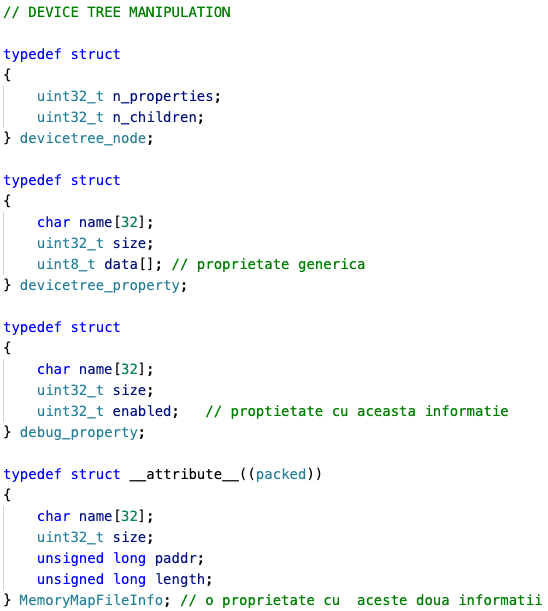
*Figura 3.22 Verbose boot*

Acum este timpul să modificăm kernel-ul. Vrem ca în userland orice aplicație să poată rula cod nesemnat și să poată accesa task\_port-ul kernel-ului sau al oricărui alt proces. În Mach, dacă un proces are send\_right către un task, atunci acesta poate face orice în memoria sa virtuală cu ajutorul api-urilor Mach, mach\_vm\_\*. Putem aloca cu mach\_vm\_allocate, putem citi cu mach\_vm\_read și putem scrie cu mach\_vm\_write. Dar ca să facem rost de task\_port-ul unui proces trebuie să trecem o serie de verificări în mach trap-ul “task\_for\_pid”. Pentru a rula cod nesemnat însă, trebuie doar să trimitem niște argumente kernel-ului [9]. Acestea sunt “amfi\_get\_out\_of\_my\_way=1” și “cs\_enforcement\_disable=1”. amfi\_get\_out\_of\_my\_way pare să nu mai fie funcțional pe ultimele versiuni, dar cs\_enforcement\_disable pare să fie suficient [10]. Adăugăm “cs\_enforcement\_disable=1” la linia de comandă și bootăm telefonul, dar observăm că acesta a intrat în panică:

****

*Figura 3.23 Kernel panic*

Mesjul este “can’t has cs\_enforcement\_diable(…)”. Căutăm acest string în kernel și observăm că pentru a putea trece de acest check trebuie ca funcția PE\_i\_can\_has\_debugger să returneze true (1). Pentru a descoperi o parte din simboluri ne folosim de sursele BSD publicate de Apple și de tool-ul “jtool” [11], scris de Jonathan Levin. Aflăm că PE\_i\_can\_has\_debugger returnează true dacă o variabilă globală este la rândul ei true. O sa ne referim la această variabilă ca debug\_enabled. Putem să modificăm funcția să returneze true, iar în acest caz ar funcționa în regulă. Însă funcția astfel modificată nu va emula perfect vechea funcționalitate, iar acest lucru ar putea panica kernel-ul mai târziu. Prin urmare, vom seta variabila debug\_enabled. Putem face asta căutând-o în memorie și modificând-o. Dar acest lucru nu este rezistent la update-uri: de fiecare dată trebuie să verificăm dacă nu cumva adresa ei s-a schimbat. Din fericire această variabilă poate fi setată chiar de kernel dacă acesta primește în deviceTree-ul primit ca argument o proprietate “debug-enabled” cu valoarea 1. Pointer-ul către acest deviceTree se află în variabila “deviceTreeP” din structura boot\_args primită ca argument. Această structură este compusă dintr-un nod rădăcină, urmat de proprietățile sale și mai apoi de copiii lui. Fiecare proprietate are și un nume: spre exemplu pentru a afla numele unui nod trebuie să căutăm proprietatea cu numele “name” și să citim din campul “data” acest nume.

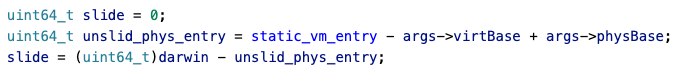
****

*Figura 3.24 Structuri pentru nodurile și proprietățile unui deviceTree*

Pentru a modifica variabila “debug\_enabled”, este de ajuns să găsim proprietatea cu același nume și să scriem în câmpul “data” 1.

Următoarele modificări pe care va trebui să le aducem kernel-ului vor trebui făcute direct de către noi. Pentru asta trebuie să știm adresa instrucțiunilor pe care le dorim modificate. Dacă decompilăm și analizăm kernel-ul în Ghidra sau IDA putem vedea adresele statice de memorie. Apple, însă, aplică kernel-ului o măsură de protecție cunoscută ca KASLR [12]. Pentru multe exploit-uri este necesar să știm adresele anumitor funcții sau variabile. Acest lucru nu este dificil având în vedere că un atacator poate face rost de aceste locații inspectând executabilul. Kernel Address Space Layout Randomization este o mitigație împotriva acestor atacuri. Toate adresele sunt glisate acum cu un număr de bytes numit slide. Nu consumă mult timp să aflăm acest slide prin bruteforce, dar dacă după ce activăm MMU încercăm să citim de la o adresa virtuala care nu a fost mapata, vom panica kernel-ul. O soluție pentru a calcula acest slide era să începem să căutăm baza imaginii kernel-ului de la adresa lui statică de început adunată cu slide-ul maxim. Însă acest lucru nu mai este practic în ultimele versiuni: granularitatea slide-ului a fost mărită considerabil și la fel și valoarea lui maximă. Șansele să picăm într-o zonă de meorie nemapată sunt mult mai mari.

iBoot calculează slide-ul pentru KASLR, aplicând o funcție de hash pe un număr generat random. Când iBoot încarcă în memorie kernel-ul, îl încarcă la adresa specificată de Mach-O-ul kernel-ului ca bază + slide. Puteam modifica calcularea slide-ului în iBoot, dar acesta este in-lined. Nu este atat de simplu precum modificarea unei funcții să returneze mereu 0. Putem găsi baza adevărată căutând în memorie, de la baza statică până la baza statică + slide-ul maxim, valoarea 0xFEEDFACF care indică începutul unui fișier Mach-O. Din fericire iBoot trimite kernel-ului, prin intermediul structurii boot\_args, baza virtuală și cea fizica. În X0 ne amintim că se află adresa la care va începe execuția în kernel. Aceasta este funcția “entry” și îi putem observa adresa statică în Ghidra. În acest moment putem calcula slide-ul kernel-ului. Îl vom aduna tuturor adreselor fizice statice pentru a calcula adresa fizică reală.

****

*Figura 3.25 Calcularea slide-ului*

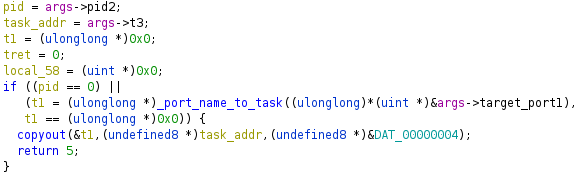
Acum putem modifica imaginea kernel-ului. Vom începe prin a reseta variabila globală “proc\_enforce” la 0. Aceasta indică dacă trebuie să se efectueze verificări de semnătură atunci când se apelează “fork()” sau alte funcții. Nu strică să o resetăm, dacă tot suntem aici. Trebuie să ținem minte că și cu toate aceste modificări tot nu putem rula executabile nesemnate. Ce putem face acum în schimb este să rulăm aplicații semnate de oricine sau semnate greșit. Acest lucru poartă denumirea de “fakesigned”. Această limitare este dată de daemonii existenți în userland care verifică că executabilele sunt semnate: de exemplu installd. Scopul lui este să blocheze instalarea aplicațiilor nesemnate de Apple. Însă, având în vedere că vom putea edita memoria acestor procese, barierele menționate nu mai reprezintă o problemă prea mare.

Putem trece la cel mai important patch, supranumit și task\_for\_pid0 sau tfp0. Urmează să modificăm funcția task\_for\_pid pentru a ne putea oferi drept de send către task-ul kernel-ului (pid 0) și chiar către celelate task-uri care rulează. Folosim din nou jtool pentru a afla, fără prea mult efort, adresa acestei funcții:

**https://lh6.googleusercontent.com/ur51Ye2i-ttztpp8YZH3Z0NDeTKRg2Ipn8Sw7Uw2KV83y_ip6Ox1KLWgOBPdFbccf-CF-iyZ0UkxG9anxtL2WRDnDwd1HapuOU2j5bmvNiJlSx4goVoVCMZU9iUzliys8KsTHah7**

*Figura 3.26 Simbolurile aflate încă în kernel cu* *“task\_for\_pid” în nume*

Ne ducem în executabil la adresa indicată și observăm un prim lucru: dacă pid-ul căruia îi vrem task-ul este 0, atunci funcția se termină și returnează 5 (KERN\_FAILURE); din nou am redenumit funcțiile și recreat structurile pentru a face codul mai ușor de înțeles:

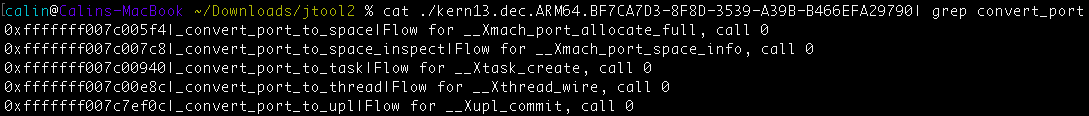
****

*Figura 3.27 Începutul mach trap-ului task\_for\_pid*

În disassembly putem observa că în registrul W23 este ținută valoarea pid-ului și că o instrucțiune “CBZ W23, fail” (branch on 0) oprește execuția funcției. Vom scrie în locul acestei instrucțiuni, instrucțiunea “NOP” (no operation). În continuare, funcția “proc\_find” este chemată cu acest pid ca argument. Rezultatul este structura “proc” cu informații despre procesul cu acel id. Mai departe, funcția “task\_for\_pid\_posix\_check” verifică dacă putem lua task-ul acestui proces. Vom modifica task\_for\_pid să își continue execuția indiferent de rezultatul întors de acea funcție. Următorul punct de care trebuie să trecem este mac\_proc\_check\_get\_task. Ca și în cazul anterior, modificăm task\_for\_pid să nu ia în seama acest rezultat.

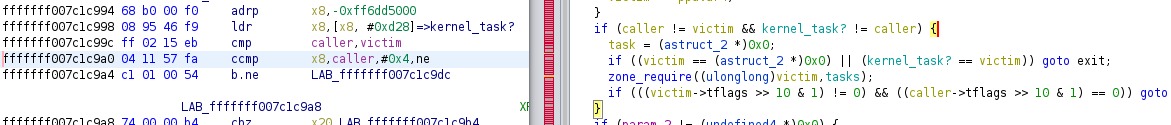
Urmează să fie verificat procesul nostru: dacă suntem root sau cerem chiar task-ul procesului nostru, atunci ne este permisă primirea task-ului. Putem modifica aceste verificări cu o serie de NOP-uri sau putem, tot aici, să ne dăm drepturi de root. Rezultatul ar fi că orice proces care cheama task\_for\_pid va primi și drepturi de root la întoarcerea din kernel.

Compilăm exploit-ul și rulăm. După ce ajungem în userland testăm codul chemând task\_for\_pid pentru pid-ul 0. Observăm că deși task\_for\_pid ne întoarce cu succes task-ul kernel-ului (și al oricărui alt proces), nu putem folosi api-ul mach\_vm\_\* decât în procesul nostru și alte procese neprivilegiate, care nu dispun de dreptul “platform-application”. Devine clar faptul că există verificări și în alte parți în kernel care se asigură că și dacă facem rost de task\_for\_pid0, nu îl putem folosi. După ce studiem sursele Apple aflăm că există o astfel de funcție: task\_conversion\_eval [13]. Pornim jtool și căutăm această funcție, însă nu obținem nimic. Fie acest simbol nu există în kernel, fie funcția a fost compilată in-line. Știm că această funcție accesează kernel\_task din memorie și știm unde ar putea fi apelată: în funcții care accesează informația unui port. În final aflăm care sunt posibilele funcții care cheamă task\_conversion\_eval:

****

*Figura 3.28*

Verificăm fiecare adresă în Ghidra. Ne uităm după un cod care decompilat să semene cu cel din sursele Apple. Într-adevăr găsim două funcții care apelează acest check: “convert\_task\_to\_port” și “convert\_port\_to\_space”; variabilele și funcțiile au fost redenumite și structurile recreate pentru a face codul mai ușor de înțeles:



*Figura 3.29 task\_coversion\_eval in-line într-una din funcții*

Intrăm în userland, testăm din nou api-ul mach\_vm\_\* și observăm că îl putem utiliza cu succes în orice proces. Putem scrie/citi/aloca/dezaloca memorie chiar și în memoria virtuală a kernel-ului. Acest lucru este suficient să putem face orice dorim, direct din userland.

## 3.4. Aplicație proof-of-concept

Un ultim lucru pe care îl putem face din iBoot este să printăm pe ecran câteva informații care ne-ar putea fi de folos în userland: slide-ul kernel-ului și adresa virtuală la care se află acesta. Putem să le regăsim și manual, uitându-ne după valoarea 0xFEEDFACF, dar dacă tot suntem aici putem să le oferim utilizatorului. Pentru asta avem nevoie de o tabelă “teletext” cu informații despre caracterele pe care vrem să le printăm și o funcție “print\_addres” cu care să scriem la ecran [14]. Trebuie să ne amintim să punem tabela teletext în zona de TEXT a payload-ului, altfel va fi pierdută după parsarea binarului în tool-ul macho\_text pe care l-am scris. Toate variabilele globale trebuie puse în această zonă.

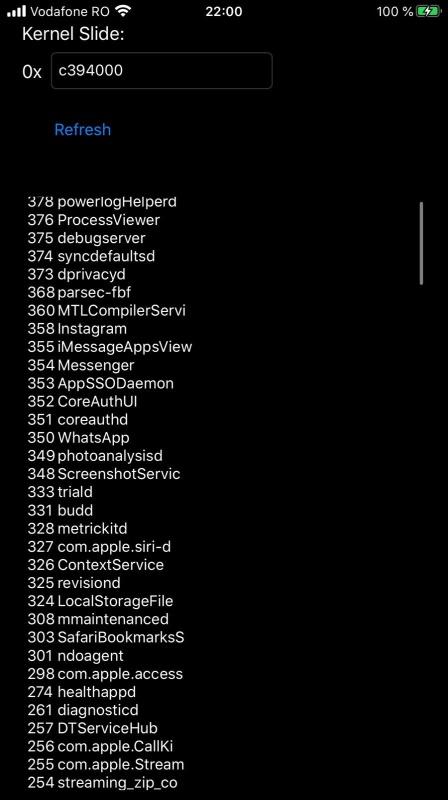
Pentru a scrie pe ecran trebuie să știm unde începe adresa memoriei video. Desigur, aceasta este trimisă ca argument kernel-ului, altfel sistemul de operare nu ar putea afișa nimic la ecran. Adresa la care începe memoria video se află în campul “v\_baseAddr” al structurii Boot\_Video. Un pointer către structura Boot\_Video care reține această informație se află în câmpul Video al structurii boot\_args. Adăugăm și acest feature exploit-ului și bootăm. Utilizatorul poate nota informațiile pe care i le vom printa la ecran.



*Figura 3.30 Pătrățelele indică succesul diferitelor stadii ale funcției kernel\_patcher. Primul număr este slide-ul kernel-ului. Al doilea este adresa la care a fost încărcat.*

Acum, dacă o aplicație dorește acces la memoria virtuală a vreunui proces trebuie doar să cheme “task\_for\_pid”. De exemplu, pentru a putea citi sau scrie memoria kernel-ului, trebuie să chemăm “task\_for\_pid(mach\_task\_self(), 0, &task\_port)”: mach\_task\_self(), pentru că noi dorim această capabilitate, 0 este pid-ul kernel-ului, iar în task\_port va fi pus send\_right-ul. Acesta este ca un fel de file descriptor pe care va trebui să îl trimitem ca argument funcțiilor mach\_vm\_\*.

Cu acces la memoria oricăriu proces, putem face orice. O să scriem, de exemplu, o aplicație care să listeze toate procesele care rulează la un moment dat. Acesta este un feature care se poate dovedi important, dar Apple a dorit să nu îl ofere consumatorilor. Multe exploit-uri nu își ascund execuția față de sistem sau utilizator, pentru că acesta nu are cum sa vadă ce rulează pe telefon. Au fost cazuri in-the-wild, în care exploit-uri 0-day au fost folosite pentru a instala pe telefonul victimelor un daemon care să extragă diverse date din telefon, precum conversații decriptate din Whatsapp [15]. Codul aplicației se afla în ProcessViewer.



*Figura 3.31* *Aplicație care listează procesele (pid-ul și numele lor)*

Desigur, putem face lucruri și mai interesante, având în vedere că există proiecte open source care pot instala diferite tool-uri pe iPhone cu ajutorul tfp0; de exemplu jelbrekLib [16] sau Chimera13 [17]. Vom opta pentru Chimera și o vom modifica să meargă cu backdoor-ul nostru. Codul modificat se află în folder-ul Chimera13. Cu ajutorul lui putem instala dropbear, un server de SSH pe telefon. Parola default pentru user-ul root este “alpine” (aleasă de Apple).

https://lh3.googleusercontent.com/hgKoGQk2xvdFUHyaC305caSlOmTHoXHi809AvnvTfSlAZl70a8bCkM6lJX8I7LGvp8vbUxtItOnlzA2RU-yDfHrsJGAhl8u0RJ3qyzVPfLdg_nBMHgaYDYEgb-h_Cxoza4JTlFhQ

*Figure 3.32*

# 4. Concluzii

Am descoperit că un utilizator trebuie să depună un efort considerabil ca să poată folosi hardware-ul pe care l-a cumparat așa cum își dorește el. Apple impune foarte multe restricții, o parte din ele având sens pentru aplicații străine, dar nu și pentru utilizator. De exemplu, consider că noi ar trebui să putem verifica ce procese rulează pe telefon, dar desigur aplicații străine precum Facebook nu ar trebui să aibă acces la astfel de informații pentru că ar putea pune o amprentă digitală telefonului mult mai ușor.

O altă măsură cu care nu sunt de acord este necesitatea aplicațiilor de a fi semnate de Apple, deci să fie distribuite doar prin magazinul lor. Compania susține că acest lucru este necesar pentru a se asigura că aplicațiile sunt sigure. Acest argument pică din start, având în vedere că au fost distribuite cel puțin două exploit-uri 0-day, chiar în AppStore. Mai mult, de ce este nevoie ca aplicațiile să fie semnate de Apple, când indiferent de semnătură aplicațiile rulează într-un sandbox extrem de restrictiv? Orice operație trebuie aprobată explicit de utilizator indiferent de semnătura aplicației: o aplicație nu poate accesa galeria fără acordul utilizatorului. În acest mod, Apple deține un monopol motivat de “siguranța” utilizatorilor. Dezvoltatorii sunt astfel obligați să plătească 30% din profit pentru a-și distribui aplicațiile. Acest lucru este nedrept, având în vedere că firme precum Spotify, Netflix sau Amazon, aflate în competiție directă cu produse Apple, trebuie să concureze cu prețurile companiei. Browserele de pe platformă trebuie să concureze și cu viteza superioară a motorului Safari. Acesta este singurul executabil care are voie să mapeze pagini ca RWX (read-write-execute) și deci poate executa cod JIT.

În plus, au fost ocazii în care update-uri ale iOS-ului au oprit funcționalitatea butoanelor Touch ID sau a ecranelor înlocuite de service-uri neautorizate de Apple. Desigur, securitatea sistemului poate fi slăbită de butoane fabricate să transmită amprenta către un agent (răuvoitor). Dar care sunt șansele ca acest lucru să se întâmple? Consumatorii ar trebui notificați și ei ar trebui să aleagă dacă își asumă sau nu riscul de a folosi componente aftermarket.

Acestea și pasiunea pentru inginerie inversă m-au motivat să scriu această lucrare. În proiect se regăsesc sursele pentru tool-urile scrise de mine și pentru Chimera13 care a fost modificată să funcționeze cu acest jailbreak. În folder-ul “macho\_text” se află sursa pentru programul care extrage zona de TEXT dintr-un fișier Mach-O. Se compilează cu “sudo make”, pentru a copia direct executabilul rezultat la calea “/opt/pkg/bin/”. În folder-ul “exploit” se află programul care exploatează bug-ul. El funcționează pe ultima versiune de iOS (la momentul scrierii lucrării, 13.5.1), doar pe iPhone 8, iPhone 8 Plus și iPhone X. Toate iPhone-urile lansate înaintea acestora sunt vulnerabile la acest bug, dar exploit-ul trebuie modificat ca să ruleze cu succes și pe acestea. Se compilează cu “make”. Petru aceasta este nevoie ca “libusb” să fie instalată. Se rulează neapărat cu “sudo”, altfel exploit-ul eșuează: “libusb” este un framework care rulează în userland, nu este o extensie în kernel. Fără “sudo”, alte procese sau extensii mai privilegiate nu ne vor elibera device-ul pentru a putea comunica cu el. În folder-ul teste se regăsește o serie de programe care m-au ajutat să înțeleg cum este mapată memoria de tabelele de translație sau care să mă ajute să îmi dau seama care sunt locațiile în care aveam posibilitatea de a încărca payload-ul exploit-ului. În folder-ul “ProcessViewer” se află codul aplicației care citește procesele din lista de procese aflată în kernel. În folder-ul “Chimera13” avem codul aplicației, care deschide o sesiune de SSH, modificat pentru a funcționa cu exploit-ul nostru. Parola utilizatorului root este “alpine”. Utilizatorul trebuie să modifice variabila “slide” cu numărul afișat de noi pe ecran în timpul bootării. Foarte important este de menționat că acest exploit slăbește considerabil securitatea sistemului de operare. Cât timp rulați aplicații din AppStore sau aplicații scrise/verificate de dumneavoastră totul este în regulă. Dacă doriți să dezactivați backdoor-ul, este suficient să restartați telefonul. Exploit-ul nu este persistent (untethered). De aceea, cei care cred că au fost victimile unui astfel de exploit pentru acest bug, pot restarta telefonul. De exemplu, în aeroporturi în țari precum America se poate cere telefonul persoanelor pentru a se face o serie de verificări. Checkm8 poate fi folosit pentru a instala un implant în telefon care poate să captureze parola utilizatorului și să continue să ruleze în background de unde poate accesa în continuare alte informații.

Pe viitor doresc să continui să lucrez la acest exploit, să adaug funcționalități de debug kernel-ului și să fac injecția și instrumentarea de cod în userland mult mai ușoară. Desigur, există jailbreak-uri care fac deja asta, precum checkra1n [18], dar cel mai bine înveți prin practică.

# 5. Bibliografie

[1] Quraishi, Nadeem Ahmad, et al. *Protocols and standards for USB peripheral communications.* U.S. Patent No. 7,290,072. 30 Oct. 2007.

[2] axi0mX. *ipwndfu*. *Github*, 27 Sep. 2019, [github.com/axi0mX/ipwndfu](https://github.com/axi0mX/ipwndfu)

[3] littleailo. *apollo*. *Github*, 27 Sep, 2019, [gist.github.com/littlelailo/42c6a11d31877f98531f6d30444f59c4](https://gist.github.com/littlelailo/42c6a11d31877f98531f6d30444f59c4)

[4] Warkentin, Andrei, Cyprien Laplace, and Ye Li. "Memory management in virtualized computing systems having processors with more than two hierarchical privilege levels." U.S. Patent No. 10,002,084. 19 Jun. 2018.

[5] axi0mX. *checkm8.py*. *Github*, 27 Sep. 2019, [github.com/axi0mX/ipwndfu/blob/master/checkm8.py](https://github.com/axi0mX/ipwndfu/blob/master/checkm8.py)

[6] Daniel, Mark, Jake Honoroff, and Charlie Miller. "Engineering Heap Overflow Exploits with JavaScript." *WOOT* 8 (2008): 1-6.

[7] axi0mX. *ipwndfu. Github, 27 Sep. 2019,* [*github.com/axi0mX/ipwndfu/blob/master/ipwndfu#L375*](https://github.com/axi0mX/ipwndfu/blob/master/ipwndfu#L375)

[8] opensource.apple.com/source/xnu/xnu-201.5/pexpert/pexpert/i386/boot.h.auto.html

[9] Esser, Stefan, et al. *IOS Hacker's Handbook.* United States, Wiley, 2012.

[10] Jakea James. *rootlessJB writeup.* *Github*, 6 Apr. 2019, [*github.com/jakeajames/rootlessJB/blob/master/writeup.pdf*](https://github.com/jakeajames/rootlessJB/blob/master/writeup.pdf)

[11] Jonathan Levin. *JTool2*. *newosxbook,* 2 Ian. 2019, [www.newosxbook.com/tools/jtool.html](http://www.newosxbook.com/tools/jtool.html)

[12] Canella, Claudio, et al. "KASLR: Break It, Fix It, Repeat." *ASIA CCS 2020-Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ACM/IEEE, 2020.

[13] [opensource.apple.com/source/xnu/xnu-4570.41.2/osfmk/kern/ipc\_tt.c.auto.html](https://opensource.apple.com/source/xnu/xnu-4570.41.2/osfmk/kern/ipc_tt.c.auto.html)

[14] Linus Henze. *iStrap*. *Github,* 11 Apr. 2020, [github.com/LinusHenze/Fugu/tree/master/iStrap](https://github.com/LinusHenze/Fugu/tree/master/iStrap)

[15] Ian Beer. *A very deep dive into iOS Exploit chains found in the wild. Project Zero,* 29 Aug. 2019, [googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html](https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html)

[16] Jakea James. *jelbrekLib*. *Github,* 21 Aug 2018, [github.com/jakeajames/jelbrekLib](https://github.com/jakeajames/jelbrekLib)

[17] CoolStar. *Chimera13*. *Github,* 1 Mar. 2020, [github.com/coolstar/Chimera13](https://github.com/coolstar/Chimera13)

[18] Luca Todesco, et al. *All Releases.* *checkra1n*, 2020, [checkra](https://checkra.in).in/releases/