Jailbreaking iOS 11.1.2

An adventure into the XNU kernel – Bryce Bearchell

Table of Contents

Introduction 2

Background to December 11th 2

Async Wake 2
- **What is tfp0** 2
- **Async Wake Bugs** 2

Getting Root 3
- **Getting root with tfp0** 4

Signed process execution woes 8
- **AMFID** 8
- **Kernel Page Protections** 8
- **Sandbox Protections** 9

It Gets Worse 9
- **You can't patch the kernel** 9
- **You can "almost" patch the kernel** 9
- **You shouldn't patch the kernel** 9

Extracting the kernel 10
- **Reverse Engineering Useful Things** 11

Platform attributes 15
- **Problems getting root** 16

Getting access to the root filesystem 17

Neutralizing Apple Mobile File Integrity Daemon 18

Task For PID Entitlements 18

Completing the Jailbreak 19
- **Notes** 21

TL DR; How to jailbreak your phone 21
- **Loading the Jailbreak onto your phone** 22
- **Build and Load Jailbreak using XCode** 22
- **Side Load using Cydia Impactor** 22
  - Obtain an App-Specific password 22
  - Use Impactor to side load the IPA 23
- **Side load using Apple Configurator 2** 23
- **Win!!!** 24

Final Thoughts 24

Credits 24

Introduction

This was an exercise in quickly turning around a Proof Of Concept (POC) and weaponizing it into a useful jailbreak. The POC was released on December 11th, and the final jailbreak was finished on January 4th, 24 days. I was able to begin full time on the 18th and took a half day on Christmas and 2 days off over the New Year for alcohol related activities. From January 4th to the 12th I spent fixing critical bugs (e.g. pressing menu would cause a kernel panic) and writing my report, bringing the time I worked on the jailbreak to 13.5 days. Six books were read and 4k+ lines of Object C / C code were written, thrown away, and re-written. The entire project is available open source on GitHub and the following text is my journey from knowing nothing of iOS kernel internals to writing a jailbreak for it.

Background to December 11th

The fundamental kernel that drives both iOS and MacOS is the XNU kernel, which is a combination of BSD and Mach duct taped together to form the iOS kernel. As of late 2017, Apple open sourced the XNU kernel, and since iOS 10 has included it *unencrypted* as part of iOS software pushes (Lending itself to higher public scrutiny and reverse engineering). There was an unofficial race of first-to-market public exploit for use in jailbreaking iOS 11, and Google Project 0's Ian Beer won it (Ian gets my nomination for 2018's Pwnie Awards). So, then a secondary race began to release a jailbreak for iOS 11.* because just the exploit was released. This race wasn't won by myself, but rather Jonathan Levin published a jailbreak using his QiLin jailbreak framework on December 29th, 2017 (please note that no discussion of jailbreaking iOS 10 is here; Pangu8, Yalu, and several others have released jailbreaks for that). Other public releases of jailbreaks have emerged recently such as Electra.

Async Wake

On December 11th, 2017, Ian Beer published to the Google bug tracker an XCode project named async_wake_ios. The project exploits several bugs and returns tfp0.

What is tfp0

The tfp in tfp0 stands for task_for_port which is a Mach trap to obtain a task port for a process. Think of a task port as an IPC mechanism to read / write memory in a process' memory space. It's even nicer than an arbitrary memory read and write, because we can use the offsets (accounting for KASLR) in the kernel to directly access (*cough* patch *cough*) code.

Async Wake Bugs

The two bugs that comprise the exploit are an information leak (CVE-2017-13865) and a Use-After-Free (CVE-2017-13861). The information leak is a result of a bug in proc_pidlistuptrs that is the result of improper bounds checking that enables the reading of 7 extra bytes, which are copied directly from kernel mode to user mode, enabling the bypass of KASLR via enumeration of a large number of address. Next, the Use-After-Free bug results from IOSyrfaceRootUserClient improperly handling references once an error has occurred in its sub-functions, which can be triggered if a previous call to it (and its subsequent async function,

wake_port), allowing a dangling pointer to freed memory in the kernel. This, coupled with some innovative heap feng shui allows for the creation of an arbitrary kernel read and write, which Ian wrangles into a pointer to the kernel task process, tfp0.

In order to properly exploit the Use-After-Free a bunch of kalloc allocations are made pointing to the target port, then a number of mach ports are allocated to ensure the current page only contains mach ports owned by our PID. Then the IOSurface dangling pointer bug is triggered by slowly reallocating memory until the garbage collector collects the page that the IOSurface pointer references.

The reallocations contain crafted ipc_kmsg messages with a fake IKOT_TASK task port which points to a process block structure (bsd_info) that contains a fake task structure. This can then be leveraged into an arbitrary read primitive

Once the read primitive has been used to find the kernel's vm_map and ipc_space then the task port is overwritten via reallocating the previous kalloc.4096.

Included in Ian's XCode project is a fantastic README that dives much deeper into the exact mechanism that were used during exploitation.

One thing to note is that, in my experience, you can throw the exploit 3 times without compromising the integrity of the phone. The 4th time, the kernel panics and the phone reboots (11.1.2 / 15B202).

Getting Root

I downloaded the XCode project from Google Bugs site (https://bugs.chromium.org/p/project-zero/issues/detail?id=1417#c3), updated the signing with my personal developer account and ran the project. I was greeted with a blank screen on my phone, but the XCode console revealed the exploit at work:

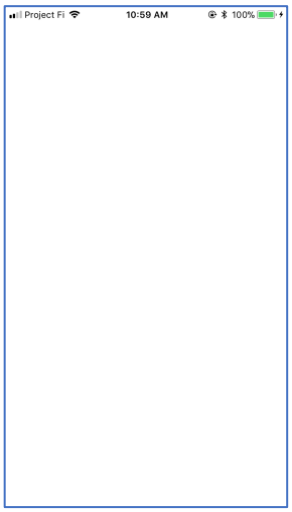


Figure 1 – iPhone view of the async_wake_ios exploit

```
build_id: 15B202
sysname: Darwin
nodename: nokia-388
release: 17.2.0
version: Darwin Kernel Version 17.2.0: Fri Sep 29 18:14:51 PDT
2017; root:xnu-4570.20.62~4/RELEASE_ARM64_S8000
machine: iPhone8,1
this is iPhone 6s, should work!
message size for kalloc.4096: 2956
got user client: 0x6207
 [+] prepared kqueue
task self: 0xffffffff11095ef40
our task port is at 0xffffffff11095ef40
found target port with suitable allocation page offset:
0xffffffff112b9fa68
replacer_body_size: 0xb74
message_body_offset: 0x448
0
e00002c9
0
0
1
2
```

Figure 2 - XCode console of async_wake exploit in progress (truncated)

```
198
199
got replaced with replacer port 45
found kernel vm_map: 0xffffffff10a55de80
second time got replaced with replacer port 0
will try to read from second port (fake kernel)
kernel read via fake kernel task port worked?
0x0000000000420000
0x0000000000000000
0xffffffff10a5863c0
0xffffffff10a586410
about to build safer tfp0
message buffer: ffffffff110ab8000
fake_kernel_task_kaddr: ffffffff110ab8000
read fake_task_refs: d00d
about to test new tfp0
kernel read via second tfp0 port worked?
0x0000000000420000
0x0000000000000000
0xffffffff10a5863c0
0xffffffff10a586410
built safer tfp0
about to clear up
cleared up
tfp0: 1888b0b
```

Figure 3 - Obtaining tfp0 with async_wake exploit (truncated)

I've truncated the output from 0,1,2,3…,198,199, but the main result is that we've obtained tfp0.

Getting root with tfp0

I want root on the device. This is NOT the same thing as a jailbreak, I just want PID 0 credentials and all of the privileges that go with it so that I can progress towards a jailbreak. A full jailbreak means I can run unsigned (or self-signed) code as root on the device, but that goal is some ways away. To get root we need to get access to all process' bsd_info which is part of a kernel structure. This structure is what makes XNU beautiful to work with, however, we have to kludge together some offsets with tfp0 to get access to it.

To do this, I wrote some code heavily based off of Abraham Masri (@cheesecakeufo). The idea is to get our process address via Ian Beer's code in Mach Portal, then perform the following:

1. Iterate through all of the process blocks via a doubly-linked-list structure
    a. A process block looks like this: {

- back-pointer(0x8)
- forward-pointer(0x8)
- PID(0x10),
- …,
- creds(0x100),
- …}

2. Find PID 0 and copy the pointer to its creds (offset 0x100 in the process block structure).
3. Find my process PID (obtained via getpid()) and overwrite my cred pointer.
4. Loop through all the active threads in the process block
5. Copy the creds to their structure
6. Win!

One caveat is that I need to save my old credential pointer and restore it before my process exits, otherwise all hell breaks loose and the kernel panics. Likewise, any tampering with the process block could also lead to kernel panics if not done properly.

An interesting situation here is that because execution is not defined by processes, rather threads are execution tasks for the processor, (processes are seen as blocks of resources, threads are the actual code execution tasks) it is possible to completely hide a process from the kernel but keep execution in the task scheduler by modifying the back and forward pointers of the surrounding process blocks (same as process hiding on windows).

You'll notice that the steps to get root listed above are almost one-to-one with token stealing attacks on windows to gain NT AUTHORITY\SYSTEM.

Here's the attack in action:

```
tfp0: 1888d0b
[+] Attempting to obtain root
[i]     old:
[i]     uid=501 gid=501 euid=501 geuid=501
[d] Patching ourselves with pid(0) at 0xffffffff013801cd0
[d] Old creds: 0xffffffff1154aff00
[d] Patching our creds with 0xffffffff112325ef0
[i]     new:
[i]     uid=0 gid=0 euid=0 geuid=0
[+] Got ROOT!!!!!
[+] Reverting privs to avoid a crash...(getuid=501)
```

Figure 4 - Getting root by stealing creds from PID 0

Fantastic, we now have root! Now, to prove that this is the case, let's attempt to read some sensitive files on the operating system that only root should have access to:

```
tfp0: 188890b
[+] Attempting to obtain root
[i]     old:
[i]     uid=501 gid=501 euid=501 geuid=501
[d] Patching ourselves with pid(0) at 0xffffffff014001cd0
[d] Old creds: 0xffffffff11232c000
[d] Patching our creds with 0xffffffff11232db00
[i]     new:
[i]     uid=0 gid=0 euid=0 geuid=0
[+] Got ROOT!!!!!
[i] Dumping [/etc/passwd]
##
# User Database
#
# This file is the authoritative user database.
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:/smx7MYTQIi2M:0:0:System Administrator:/var/root:/bin/sh
mobile:/smx7MYTQIi2M:501:501:Mobile User:/var/mobile:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_ftp:*:98:-2:FTP Daemon:/var/empty:/usr/bin/false
_networkd:*:24:24:Network Services:/var/networkd:/usr/bin/false
_wireless:*:25:25:Wireless Services:/var/wireless:/usr/bin/false
_installd:*:33:33:Install Daemon:/var/installd:/usr/bin/false
_neagent:*:34:34:NEAgent:/var/empty:/usr/bin/false
_ifccd:*:35:35:ifccd:/var/empty:/usr/bin/false
_securityd:*:64:64:securityd:/var/empty:/usr/bin/false
_mdnsresponder:*:65:65:mDNSResponder:/var/empty:/usr/bin/false
_sshd:*:75:75:sshd Privilege separation:/var/empty:/usr/bin/false
_unknown:*:99:99:Unknown User:/var/empty:/usr/bin/false
_distnote:*:241:241:Distributed Notifications:/var/empty:/usr/bin/false
_astris:*:245:245:Astris Services:/var/db/astris:/usr/bin/false
_ondemand:*:249:249:On Demand Resource Daemon:/var/db/ondemand:/usr/bin/false
_findmydevice:*:254:254:Find My Device Daemon:/var/db/findmydevice:/usr/bin/false
_datadetectors:*:257:257:DataDetectors:/var/db/datadetectors:/usr/bin/false
_captiveagent:*:258:258:captiveagent:/var/empty:/usr/bin/false
_analyticsd:*:263:263:Analytics Daemon:/var/db/analyticsd:/usr/bin/false
_timed:*:266:266:Time Sync Daemon:/var/db/timed:/usr/bin/false

File dumped
[+] Reverting privs to avoid a crash...(getuid=501)
```

Figure 5 - Reading /etc/passwd as root

Hell yeah. After throwing "/smx7MYTQIi2M" into a password cracker it seems to be "alpine", which is a quick google could have told us. But cracking is fun!

After proving that we can read arbitrary files on the system, the next step was to make it easier by taking a C web server and adapting it to our current constraints. No HTML, so we have to list files/directories and link-ify them and we cannot have a multi-threaded process (currently fork()'ing will cause a sandboxing failure and kill our code).

So, here is the code in action:

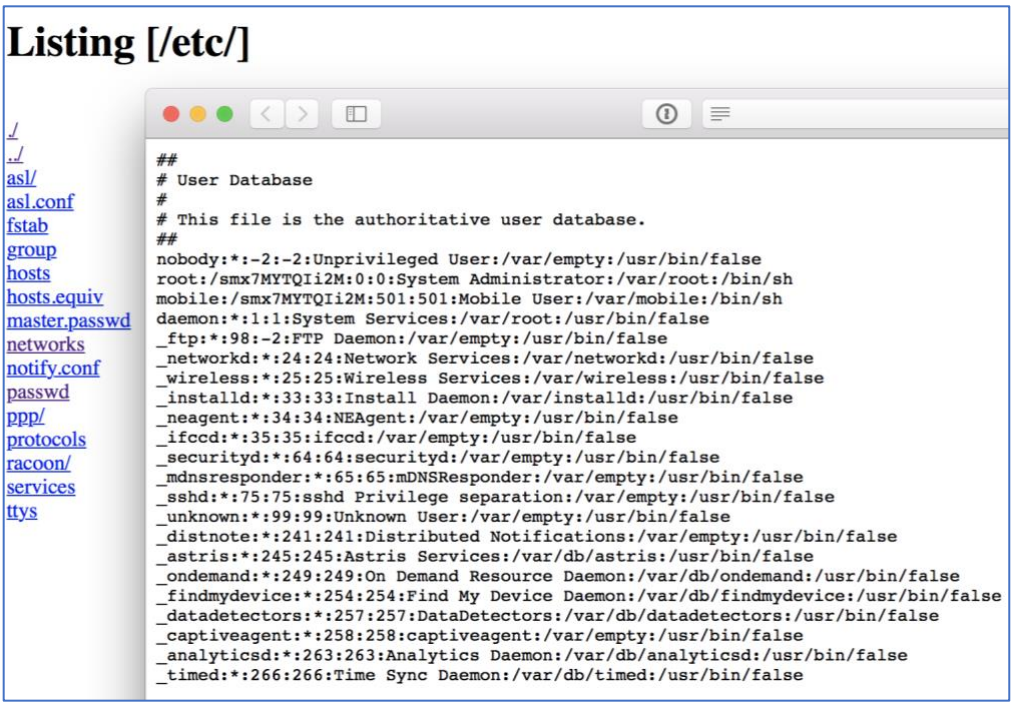


Figure 6 – Webserver serving the root file system

This turned out to be a great way to interact with the iPhone instead of rebooting / recompiling / deploying the app. Several features were added to the webserver so to aid in jailbreak development:

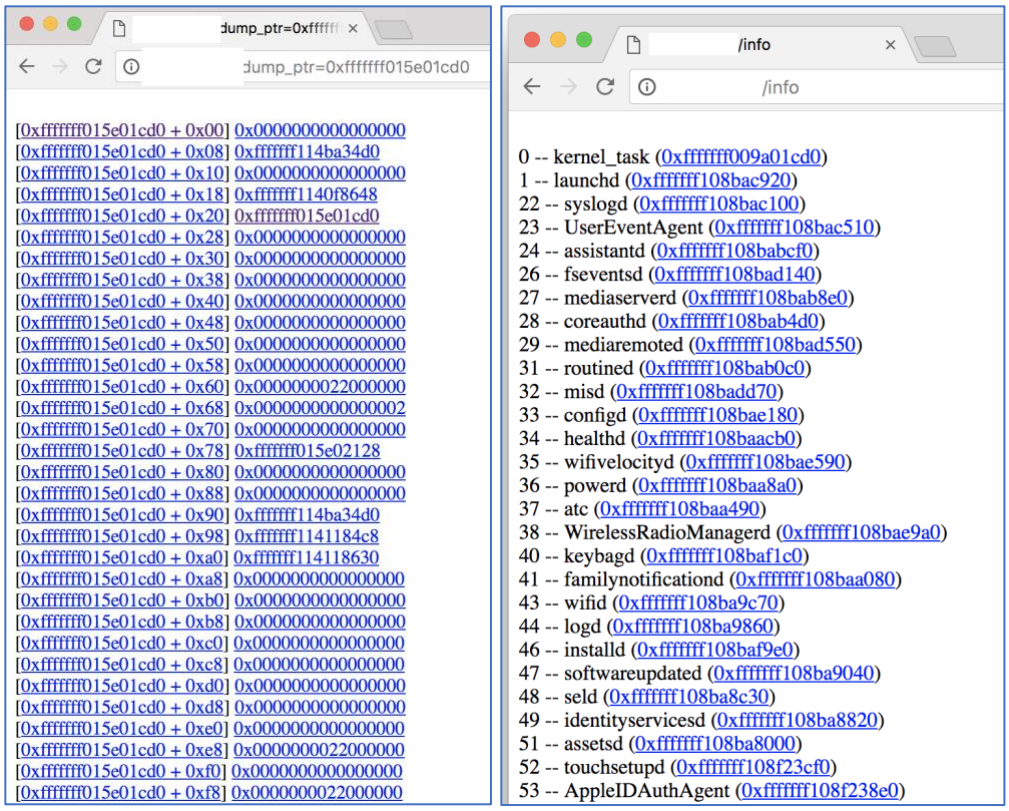


Figure 7 – Dumping live kernel memory

Signed process execution woes

So, given that we have root on the phone, we've achieved a jailbreak right? That couldn't be further from the truth. There are a number of protections in place that stop us from running arbitrary executables (both signed and unsigned).

AMFID

AMFI stands for Apple Mobile File Integrity, and AMFI**D** is the Daemon that is running as a process that gets called whenever a binary is loaded off of disk. This is the heavy hitter that we will have to deal a lot with, as *almost everything* is signed. Validation takes place of the code signing and if a binary is not properly signed, it is immediately killed off.

Kernel Page Protections

A code page in memory can never be RWX (Read / Write / Execute) unless it has the JIT entitlement, which is allowed for Safari alone (so that it's JavaScript engine runs faster than its Chrome or Firefox). We would like to run code that isn't bound by the RW / RX policy present on the system.

Sandbox Protections

There are restrictions on where binaries can be loaded from. This sandbox is to stop apps from running binaries outside of their allowed directories, and normally protects against malware but we want to subvert this functionality and be able to run arbitrary binaries on the system.

It Gets Worse

So, in order for patching of launchd and amfid to take place, we need to get a task port to them and this is accomplished via the task_for_port command. However, this has been patched since iOS 10 and we will need to patch the kernel to allow us to make this call without the proper entitlements.

You can't patch the kernel

The kernel is running a patch guard to defend against this very attack that we are trying to perform. Every 60 seconds, or when processor utilization is low, a kernel process goes through and checks each of the kernel pages and hashes them to determine if they have been changed. If they have, the kernel panics and you get a reboot.

You can "almost" patch the kernel

So, the choices of patching the kernel is to either determine where KPP process' code is at and patch *that* to make sure that the KPP process thinks everything is OK—This scenario is much more difficult due to how ARM processors are structured. Somewhat like the rings of privilege in x86, ARM has Exception Levels (EL) ranging from:

- EL0 – Untrusted user code
- EL1 – Trusted kernel code
- EL2 – Hypervisor mechanism (unused in iOS AFAIK)
- EL3 – EL1 protection mechanisms (Kernel Patch Protection et. All.)

An alternative is to perform the patch, keep the utilization of the processor high, and revert the patched code to the original once you've used it and hope the KPP process running in EL3 doesn't run during your window of execution.

You shouldn't patch the kernel

Multiple books and write-ups addressed the KPP present in iOS, each with their mix of pros and cons. I would recommend starting with the following:

- *OS X and iOS Kernel Programming - Ole Henry Halvorsen, Douglas Clarke*
- *Mac OS X Internals: A Systems Approach - Amit Singh*
- *iOS Hacker's Handbook - Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, Ralf-Philip Weinmann*
- *A Guide to Kernel Exploitation: Attacking the Core - Enrico Perla, Massimiliano Oldani*
- *MacOS and iOS Internals, Volume I - Jonathan Levin*
- *MacOS and iOS Internals, Volume III - Jonathan Levin*

- *(Update): A complete write-up for QiLin was released by its author, Jonathan Levin here: http://newosxbook.com/QiLin/qilin.pdf.*

The end-goal of getting task_for_pid() for other process' has trended away from patching the kernel to overwriting the exported pointer from AMFI to the kernel when the kernel wants to perform code signing checks or stuffing the trust cache AMFI maintains with the signatures of the binaries that aren't properly signed. For patching AMFI, we end up setting an exception handler and then overwriting this pointer with trash (to ensure a crash), our exception handler will be called and this can return to the kernel with saying every binary is properly signed (regardless if it is or isn't). This is the easier way of going about bypassing AMFI, due to the added complexity of parsing the Mach-O method and then properly inserting the hash into the AMFI trust cache.

Extracting the kernel

Unless you have updated your iPhone at just the right time to get the proper kernel and didn't delete the IPWS iTunes file from disk before it was cleaned up, several sites backup the full system IPSW, which includes the entire iOS system. The kernel resides inside of the kernel cache file, named "kernelcache.release.iphone9". The IPSW is a glorified zip file, so extraction from that is relatively painless. However, the, "kernelcache.release.iphone9" file is compressed, so extraction is performed via Willem Hengeveld's lzss decompression tool. First, the offset of the binary is discovered by inspecting the file with Binary Ninja:

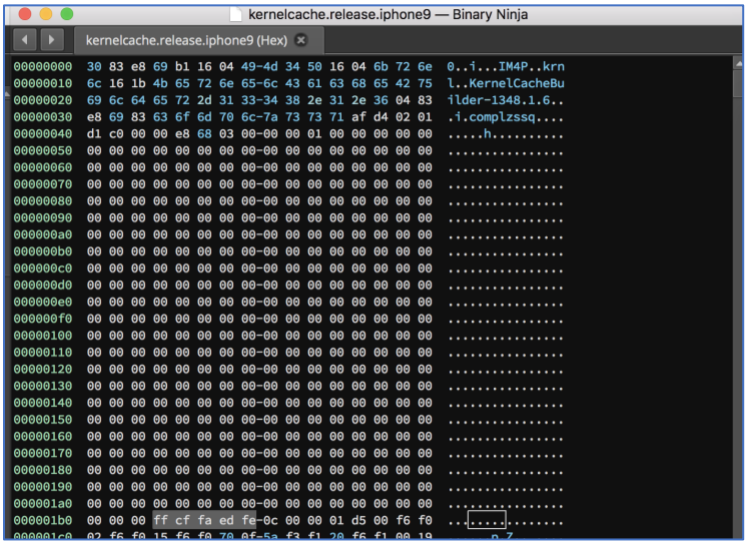


Figure 8 – Kernel cache in compressed format. Notice 0xfeedfacf (little endian) starting at 0x1b4

The offset here is 0x1b3 (including 0xff), so decompression was performed like so:

```
./lzssdec -o 0x1b3 < kernelcache.release.iphone9 > kernel.dec
```

Figure 9 – decompression with lzssdec

Then, using Binary Ninja, the kernel was disassembled (and later lifted to an Intermediate Language representation):

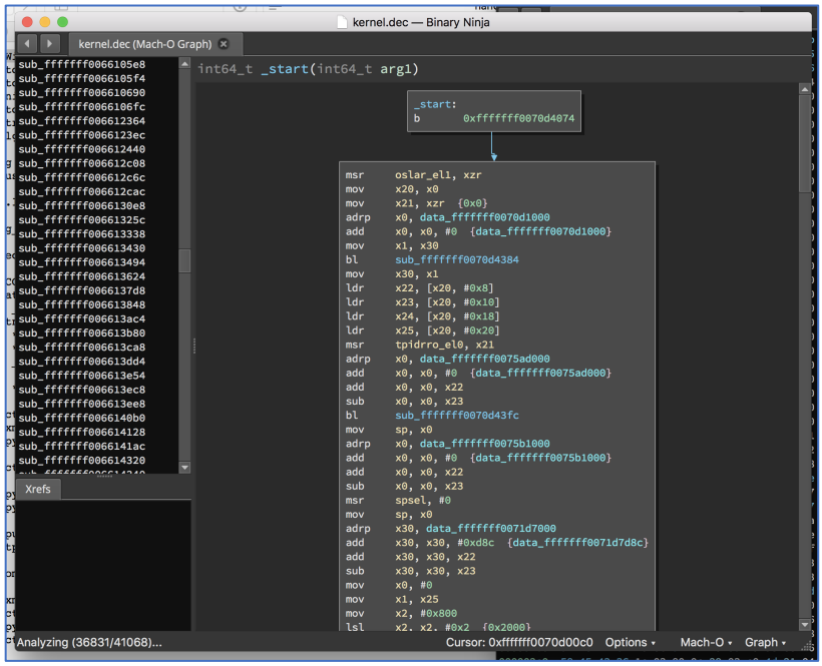


Figure 10 – Disassembling the kernel with Binary Ninja

The address where the kernel is loaded on iOS 15B202 start at 0xffffffff00760a0a0 + KASLR (Kernel Address Space Layout Randomization), which is 12 bytes of entropy. Given the page size of 2MB (0x2000 hex), there is a total of 256 slots that the kernel can start at in memory (not exactly the hardest protection to break) and iteration through these is possible without crashing the kernel due to the error handling present on the kernel read / write given by Ian Beer.

Reverse Engineering Useful Things

Having the kernel is fantastic. If we end up messing with the kernel, it's essential to know where certain interesting functions lie. There is still the issue of bypassing the Kernel Address Space Layout Randomization (KASLR), but we will cross that bridge once we get to it.

The part of the kernel that we will have to patch is task_for_pid, which is currently stopping us from getting a task port so that the process space of amfid and launchd can be modified to allow for arbitrary binaries to be run on the device. _port_name_to_task was symbolized in the kernel cache, and was used in the task_for_pid function so finding task_for_pid was simply going through the references that call _port_name_to_task, after some reverse enmgineering this was discovered:

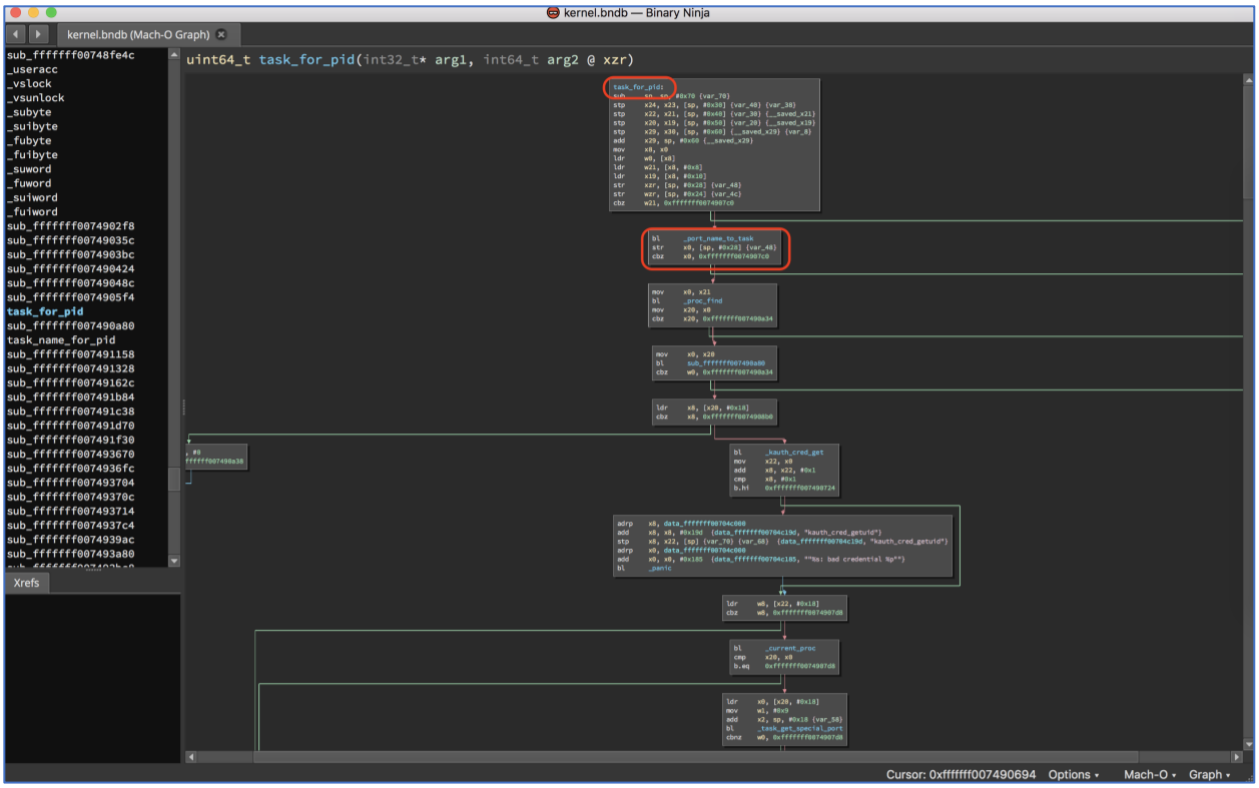


Figure 11 - task_for_pid in the kernel cache

Now, it was at this point that Jonathan Levin released his jailbreak “LiberIOS”, available here, which is based on the Qilin jailbreaking toolkit. This proved to be a fantastic resource to fact check my offsets and guide my jailbreaking efforts. Initially, I just wanted to see the internals of it to validate there was no malicious code (spoilers: I couldn’t find any):

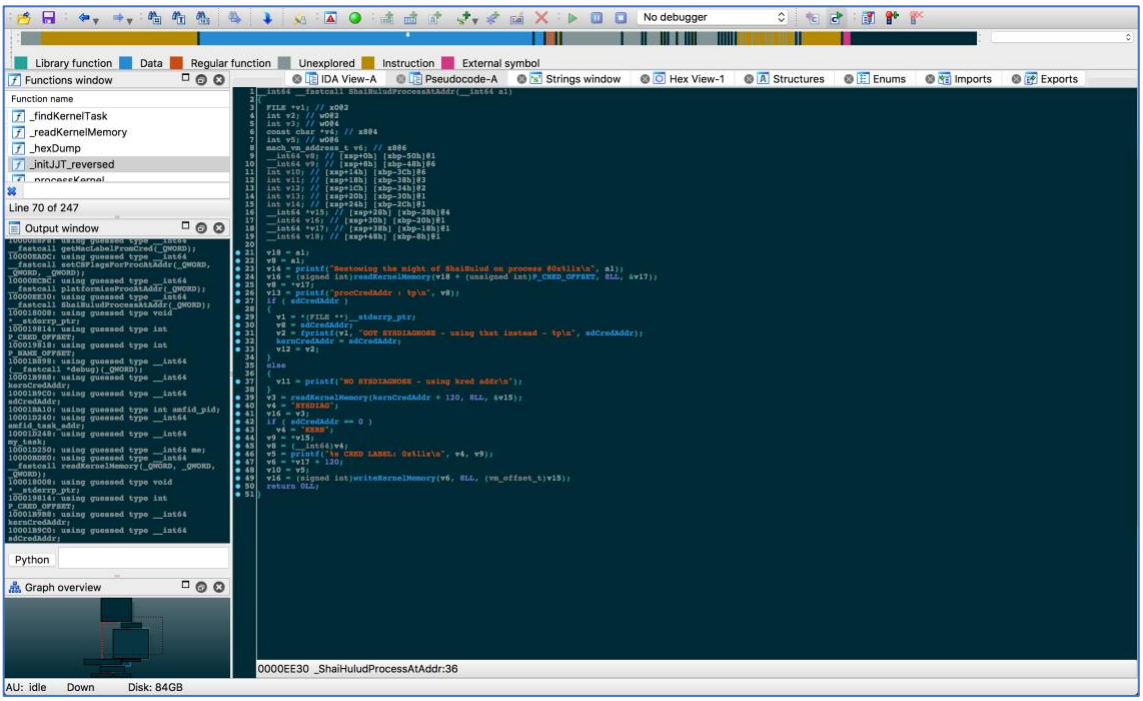


Figure 12 – Qilin decompiled with IDA

After a suitable amount of time peering at the functionality of the code, it appeared to perform as expected so I moved to testing it on a live device. It does indeed work *very* well:

Before LiberIOS Jailbreak	After LiberIOS Jailbreak
Figure 13 – LiberiOS jailbreak	*Figure 14 – LiberiOS jailbreak successful install*

And at this point it is possible to SSH into the iPhone and be presented with a root console prompt. Hooray!

```
[-bash-3.2# uname -a
Darwin nokia-388 17.2.0 Darwin Kernel Version 17.2.0: Fri Sep 29 18:14:51 PDT 2017; root:xnu-4570.20
.62~4/RELEASE_ARM64_S8000 iPhone8,1
[-bash-3.2# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(p
rocmod),20(staff),29(certusers),80(admin)
-bash-3.2#
```

Figure 15 – Successful SSH login to root on the iPhone

My next thought was how well does this map up to my perceived path of jailbreaking? Initially my thought was to patch the kernel to get TFP0, get a handle to AMFI / other significant processes, then revert the kernel quickly to avoid KPP, then using that add an exception handler to AMFI so that when the code signing check was called it would fail, trigger the exception handler, and return that everything was OK and the code could be run. After a bit of reversing, I had this as a code flow for Qilin:

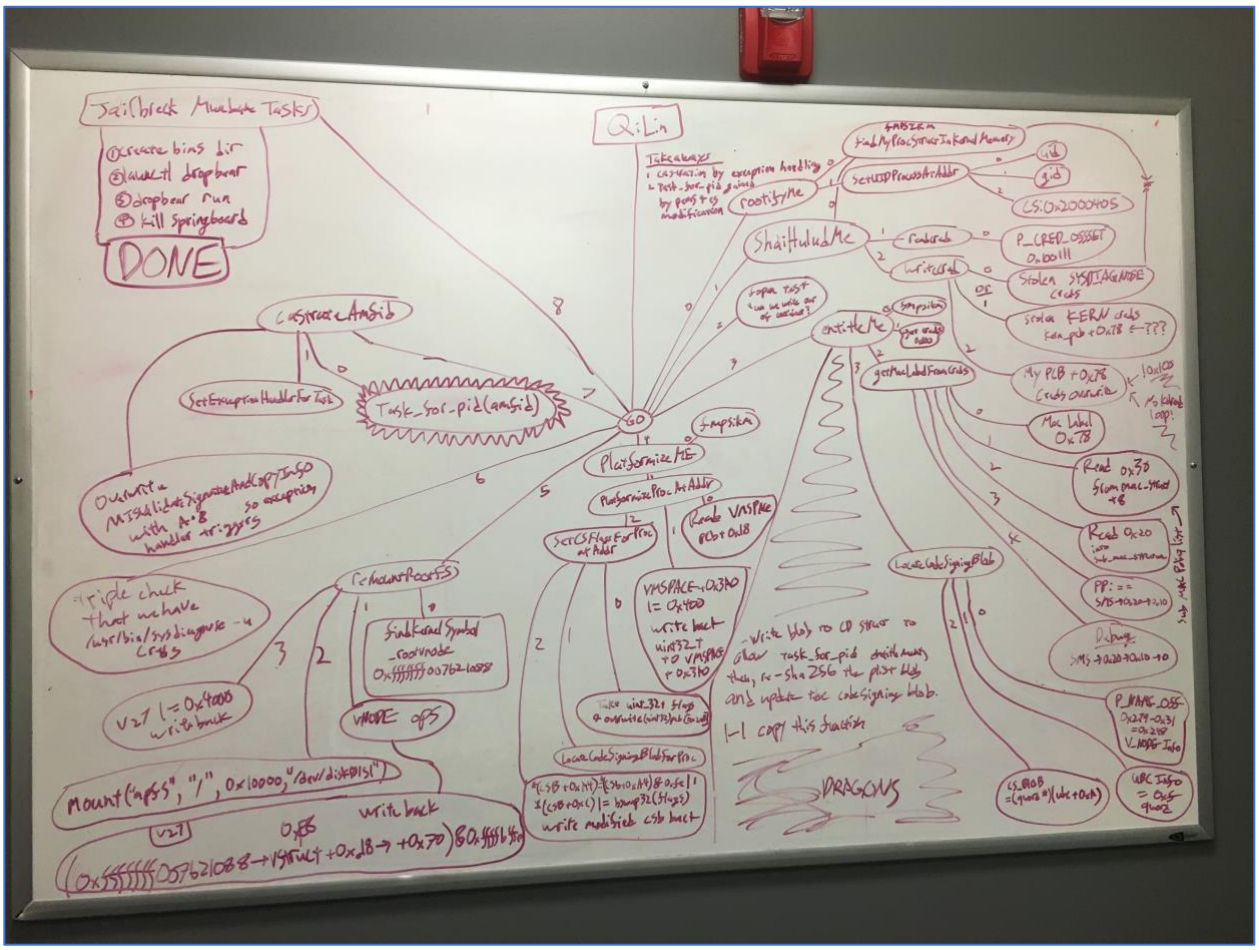


Figure 16 – The functionality of Qilin reversed

The simplicity of Qilin is beautiful:

a) Steps 1 through 7 are to get a handle to task_for_pid and disable code signing
b) Step 8 is dropping binaries on the system and running them to allow remote access.

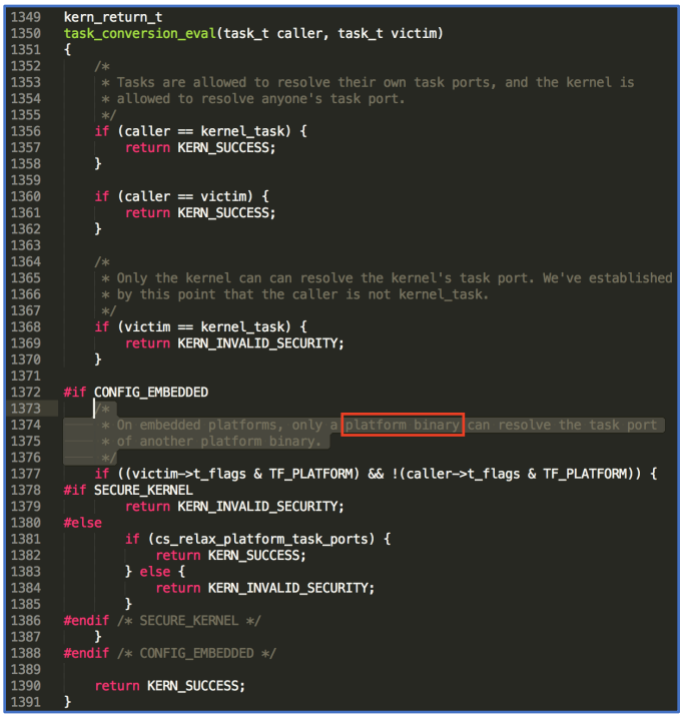
Simple, right? (notice the "Dragons" in the code entitlement section) This could not be further from the truth.

So—using this knowledge—I pulled down the Qilin kit and inserted into my jailbreak. Given the black box nature fudging around in kernels (sometimes the phone crashes *before* error messages hit the console log), and lacking a good debugging environment, Qilin served as a "known good" set of functionality as I compared the output of my functions and operations to what the Levin's "Rosetta stone" said was saying. Right off the bat I discovered that I was finding the kernel base unreliably. I was searching for 0x0100000cfeedfacf too far forward in memory, and not aligned to the proper kernel base. After fixing my search algorithm, I was able to reliably determine the base of the kernel.

Several things that pop up that I initially didn't anticipate. Instead of patching the kernel task_for_pid, Qilin properly entitles the binary via the entitleMe() function to use task_for_pid(), bypassing the need to sneak around the Kernel Patch Protection running in Exception Level 3 (ARMv8 runs untrusted user code in EL0, the iOS kernel in EL1, and KPP in EL4). This is a rather elegant work around to successfully call task_for_pid(), so instead of fighting with KPP my code will follow the same thought-path. Initially when I obtained file system access (see Figure 5 & 6) I didn't write in functionality to upload files so the fact that the root of the filesystem wasn't writeable was not apparent and would have caused several problems dropping binaries. The platformize functions in Qilin also showed that there were additional pitfalls I didn't take into account or realize was going to be a barrier.

Platform attributes

Looking at the source code for XNU, in ipc_tt.c, the following will stop us from obtaining a task port for another process:



```
1349  kern_return_t
1350  task_conversion_eval(task_t caller, task_t victim)
1351  {
1352      /*
1353       * Tasks are allowed to resolve their own task ports, and the kernel is
1354       * allowed to resolve anyone's task port.
1355       */
1356      if (caller == kernel_task) {
1357          return KERN_SUCCESS;
1358      }
1359
1360      if (caller == victim) {
1361          return KERN_SUCCESS;
1362      }
1363
1364      /*
1365       * Only the kernel can can resolve the kernel's task port. We've established
1366       * by this point that the caller is not kernel_task.
1367       */
1368      if (victim == kernel_task) {
1369          return KERN_INVALID_SECURITY;
1370      }
1371
1372  #if CONFIG_EMBEDDED
1373      /*
1374       * On embedded platforms, only a platform binary can resolve the task port
1375       * of another platform binary.
1376       */
1377      if ((victim->t_flags & TF_PLATFORM) && !(caller->t_flags & TF_PLATFORM)) {
1378  #if SECURE_KERNEL
1379          return KERN_INVALID_SECURITY;
1380  #else
1381          if (cs_relax_platform_task_ports) {
1382              return KERN_SUCCESS;
1383          } else {
1384              return KERN_INVALID_SECURITY;
1385          }
1386  #endif /* SECURE_KERNEL */
1387      }
1388  #endif /* CONFIG_EMBEDDED */
1389
1390      return KERN_SUCCESS;
1391  }
```

Figure 17 - @Siguza's response to TFP query on twitter

Therefore, our process block needs to have the proper platform attributes to bypass that check. Because of all this, platform attributes are essential for allowing our process to call task for PID, and we also end up needing to set the platform attributes for the AMFID process in order for us to get a handle to perform mach_vm_read and mach_vm_write operations.

Problems getting root

The first thing to note about stealing the root credentials is that the reference counter to the credential pointer will not be updated and so when the application closes down the kernel will panic. This can be fixed by modifying the appropriate value by +1, but there are other problems down the line that cause problems. The way that initially worked for me is parsing into the creds structure and manually setting the UID, effective UID, and saved UID to get root. After struggling with overwriting just the right bit, I went ahead and overwrote the entire cred pointer, saving my old one to be restored before program exit. This hearkens across to the Windows NT kernel where the end goal for an Escalation of Privilege (Epos) is token stealing, usually accomplished once a kernel read/write. Token stealing in NT is parsing through similar kernel structures, copying process 4's pointer (SYSTEM) to your own process to grant NT AUTHORITY\SYSTEM to your process. Initially, the system would panic when we left creds from another process in our cred pointer and exited (and this is due to reference counting in the kernel getting out of whack), so we were replacing our old credential pointer once the process was exiting. Later on, I'll explain why this stopped working for me, as this is a result of spinning up a thread to make sure AMFID ignores improperly signed binaries.

Getting access to the root filesystem

So, there's a couple of problems with the sandbox that we have to deal with, one being that the filesystem for / is not mounted in a writeable way. Normally this wouldn't be a problem, but there are other considerations with sandbox protections in iOS: we can't run binaries from /tmp, /var, /private, etc. In order to get use the root level access we need, we have to drop binaries into /jailbreak and use that as the base for all non-apple code. So, we need to remount the root filesystem as read / write. This is a problem, as the kernel enforces these protections by filtering the arguments to mount* calls. Fortunately, this isn't a new problem, and has been solved by several people. I went with Xerubs solution, as it seems very short and succinct. The code simply flips off the MNT_ROOTFS flag, remounts the drive, then flips it back on. Simple is beautiful!

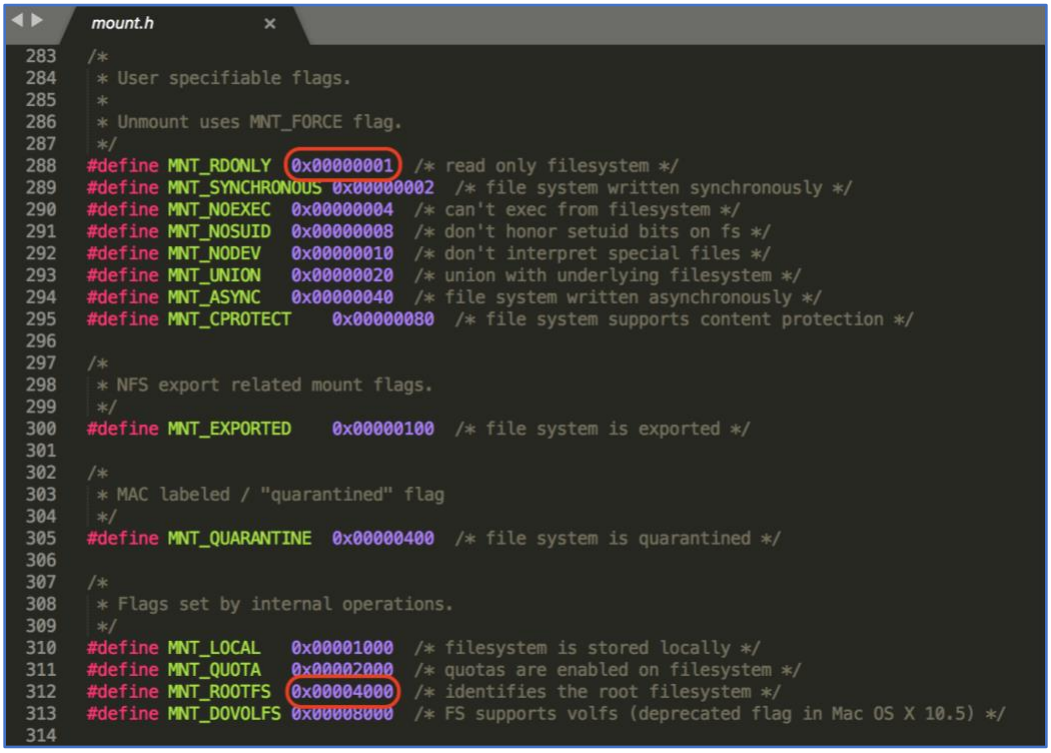


Figure 18 – We need to remove the flags and remount (darwin-xnu/bsd/sys/mount.h)

```
// Remount / as rw - patch by xerub, modified with Morpheous' symbol finding
// retrieved from: https://github.com/ninjaprawn/async_wake-fun/blob/85c32e3
    async_wake_ios/the_fun_part/fun.m
// discovered from: https://twitter.com/_argp/status/942429791520731136
void xerub_remount_code(uint64_t kaslr)
{
    //rootfs_vnode->vnode_val+0xd8->node_data->data+0x70->flags
    printf("[+]\tGot kaslr == 0x%llx\n", kaslr);
    vm_offset_t offset = 0xd8;
    uint64_t _rootvnode = kaslr + 0xffffffff00760a000 + 0x88;
    printf("[+]\tGot _rootvnode = 0x%llx\n", _rootvnode);
    uint64_t rootfs_vnode = rk64(_rootvnode);
    printf("[+]\tGot rootfs_vnode = 0x%llx\n", rootfs_vnode);
    uint64_t v_mount = rk64(rootfs_vnode + offset);
    uint32_t v_flag = rk32(v_mount + 0x70);
    printf("[+]\tv_mount=0x%llx\n"
           "[+]\tv_flag_location=0x%llx\n"
           "[+]\tv_flag_value=0x%x\n", v_mount, v_mount + 0x70, v_flag);
    //darwin-xnu/bsd/sys/mount.h
    #define MNT_RDONLY  0x00000001  /* read only filesystem */
    #define MNT_ROOTFS  0x00004000  /* identifies the root filesystem */
    printf("[+]\tSetting v_flag to 0x%x\n", v_flag & 0xFFFFBFFE);
    wk32(v_mount + 0x70, v_flag & 0xFFFFBFFE);
    char *nmz = strdup("/dev/disk0s1s1");
    int rv = mount("apfs", "/", MNT_UPDATE, (void *)&nmz);
    printf("[+]\t[fun] remounting: %d\n", rv);
    v_mount = rk64(rootfs_vnode + offset);
    wk32(v_mount + 0x70, (v_flag & 0xFFFFBFFE) | MNT_ROOTFS);
}
```

Figure 19 – We can remove the RDONLY and ROOTFS flag by AND'ing them out logically (0xFFFFBFFE)

Neutralizing Apple Mobile File Integrity Daemon

Running unsigned code will be a problem, as the kernel has several complex mechanisms that hijack running code, but the weakest link in the chain lies in the user-mode application AMIFD. When the kernel captures the request to run a binary (via execl / dylib-loading or some other mechanism), then a call is triggered to an exported function of AMFID (_MISValidateSignatureAndCopyInfo) to validate the code signature and return whether or not the code is properly signed. One neat trick that a several people have discovered is that if you register an exception handler for AMFID, then overwrite the exported pointer to a bad address, when the kernel attempts to jump to the exported MISVSACI function, the bad address will cause an exception and we can then control the state of AMFI, giving us the ability to jump in the code to the "everything OK" path and resume execution, allowing all unsigned (or badly signed) code to run as if it was blessed from Apple. Fortunately, this whole process is relatively easy, as the exception handling code was already provided by Ian Beer in his mach_portal code he released, so I was able to insert it as an exception handler and modify the offsets and change the hashing function from SHA1 to SHA256.

Task For PID Entitlements

task_for_pid is a function used as the pre-cursor for inter-process Mach-o traps like mach_vm_read and mach_vm_write, which gives us the power to read and write into other process' memory space. Ian's kernel read and write grants easy control over the kernel, but to neutralize AMFID we need to be able to write into its process space, which is managed by the

Mach-o traps. So, to hijack and bypass AMFID, so it is essential for us to read into our own entitlement blob, replace permissions, replace the SHA256 signature of the permissions with the appropriate value.

```
[!] VNODE info : 0xffffffff10acf7cf0
[!] My UBC info is 0xffffffff10a099e78
[!] My blob is here: 0xffffffff10db60300
[!] CD blob is at : 0xffffffff10bc10645 (should end with .....93d)
[!] Entitlement blob is at: 0xffffffff10bc10446
[!] blob size: 511
[!] Entitlement blob (511 bytes) @0xffffffff10bc10446: <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>application-identifier</key>
    <string>BBBRKTL9QV.com.example.async-wake-ios</string>
    <key>com.apple.developer.team-identifier</key>
    <string>BBBRKTL9QV</string>
    <key>get-task-allow</key>
    <true/>
    <key>keychain-access-groups</key>
    <array>
        <string>BBBRKTL9QV.com.example.async-wake-ios</string>
    </array>
</dict>
</plist>

[!] Here's the old entitlentment hash: [fd4a1d2211b34b9569b540968d8b52788d44f0e1ff885683b68b861fa90bda26]
[!] Hash offset: 0xfa, Type: 2 (32 bytes), nspecial = 5
[!] hash at 0xffffffff10bc1069f
[!] New blob: <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    platform-application
    task_for_pid-allow

    com.apple.system-task-ports

    com.apple.private.xpc.service-configure

</dict>
</plist>

[!] Here's the new entitlentment hash: [6c3d3666a210084fc19fce17272940719041bb3bd78bcf053ceb7d6ef59aa786]
```

Figure 20 – Replacing my old entitlements with the task_for_pid-allow entitlement ,then updating the SHA256 hash

Completing the Jailbreak

We can place relevant files on the device by including them in the XCode project. For this purpose, I am using pre-compiled binaries for iOS / ARM64 from Jonathan Levin. There is a problem signing the binaries, as XCode will attempt to sign all of them using my signature, however if I overwrite the first several bytes with junk and restore it before attempting to run, it bypasses the XCode IPA packaging check. Now, this gets annoying with a large number of binaries to perform this overwriting operation for, so as a method of obfuscating them I tarred them all up and can just drop a tar file and un-tar it. This method is derivative from the file dropping technique from LiberIOS.

Now, after completely phasing out Qilin “known good” functions completely for my own, I know have my very own jailbreak!

Before Jailbreak	Jailbreak complete

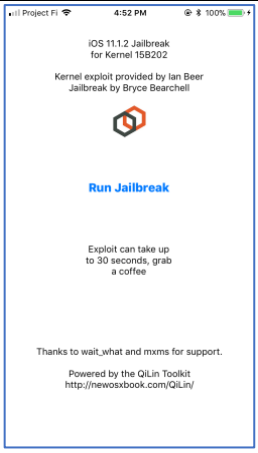
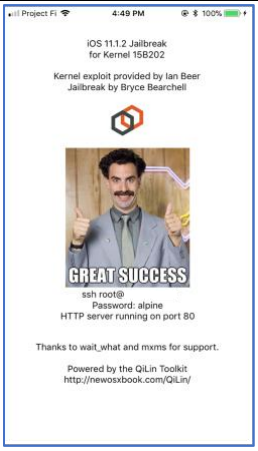

	

Figure 21 – Before the Jailbreak is run	*Figure 22 – Successful exploitation and SSH setup!*

And to validate that everything is working, I SSH'd into it:

```
$ ssh root@172.30.5.38
root@172.30.5.38's password:
root@ (/var/root)# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(p
rocmod),20(staff),29(certusers),80(admin)
root@ (/var/root)#
```

Figure 23 – Successful login as root

And, accessing the HTTP server running in another thread:

Listing [/]

Other HTTP Options: /dump_ptr=0x0011223344556677 - dump kernel memory | /info - list processes | /exit - exit HTTP server

kernel_base = 0xffffff801bc04000

./
../
.HFS+ Private Directory Data /
.file
.mb/
Applications/
Developer/
Library/
System/
bin/
cores/
dev/
etc/
jailbreak/
private/
sbin/
tmp/
usr/
var/

Figure 24 – Webserver exposing kernel memory and file system

Notes

An interesting take-away that I discovered that if you disable updates, several built-in apps (TV, Reminders, Notes, Podcasts, Compass, etc.) will not install. You can fix this by *temporarily* enabling updates and installing them. Once you've finished, disable them again!

You can add the following to /etc/hosts to block the resolution of the Apple update service:

```
127.0.0.1 mesu.apple.com
```

And to disable it (enable updates / built-in app installs):

```
sed '$ d' /etc/hosts > /tmp/hosts_new
mv /tmp/hosts_new /etc/hosts
```

Updates are stored at:

- /var/MobileAsset/Assets/com_apple_MobileAsset_SoftwareUpdate/
- /var/MobileAsset/Assets/com_apple_MobileAsset_SoftwareUpdateDocmentation/

After I got the jailbreak working, everything would be fine until I hit the menu button, then the kernel would panic. I couldn't figure this crash out for a weekend, but after sacrificing enough qwords to the binary gods I discovered that I was improperly rebuilding my entitlement section. Ugh! After some wrangling I got it working, but I still am generating entitlement errors—however this doesn't impact the jailbreak (arbitrary code still runs, etc.). If you happen to know the root cause, please let me know.

I also noticed that after properly replacing my process credentials at the end of execution effectively cut off the access from the error handler thread and the process it was supposed to be handling, AMFID. There were two ways I saw of getting around this issue: first was to edit the error handler's thread credential pointer to allow it access, second was to never end up reverting our process credentials so that we maintain the high level of access needed. A better solution for this will be implemented soon in an update to the overall jailbreak, including additional kernel versions.

One thing to note, if AMFID is restarted you will lose the ability to run unsigned code. I'm not sure how launchd controls the process, but XCode will restart AMFID if it encounters an error launching, so be aware of that. Also, suspending the thread that is handling errors for AMFI will stop all code from passing signing checks. LiberIOS solved this problem by running a dedicated process outside of the app to provide constant error handling and to check (and replace the error handler if need be) if AMFID has been restarted.

TL DR; How to jailbreak your phone

Here's the steps to jailbreak your device, but please remember that this is for **iOS 11.1.2 (15B202) only**. If you have higher or equal to 11.0 and *under* 11.2, **I would highly recommend**

the LiberIOS jailbreak here. For iOS 10 devices, both Pangu8 and Yalu support iOS 10, although I cannot vouch to their code, as I haven't looked at it. The IPA for my jailbreak is here.

Loading the Jailbreak onto your phone

1. Build and Load Jailbreak using XCode
2. Side load using Cydia Impactor
    a. Obtain Cydia Impactor
    b. Obtain an App-Specific password
    c. Use Impactor to side load the IPA
3. Side load using Apple Configurator 2
4. Win!

Build and Load Jailbreak using XCode

The most reliable method of loading the jailbreak is to load the source code into XCode, select your iPhone as the target, and hit run! XCode will take care of properly building, signing, and loading the app onto your phone, and this method allows you to see all the debugging information associated with the jailbreak.

Side Load using Cydia Impactor

Impactor is a fantastic tool developed by Saurik and can be found at http://www.cydiaimpactor.com.

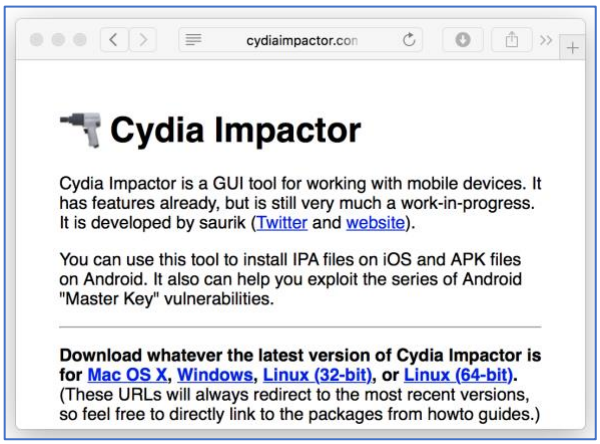


Figure 25 – Cydia Impactor

Obtain an App-Specific password

A temporary password for your iOS account can be obtained from https://appleid.apple.com/ so that the IPA can be properly side loaded (and signed) by Apple. Note: Apple requires 2-factor authentication in order to create an app-specific password, if you don't have it set up you won't see the option to generate it.

Figure 26 – The Apple ID login page

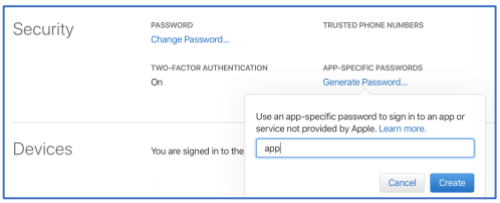


Figure 27 – Type a random name to generate an app specific password

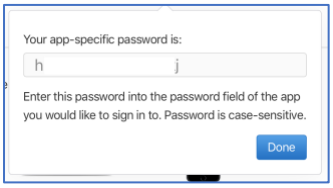


Figure 28 – Here is the (censored) output from the password generation process

Use Impactor to side load the IPA

Simply drag the IPA onto Impactor, type in your username and the previously generated password, and the app is loaded!

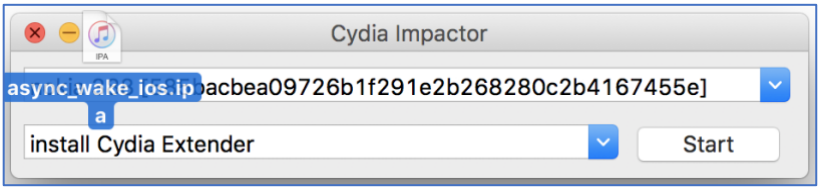


Figure 29 – Just click and drag to load IPA

Side load using Apple Configurator 2

Apple configurator is obtainable from the App store on OSX. Once installed, you can simply right-click on your device, select Add, then Apps, then load an app from disk. Selecting “Choose from my Mac” will allow you to load an arbitrary IPA. This is probably the easiest method of side loading the jailbreak:

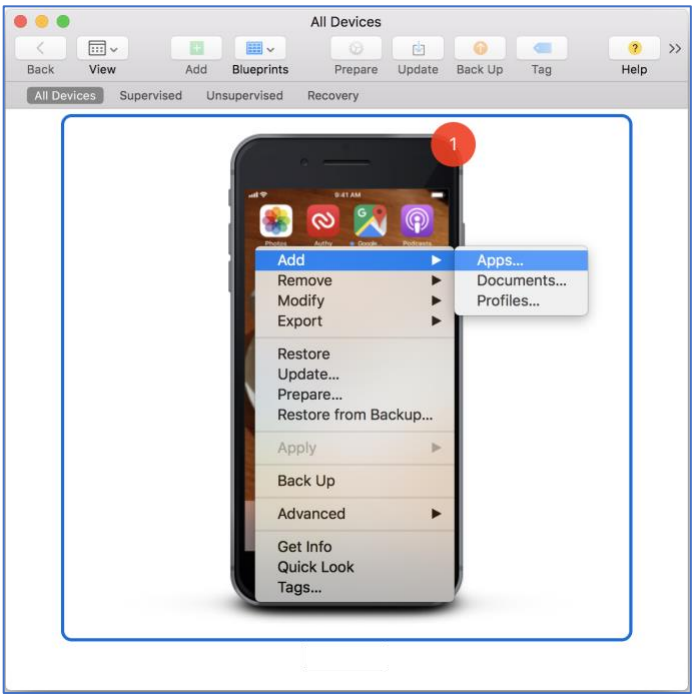


Figure 30 – Apple Configurator 2 side loading Jailbreak

Win!!!

Simply tap on the app to start it up, then tap on "Run Jailbreak" to run the exploit. Once the jailbreak completes, you can SSH to your phone using the username "root" and the password "alpine" to get root access!

Also, the webserver is running on port 80, it has complete access to the filesystem for easy browsing. Included is a link to the kernel base / process structures, so you can peruse the running kernel memory in an easy fashion!

Final Thoughts

As a kinetic learner, this was an amazing experience for me to go through and fall in all the pitfalls and traps of making a jailbreak for iOS. I would highly recommend this process if that is how you learn, as making the mistakes and errors firsthand is very enlightening. It appears that for future projects, once an exploit gains tfp0, most jailbreaking efforts will be abstracted away with jailbreak toolkits (like QiLin) that perform standard operations and have a wide range of supported devices.

Credits

This jailbreak could not have been born without the magnanimously provided kernel read / write by Ian Beer. If you're at DefCon I'll buy you a beer^H^H^H^H whiskey!

Thank you Coalfire for supporting me in this journey and enabling and encouraging me along the way: Ryan Jones for giving me research time, Marcello Salvati for initiating the whole process and providing the dank memes.

Thanks, wait_what for pointing me in the right direction.

Mxms, you gave a random stranger on Freenode hope and great advice. Thanks for keeping my spirits up!

Jonathan Levin for providing a stellar (amazing (mind blowing)) jailbreak for me to learn from, and fact check my assumptions. I was so off man, for like a week I was finding random binaries in memory and thinking they were the kernel cache and raging when my gadgets failed or pointers wouldn't go where I expected, when all that needed to be done was brute force back to get the kernel base. There is beauty in simplicity. My implementation of neutralizing AMFI is a bastardization of a number of people's code and Sulphur-smelling runes, yours is so nice. Cheers man, you the real MVP.

Props to Brad Conte for his SHA256 and SHA1 code, and thank you Logan Evans for sending them to me!

Last (but not least), thank you Rusty and Jordan for making Binary Ninja, a most excellent tool for reverse engineering. Hi Brian & everyone from Vector35!

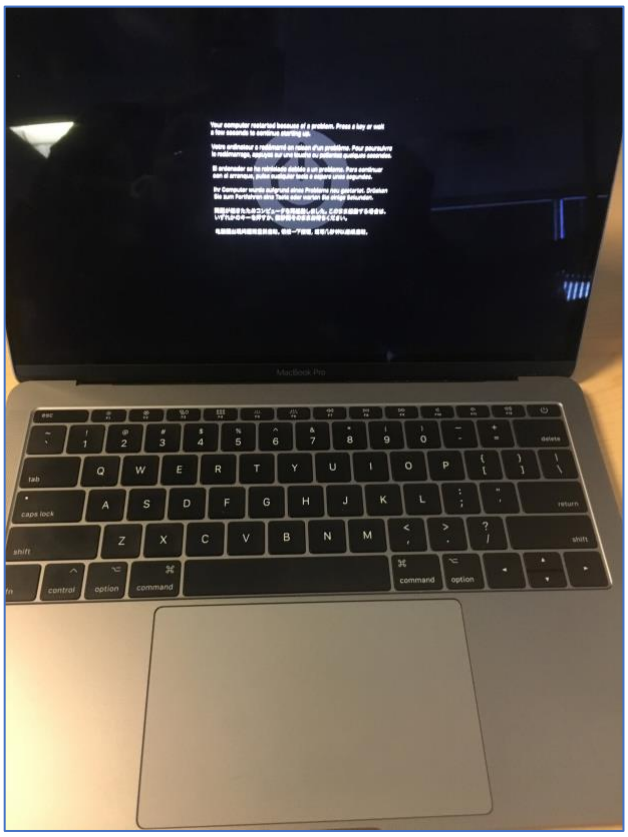


Figure 31 – Learning is fun, thanks for Coalfire not firing me while I did all this!