*OS: iBoot

iBoot is the collective name given to Apple's boot loader components, and sometimes to the particular second stage boot loader. These components form the boot chain of iOS, starting with the SecureROM, and (during normal boot flow) ending at iBoot, which loads the kernelcache, unless any part of the process fails, in which case the device is put into recovery mode.

Although Apple claimed to have stopped encrypting iOS images "to help researchers" and the iBoot 32-bit images were indeed plaintext as of iOS 10, those of iBoot 64-bit remain encrypted with the device-specific (GID) keys to this day. 64-bit images have several important differences than those of 32 (notably, coprocessor and KPP support), and 32-bit images are no longer relevant as of iOS 11 anyway. This has not stopped researchers, who find ways to publish the keys and IVs (initialization vectors) required to decrypt the iBoot 64 images. These keys were scarce for the longest time, until multiple ones for all 12.x versions started appearing on the iPhone Wiki. Although of dubious origin (an iBoot-level exploit or development-fused devices), they are nonetheless welcome, as they provide visibility onto this important codebase, which forms the fulcrum for all of *OS security.

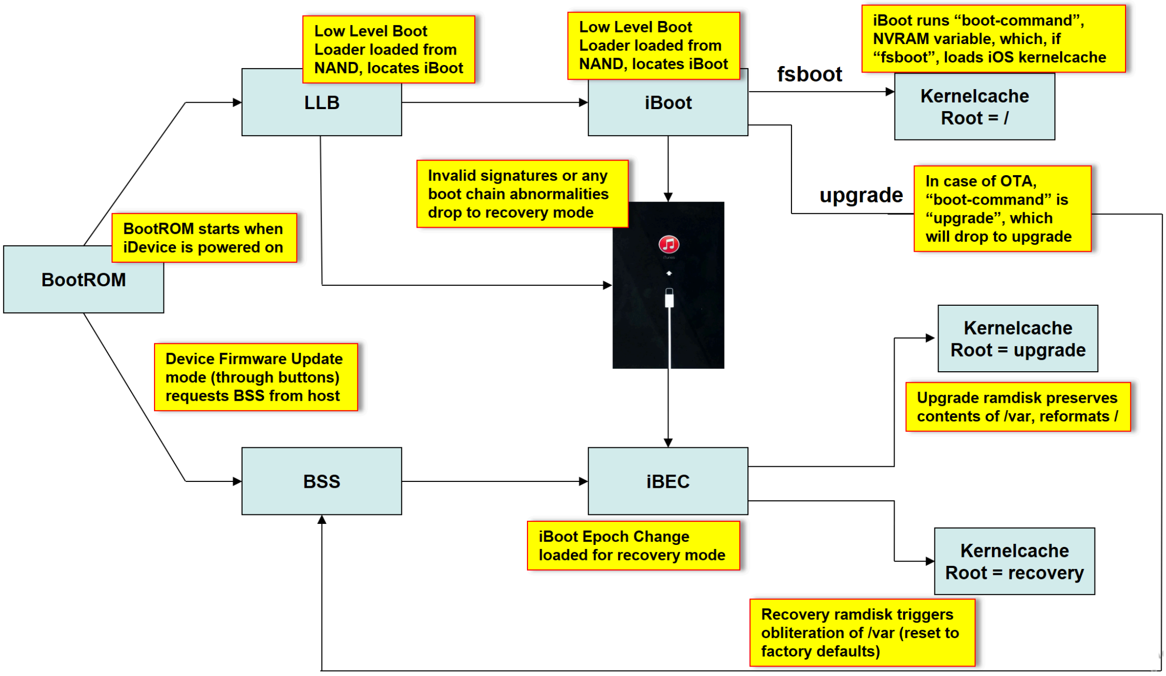
The iBoot sources were also leaked from Apple - making a public appearance on GitHub in mid February 2018 (with apparent links posted even earlier on Reddit). Although the exact iBoot version (which is provided as an external `#define`) is hard to determine, the latest board support - t8010 as a simulator/FPGA - indicates iPhone 6S and along with the copyright (2015) is thus likely no later than iBoot-2817, i.e. early iOS 9.

This work cannot show code snippets from the iBoot sources, as those were leaked illegally and are copyrighted by Apple. Apple quickly invoked the DMCA in an effort to contain the leaked sources, but this proved futile, as countless mirrors can still be found - which the interested reader might want to locate. These pages, instead, demonstrate disassembly from the publicly available decrypted image of iBoot, thanks to iOS 13 beta keys posted anonymously to the NewOSXBook.com Forum[ibk].

The Boot Chain

The boot chain of *OS refers to the sequence of steps carried out by the application processor, starting with the loading of the SecureROM image, and ending with the loading of the XNU kernelcache. This is considered a chain, since it is comprised of distinct stages, wherein each stage is charged with its own various initialization tasks, as well as locating and loading the next stage. Figure xx-ibootchain shows the traditional boot chain of *OS.

Figure xx-ibootchain: The traditional boot chain of *OS


Low Level Boot Loader loaded from NAND, locates iBoot
Low Level Boot Loader loaded from NAND, locates iBoot
iBoot runs "boot-command", NVRAM variable, which, if "fsboot", loads iOS kernelcache
LLB
iBoot
fsboot
Kernelcache Root = /
Invalid signatures or any boot chain abnormalities drop to recovery mode
upgrade
In case of OTA, "boot-command" is "upgrade", which will drop to upgrade
BootROM starts when iDevice is powered on
BootROM
Kernelcache Root = upgrade
Upgrade ramdisk preserves contents of /var, reformats /
Device Firmware Update mode (through buttons) requests BSS from host
BSS
iBEC
iBoot Epoch Change loaded for recovery mode
Kernelcache Root = recovery
Recovery ramdisk triggers obliteration of /var (reset to factory defaults)

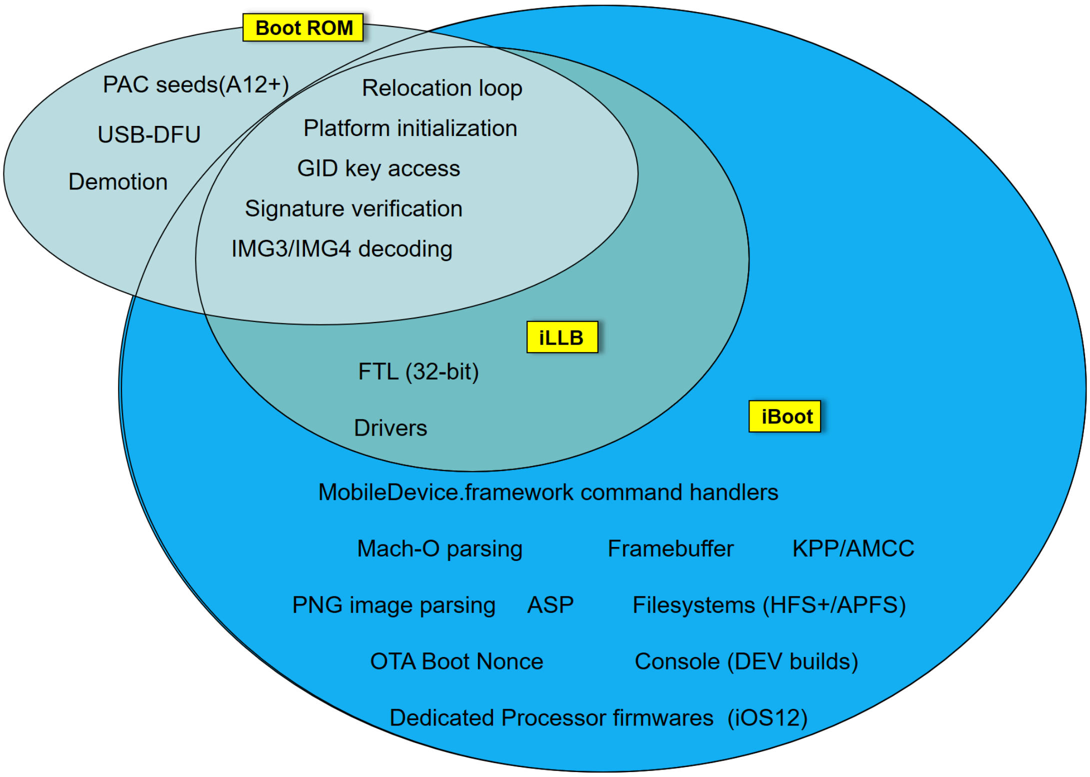
Common Code

Dumping the boot components reveals a very large amount of common code, and the eventual leak of the sources corroborated this, by showing that the components are all built from the same source base. All components are known to contain the following:

- Rebase loop: which checks if the code has been loaded at a predefined virtual address (e.g. 0x100000000 for the t8020 ROM, or 0x19c030000 for iBoot-5540). If not, the code is moved by copying using `LDP`/`STP` instructions and X3, X4.
- Platform bootstrapping: CPU initialization, Setting up the Vector Base Address Register, the MMU and other ARM MSRs.
- Task management, allowing multithreading. This is primarily used by the the second stage iBoot, but even the SecureROM has support for spinning a USB task, when in DFU mode.
- IMG3/IMG4 code: 32-bit iDevices (excluding watches) used the proprietary IMG3, and 64-bit images (and watched) use IMG4 (a form of DER) to encode components - binaries and graphic images.
- Certificate handling: including both an X.509v3 hard-coded certificate, and the DER-code to parse and verify images signed with its public key.

The later components in the chain have increasing functionality, culminating with iBoot's ability of reading file systems, Mach-O objects, and console support. Conceptually, this could be visualized as follows:

Figure xx-ibootchain: The traditional boot chain of *OS



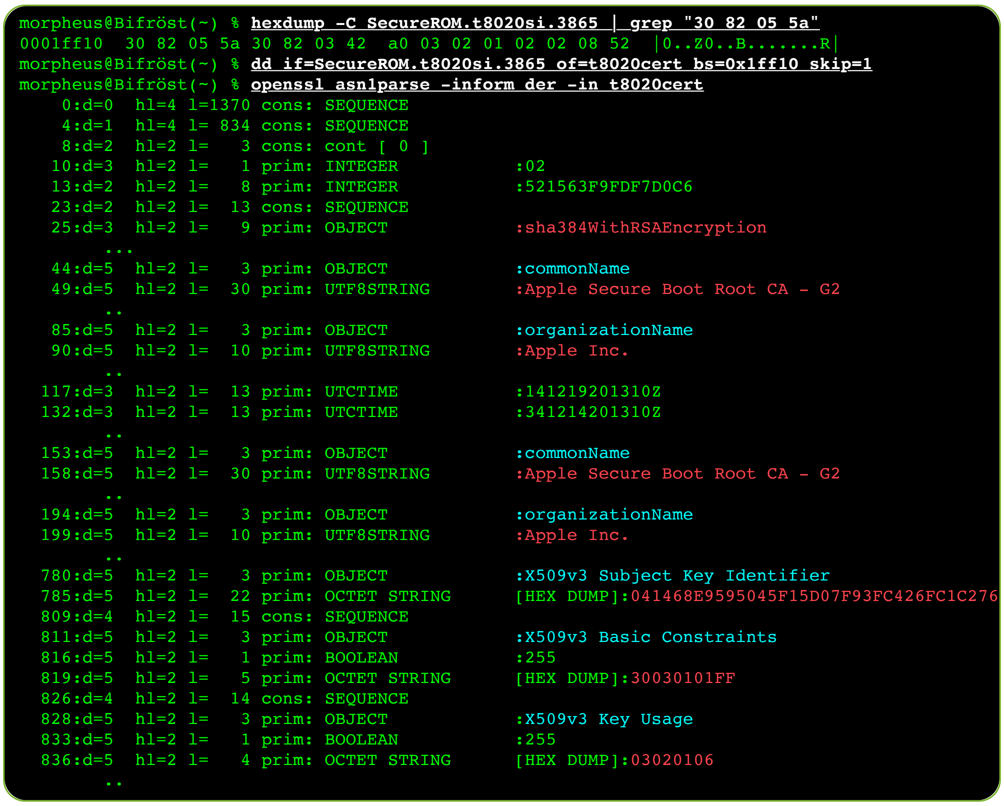
The reason for using four different components (LLB, iBoot, iBEC and iBSS) with so much common code was likely limitations of SRAM, and possibly security considerations (e.g. due to DMG loading in iBEC). Newer devices (and thus, SecureROM versions) still have four separate images, but inspecting them (past decryption) reveal they are all identical. In other words, one single iBoot image now handles all phases. LLB is no longer required, and thus the SecureROM (which still shares its code with iBoot) directly loads the iBoot image, which is still tagged as 'illb', rather than 'ibot'.

The Chain of Trust

An important aspect of the boot chain is its resulting **chain of trust**. The trust begins with a hard-coded root certificate, whose bytes are burnt into the ROM. Normally, root certificates pose a problem to implicitly trust, but because the certificates are in a read-only memory component, they cannot be modified in any way, and can thus kick off the chain. In this way, the ROM certificate can vouch for the integrity of the iBoot components, and it is iBoot, in turn, which verifies the digital signature (and APTicket entry) and integrity of the kernelcache with the certificate embedded in its image.

The hard coded and embedded certificates are easy to extract, given a ROM dump or decrypted image. They are DER encoded, and can thus be found by looking for the "30 82 05 5a" encoding a `SEQUENCE` of 1,370 bytes. Output xx-bootcert shows the certificate from a SecureROM dump, but could be applied on any iBoot image:

Output xx-bootcert: Locating the Apple Secure Boot CA in a SecureROM image

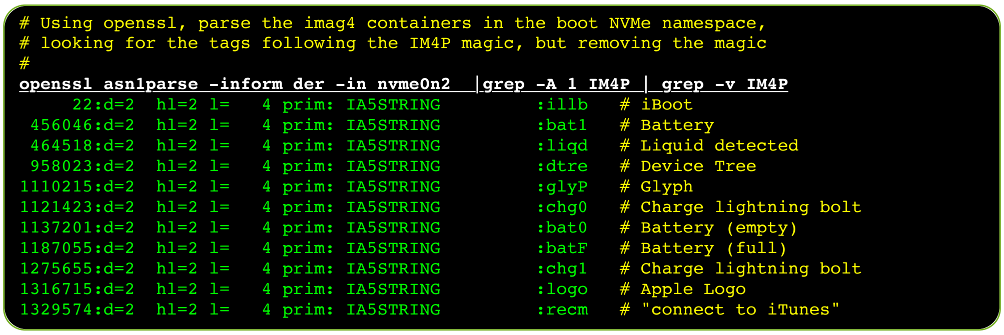


```
morpheus@Bifröst(~) % hexdump -C SecureROM.t8020si.3865 | grep "30 82 05 5a"
0001ff10  30 82 05 5a 30 82 03 42  a0 03 02 01 02 02 08 52  |0..Z0..B.......R|
morpheus@Bifröst(~) % dd if=SecureROM.t8020si.3865 of=t8020cert bs=0x1ff10 skip=1
morpheus@Bifröst(~) % openssl asn1parse -inform der -in t8020cert
    0:d=0  hl=4 l=1370 cons: SEQUENCE
    4:d=1  hl=4 l= 834 cons: SEQUENCE
    8:d=2  hl=2 l=   3 cons: cont [ 0 ]
   10:d=3  hl=2 l=   1 prim: INTEGER           :02
   13:d=2  hl=2 l=   8 prim: INTEGER           :521563F9FDF7D0C6
   23:d=2  hl=2 l=  13 cons: SEQUENCE
   25:d=3  hl=2 l=   9 prim: OBJECT            :sha384WithRSAEncryption
        ...
   44:d=5  hl=2 l=   3 prim: OBJECT            :commonName
   49:d=5  hl=2 l=  30 prim: UTF8STRING        :Apple Secure Boot Root CA - G2
        ..
   85:d=5  hl=2 l=   3 prim: OBJECT            :organizationName
   90:d=5  hl=2 l=  10 prim: UTF8STRING        :Apple Inc.
        ..
  117:d=3  hl=2 l=  13 prim: UTCTIME           :141219201310Z
  132:d=3  hl=2 l=  13 prim: UTCTIME           :341214201310Z
        ..
  153:d=5  hl=2 l=   3 prim: OBJECT            :commonName
  158:d=5  hl=2 l=  30 prim: UTF8STRING        :Apple Secure Boot Root CA - G2
        ..
  194:d=5  hl=2 l=   3 prim: OBJECT            :organizationName
  199:d=5  hl=2 l=  10 prim: UTF8STRING        :Apple Inc.
        ..
  780:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Subject Key Identifier
  785:d=5  hl=2 l=  22 prim: OCTET STRING      [HEX DUMP]:041468E9595045F15D07F93FC426FC1C276
  809:d=4  hl=2 l=  15 cons: SEQUENCE
  811:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Basic Constraints
  816:d=5  hl=2 l=   1 prim: BOOLEAN           :255
  819:d=5  hl=2 l=   5 prim: OCTET STRING      [HEX DUMP]:30030101FF
  826:d=4  hl=2 l=  14 cons: SEQUENCE
  828:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Key Usage
  833:d=5  hl=2 l=   1 prim: BOOLEAN           :255
  836:d=5  hl=2 l=   4 prim: OCTET STRING      [HEX DUMP]:03020106
        ..
```

The Boot Partition

The updatable boot components of *OS are copied from their respective files in the .ipsw (from firmware/all_flash) to the `boot` partition of the iDevice (now the second NVMe namespace), wherein they are stored in a large IMG4 container along with the APTicket. In this way, iBoot can load them by their four letter tag. Images are stored in LZSS compressed ARGB form (and as of A11, illb itself is compressed, as well). The device tree is stored in raw form (as shown in Listing xx-dtim4p), and iBoot/LLB is stored encrypted by the device's GID key.

Output xx-nvmeboot: The contents of the `boot` NVMe namespace



```
# Using openssl, parse the imag4 containers in the boot NVMe namespace,
# looking for the tags following the IM4P magic, but removing the magic
#
openssl asn1parse -inform der -in nvme0n2  |grep -A 1 IM4P | grep -v IM4P
     22:d=2  hl=2 l=   4 prim: IA5STRING         :illb   # iBoot
 456046:d=2  hl=2 l=   4 prim: IA5STRING         :batl   # Battery
 464518:d=2  hl=2 l=   4 prim: IA5STRING         :liqd   # Liquid detected
 958023:d=2  hl=2 l=   4 prim: IA5STRING         :dtre   # Device Tree
1110215:d=2  hl=2 l=   4 prim: IA5STRING         :glyP   # Glyph
1121423:d=2  hl=2 l=   4 prim: IA5STRING         :chg0   # Charge lightning bolt
1137201:d=2  hl=2 l=   4 prim: IA5STRING         :bat0   # Battery (empty)
1187055:d=2  hl=2 l=   4 prim: IA5STRING         :batF   # Battery (full)
1275655:d=2  hl=2 l=   4 prim: IA5STRING         :chg1   # Charge lightning bolt
1316715:d=2  hl=2 l=   4 prim: IA5STRING         :logo   # Apple Logo
1329574:d=2  hl=2 l=   4 prim: IA5STRING         :recm   # "connect to iTunes"
```

The SecureROM

When power is first supplied to the system, the first code to execute on the application processor is that of the boot ROM, referred to by Apple as "SecureROM". The code of the SecureROM is a subset of the iBoot source code, but unlike it cannot be updated. As its name implies, the SecureROM is a **Read Only Memory** component, which - for better or for worse - cannot be modified after fabrication by anyone, not even Apple. The version of the SecureROM therefore matches the iBoot version at the time of the device release (e.g. iBoot-2696.0.0.133 for A10, or iBoot-4479.0.0.100.4 for the A13). This version is reported to the host as part of the serial number when the device is in DFU mode, and is visible in the IORegistry.

Other than that identification, the SecureROM remains largely invisible, as access to it is disabled when loading the next stages. Nonetheless, the first public demonstration of SecureROM dumping was demonstrated by Ramtin Amin, who ingeniously devised a method to perform a Man-in-the-Middle attack against the system's PCIe. This landmark achievement, however, faded away once the ROM source code leaked along with the rest of the iBoot sources. JTaggable ("dev-fused") iDevices can also be made to dump the SecureROM, as can a SecureROM exploit such as @axi0mX's Checkm8, discussed later.

The SecureROM is very small - about 150,320 or so bytes in the T8020 - and is therefore limited in the amount of code it contains. It presently consists of the following routines:

- Platform startup: The code starts with a call to a board specific platform startup, which sets board specific registers.
- Relocation Loop: A check to see if it is already based at its desired address in memory. If not, it relocates itself to that address using the relocation loop described earlier.
- continue start: which sets up the address of the main as its return (i.e. in `LR`), then proceeds to low level initializations, which include setting the `VBAR_EL1`, exception, interrupt and bootstrap stacks, page tables, and a heap guard page, before returning into the main, which proceeds according to the following stages:
- CPU initialization: Initializes the MMU, and sets the value of the System Control Register, `SCTLR`. The bits set are `M` (#0, enabling MMU translation), `SA` (#3, SP alignment check), `C` and `I` (#2 and #12, for data and instruction cache), and `WXN` (#19, for Write-eXecute-Never).
- Initialization of clocks, the SRAM bus, and GPIO pins.
- On A12, the SecureROM sets the ARMv8.3 Pointer Authentication Code (PAC) random seeds.
- Miscellaneous initializations: Consisting of clock, internal memory and GPIO pins.
- System initialization: Consisting of setting up the heap and task subsystem.
- Platform early initialization: An i-Device specific initialization routine, branching to other routines for power management, board and chip specific initialization, UART, etc.
- Checking the force DFU pin: to determine if Device Firmware Update is forced.
- Platform late initialization: Another i-Device specific routine, branching to other board component initialization routines.
- Obtain the boot device and selected boot configuration. This will normally load the next stage (marked by the '`illb`' tag), but if the stage cannot be loaded (or if DFU is forced), it will drop to DFU.

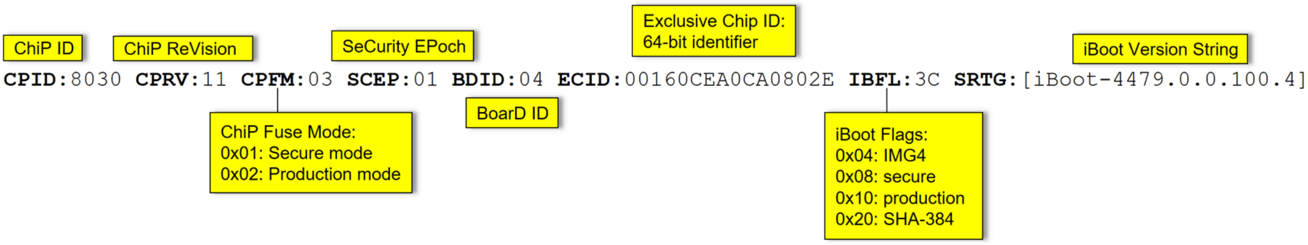
Once loaded in memory, booting an image is performed by preparing and jumping, using a function which takes the boot image type, load address, and argument pointer. The function quiesces the hardware and CPU, disables interrupts, and transfers control by calling the load address as a function (optionally, through a trampoline). This is expected to never return.

DFU mode

If the boot process cannot continue past the SecureROM (for example, if the next stage cannot be loaded), the boot chain cannot be trusted, and the device is effectively bricked. To allow recovery, the SecureROM contains code to implement a fallback to **Device Firmware Update**(DFU) mode, as a failsafe for recovery when the boot chain fails. This mode can also be requested by the GPIO keys (i.e. iDevice buttons) as documented in the iPhone Wiki[dfu].

DFU is a well documented USB standard[usbdfu]. The SecureROM starts a `usb` task, which emits the iDevice identifier ("Apple Mobile Device (DFU Mode)") a long string as a serial number, consisting of entries as shown in Figure xx-usbdesc. When the phone is booted normally, the CPID and the ECID concatenated together make up the actual serial number.

Figure xx-usbdesc: A DFU emitted serial number string



The SecureROM then awaits a DFU download request, through which it expects an iBSS (recovery mode LLB/iBoot) image. The image is subject to signature and APTicket verification. If verified, control is transferred and the image is booted.

Demotion

Retail devices are hard fused with a CPFM of 03 (i.e. 00000011b), indicating secure and production, as shown in Figure xx-usbdesc. The highly sought after (and often illegally obtained) "dev-fused" devices have CPFM of 01 (secure, development), or 00 (insecure, development). The fusing is performed by hardware, so once the device is fabricated, it cannot be changed. The SecureROM is responsible for loading the fuse values during early startup, and enforcing security.

Code execution at the SecureROM level allows **demotion**, which enables overriding the CPFM bits from the fused value. Although the fuses are immutable, they are loaded into a memory mapped register, whose value *can* be changed. Overriding the CPFM is thus simply a matter of writing to that memory mapped address. Note, that the demotion does not actually change the state of the fuses, and therefore is only temporary, persisting until the next reboot.

Demoting a device to 00 enables the **JTAG**. The **J**oint **T**est **A**ccess **G**roup standard for testing devices (more formally known as IEEE Standard 1149.1-1990), is a powerful testing mechanism which consists of a test access port (TAP), and a set of debug registers. JTAG's capabilities are nigh omnipotent. Using JTAG, one can directly the debug the Application Processor at any time. While this will probably still not allow retrieval of the UID/GID keys (as those are maintained by an AES coprocessor), it would still mean that execution could be controlled at any time, allowing usage of the keys for encryption and decryption (chosen ciphertext/plaintext attacks). Execution could be stopped and inspected at any stage of the boot chain – from as early as the ROM to the kernel – and the bootchain trust could easily be compromised, allowing any iOS configuration – jailbroken or other – one would see fit.

A special proprietary Apple cable with a symian code name ("Kong" or "Kanzi") is required to use JTAG on an iDevice. Additionally, proprietry software called "Astris" (traces of which can be seen in the *OS filesystem) handles the transport to allow a debugger interface. Nonpareil hardware hacker Ramtin Amin has managed to reproduce its function with a "Bonobo" cable, available through his company's website[lc], and compatible with OpenOCD.

iBoot (the Second Stage Bootloader)

The second stage boot loader, referred to internally as `iBootStage2`, is also the main one. It starts with a relocation loop to move the image to a specific virtual address, if not already there. The address changes between releases - in iOS 12 it is 0x1800b0000, and in iOS 13 is `0x19c030000`. The address is clearly visible in the relocation loop, as shown in Figure xx-ibr:

Listing xx-ibr: The iBoot relocation loop

```
#
# Get loaded address (here, 0 because disarm assumes image was mapped there)
# and compare to fixed address (stored at 0x318, here 0x19c030000).
#
_start:
0x00000000      ADRP X0, 0        ; X0 = 0x0..
0x00000004      ADD X0, X0, #0    ; X0 = X0 = actual load address
0x00000008      LDR X1, #784      ; X1 = *(0x318) = 0x19c030000 = desired load address
0x0000000c      BL 0x1b5e8        ; = platform_start(0x0, 0x19c030000)..
0x00000010      CMP X1, X0, ...   ; ..
0x00000014      B.EQ 0x60         ; if equal, _continue_start
0x00000018      LDR LR, #48       ; X30 = *(0x48) = 0x19c030050 else, set desired address
0x0000001c      LDR X2, #772      ; X2 = *(0x320) = 0x19c1a33c0 (end of image)
0x00000020      SUB X2, X2, X1    ; X2 = end – beginning = image len
0x00000024      MOV_R X5, X0      ; X5 = X0 (0x0)..
0x00000028      MOV_R X6, X2      ; X6 = X2 (0x19c1a33c0)..
0x0000002c      MOV_R X7, X1      ; X7 = X1 (0x19c030000)..
copy loop:
0x00000030      LDP X3, X4, [X0], #0x10         ; [X0, X0] = *[X0]..
0x00000034      LDP X3, X4, [X1], #0x10         ; [X0, X0] = *[X0]..
0x00000038      SUBS X2, X2, #16 ;
0x0000003c      B.NE 0x30         ;  copy loop
0x00000040      RET               ;  Goto LR
0x00000044      NOP               ;
0x00000048      DCQ 0x19c030050   ;
0x00000050      LDP XZR, XZR, [X5], #0x10       ; [X0, X0] = *[X0]..
0x00000054      SUBS X6, X6, #16,; ..
0x00000058      B.NE 0x50         ;
0x0000005c      BR X7             ; jump to 0x19c030000 (relocated)
_continue_start:
0x00000060      MSR DAIFSet, #15        ; D=1 A=1 I=1 F=1 ..
```

Following the relocation, startup continues with miscellaneous initializations (`VBAR_EL1` to 0x19c031000, etc), before setting the `LR` value to the generic main function (`0x19c03340c` in the 13β1 iBoot image), and then returning to it. This function, in turn, performs much of the initialization already discussed in the SecureROM section (notably, setting up the default environment), and starts up the `main` task (`0x19c0337d4`, using a thread starting function (`19c06b584`).

The `main` is one of several tasks (= threads) running in parallel. The various tasks are responsible for handling asynchronous events, such as USB current detection, idle timers, etc. Isolating calls to the thread string function will reveal these tasks, as shown in Output xx-iboottasks:

Output xx-iboottasks: iBoot tasks (Addresses from iBoot-5540.0.15.0.2 d321, iOS 13β1)

```
morpheus@chimera (~)$ disarm -s 0x19c0a9d00-0x19c0ae7a0 -opcodes -base 0x19c030000 iBoot-5540.0.15.2 |
pipe>                 grep "func.*6b584"   # 0x6b584 is task_create()
0x19c0335e4  BL 0x19c06b584 ; func_06b584("main",0x19c0337d4)             # Main thread
0x19c033d50  BL 0x19c06b584 ; func_06b584("poweroff",0x19c033ef4)         # Button power off
0x19c033d7c  BL 0x19c06b584 ; func_06b584("command",0x19c09a298)          # command prompt (disabled)
0x19c033dc0  BL 0x19c06b584 ; func_06b584("idleoff",0x19c034098)          # Idle power off
0x19c06ddb4  BL 0x19c06b584 ; func_06b584("usb req",0x19c06de68)          # USB "vendor" requests
0x19c06ddfc  BL 0x19c06b584 ; func_06b584("usb-high-current",0x19c06deb0) # Detects iDevice plugging in
0x19c06de50  BL 0x19c06b584 ; func_06b584("usb-no-current",0x19c06dee0)   # Detects iDevice disconnection
0x19c07fe0c  BL 0x19c06b584 ; func_06b584("usb",0x19c07fe2c)              # Controller (Synopsis) OTG task
0x19c083138  BL 0x19c06b584 ; func_06b584("usb_serial",0x19c083478)       # USB serial console
```

The task structure has changed size over the years, but as recently as iBoot-5540 it is 432 bytes, with two magic values ('task' and 'tsk2') at their ends, further containing the task name (@408), its function pointer (@376) and argument (@384), and a pointer to a stack (@392) of a given length (@400), preinitialized to a magic of "stak". The tasks are linked through doubly linked list entries to the global task list (@8) and to the run queue (@24, with the queue @0x19c1a2b90)

Environment Variables

The RELEASE builds of iBoot allow a very limited subset of environment variables to be set or read from the NVRAM, using two whitelist tables - one for setenv and the other for getenv. The lists are shown in Output xx-nvramvars:

Output xx-nvramvars: The whitelisted environment variables, retrieved using disarm(j)

```
morpheus@Bifröst (....) % disarm  -base 0x19c030000  iBoot-5540.0.15.2 |
pipe>                     grep -A26 0x19c0ae910:
0x19c0ae910:    DCQ      0x19c0a9ecb ; "auto-boot"
0x19c0ae918:    DCQ      0x19c0a9ed5 ; "boot-args"                # Boot arguments (ignored)
0x19c0ae920:    DCQ      0x19c0a9edf ; "debug-uarts"              # Debug flags (3 = Serial UART)
0x19c0ae928:    DCQ      0x19c0a9d7b ; "filesize"                 # USB DFU filesize
0x19c0ae930:    DCQ      0x19c0a9eeb ; "pwr-path"
0x19c0ae938:    DCQ      0x0
0x19c0ae940:    DCQ      0x19c0a9ecb ; "auto-boot"                # Boolean: Boot boot-command, or stop
0x19c0ae948:    DCQ      0x19c0a9ef4 ; "backlight-level"          # Integer specifying display brightness
0x19c0ae950:    DCQ      0x19c0a9f04 ; "boot-command"             # 'fsboot', 'upgrade', etc.
0x19c0ae958:    DCQ      0x19c0a9f11 ; "com.apple.System.boot-nonce" # Seed for APTicket nonce
0x19c0ae960:    DCQ      0x19c0a9edf ; "debug-uarts"
0x19c0ae968:    DCQ      0x19c0a9f2d ; "device-material"
0x19c0ae970:    DCQ      0x19c0a9f3d ; "display-rotation"         # Rotate display (on watches)
0x19c0ae978:    DCQ      0x19c0a9f4e ; "display-vsh-comp"
0x19c0ae980:    DCQ      0x19c0a9f5f ; "idle-off"                 # Turn off after # of seconds idle
0x19c0ae988:    DCQ      0x19c0a9f68 ; "is-tethered"              # Connected to USB?
0x19c0ae990:    DCQ      0x19c0a9f74 ; "darkboot"
0x19c0ae998:    DCQ      0x19c0a9f7d ; "ota-breadcrumbs"          # OTA update progress
0x19c0ae9a0:    DCQ      0x19c0a9d64 ; "boot-breadcrumbs"         # Boot breadcrumbs
0x19c0ae9a8:    DCQ      0x19c0a9f8d ; "recovery-breadcrumbs"     # Recovery update progress
0x19c0ae9b0:    DCQ      0x19c0a9fa2 ; "com.apple.System.tz0-size"  # TrustZone (KPP)
0x19c0ae9b8:    DCQ      0x19c0a9fbc ; "com.apple.System.rtc-offset" # RTC Offset
0x19c0ae9c0:    DCQ      0x19c0a9eeb ; "pwr-path"
0x19c0ae9c8:    DCQ      0x19c0a9fd8 ; "upgrade-retry"
0x19c0ae9d0:    DCQ      0x19c0a9fe6 ; "preserve-debuggability"   # ?
0x19c0ae9d8:    DCQ      0x0
```

RELEASE iBoot recognizes several possible values for the boot-command NVRAM variable:

- **diags**: Boot a diagnostics image. The diags image is not present on release devices, but can be found on Switchboard devices, complete with an EFI runtime environment and a rich command line full of diagnostics functions and utilities.
- **fsboot**: The default for release builds, this finds and loads the iOS kernelcache, subject to APTicket verification.
- **upgrade**: Start an upgrade of the system. Used during OTA installation.
- **recover**: Fall to recovery mode.

As of iOS 12, iBoot also handles loading the various coprocessor firmwares. The iBoot image holds an array of structures which, for each coprocessor image, keeps the coprocessor name, device tree paths (e.g. arm-io/aop/iop-aop[-nub]), path to the image (an .img4 file, in /usr/standalone/firmware/FUD), and 32-bit tag.

Booting the kernelcache

iBoot's raison d'être is to locate and boot the iOS kernelcache. The default boot-command environment variable value is, thus, fsboot. Upon executing the fsboot command, iBoot checks the environment to locate the boot-path, boot-device and boot-partition (set to 0). These parameters provide the necessary parameters for mounting the root filesystem. iBoot thus needs a filesystem driver - either HFS+ (up to iOS 10.3) or APFS (thereafter). The default boot-path is /System/Library/Caches/com.apple.kernelcaches/kernelcache, though the default boot-device varies by device type (nand0, asp_nand0 or nvme_nand0). Because neither of these values is in the environment variable white list, the defaults are also the only options for RELEASE builds.

iBoot next checks the dt-path and boot-ramdisk variables, which determine the Device Tree to pass to the kernelcache, as well as an optional RAM disk image (for iBEC mode). The struct boot_args (see Table xx-bootargs, later in this chapter), is constructed, and passed to the kernelcache (uncompressed in memory) as an argument.

Threat modeling iBoot

The boot process is the linchpin of iOS security. For all of Apple's formidable hardware security mechanisms, they must be explicitly enabled by software. But if the boot sequence is compromised, KPP can be left out, or the kernelcache can be patched a priori of loading, thus removing the instructions vital for starting KTRR and/or APRR, or neutering `AMFI` and the `Sandbox.kext`. From the jailbreaking perspective, a boot sequence vulnerability could enable a tethered jailbreak or dual booting, and (if in the SecureROM) potentially be unpatchable for the afflicted iDevice models.

Security researchers and jailbreakers alike thus invest a great deal of effort in seeking potential vulnerabilities. At a high level, three attack surfaces can be identified:

- **USB**: All boot components have some form of USB support, with functionality increasing from the bare minimum of DFU (in the SecureROM) to full iTunes protocol support (in iBoot). The A4 "limera1n" exploit was, for almost a decade, the only effective example of a SecureROM vulnerability, and was reliably exploited to provide tethered, unpatchable jailbreaks up until the iPhone 4. This changed just as this book was going to print, when @axi0mX released his "checkm8" tool[cm8] to provide a working SecureROM level exploit for all devices up to and including the A11 (iPhone 8/X).
- **NAND**: Each component of the boot chain has to locate the next stage, which involves reading either the partitions (NVMe namespace) or the filesystem (iBoot). Targeted malformations could potentially cause memory corruption at one of the boot stages, yielding code execution. Another deliberate malformation could be in the digital signature of each component (specific X.509/DER fields). A mitigating factor is that such attacks require initial low level access to the NAND in order to mount (commonly, through a pre-existing jailbreak, or NAND MiTM). An iBoot vulnerability in HFS+ handling was demonstrated and eventually fully documented by @Xerub for 32-bit iBoot, though no vulnerabilities are publicly known for iBoot 64-bit.
- **Digital Signatures**: aside from signature malformation as a vector for memory corruption, there is also the potential for digital signature forgery, which would allow loading a compromised stage. This is highly improbable (an adversary capable of reliably breaking RSA or performing a second pre-image for SHA-256 likely has even more lucrative targets than iPhones), but cannot be entirely discounted. Another risk is that Apple's root certificate private keys could be stolen or otherwise selectively leak.

Considering that companies such as Cellebrite and GrayShift make their business off of breaking into iPhones, one can only deduce vulnerabilities do exist. Their extremely high value in the exploit market, coupled with Apple's stingy bounty, makes it likely they will remain hidden for as long as possible, maybe even indefinitely. USB based vulnerabilities would be of the highest value, since they allow for "evil chambermaid" attacks, wherein an iDevice left unattended could be rebooted and compromised by an attacker with momentary access. Said attacker could compromise the kernelcache, and wait for the user to unlock their device - and thus capture their passcode. Though powerful, this attack would not survive a hard reset of the iDevice, but when coupled with NAND vulnerabilities, persistence could be achieved.

Many thanks to iH8Sn0w and @AXi0mX for reviewing this section!