The One Weird Trick SecureROM Hates

#POC2019, Seoul, South Korea

~qwertyoruiop

whoami

- Luca Todesco aka qwertyoruiop
    - Often idling in irc.cracksby.kim #chat
    - @qwertyoruiopz on Twitter
- Independent security researcher with an iOS focus
    - Grew up around the iOS hacking scene

~~whoami~~

- ~~Luca Todesco aka qwertyoruiop~~
    - ~~Often idling in irc.cracksby.kim #chat~~
    - ~~@qwertyoruiopz on Twitter~~
- ~~Independent security researcher with an iOS focus~~
    - ~~Grew up around the iOS hacking scene~~

- This talk and the work I will present would have not been possible without the contributions of several people
  - axi0mX, littlelailo and Siguza for the underlying bug and exploitation strategies
  - the checkra1n team for the work put into building the infrastructure and the software required to turn this into a jailbreak

whoareus

- checkra1n team
  - argp, axi0mx, danyl931, jaywalker, kirb, littlelailo, nitoTV, nullpixel, pimskeks, qwertyoruiop, sbingner, siguza, haifisch, jndok, jonseals, xerub, lilstevie, psychotea, sferrini, Cellebrite (ih8sn0w, cjori, ronyrus et al.), et al.

whoareus

- checkra1n team
  - A broad group of iOS hackers from all around the iOS hacking/jailbreaking scene, both public and private, focusing on both hardware and software aspects
  - Possibly the best team of people I have had the pleasure of working with in my entire “career”
    - I am very grateful for the opportunity to work in such a manner with people so skilled and passionate about what they do.

whatis SecureROM

- SecureROM is the very first code to run on the Application Processor upon cold boot
  - Stripped-down and simplified version of iBoot
  - Patterned in silicon as mask ROM, thus immutable
  - Most trusted code to run on the Application Processor
  - Main goal is to load a first-stage bootloader image from non-volatile memory and boot it
  - But it also provides an emergency recovery mechanism called DFU, which allows you to upload an image over USB
    - Can enter via a special key combo held during reset

whatis Secure Boot

- iOS pioneered the concept of Secure Boot, and SecureROM is a key part of the trusted boot sequence
  - On all devices SecureROM implements secure boot mechanisms when uploading an image over DFU
    - These also apply for images in non-volatile storage on all devices except S5L8900 (original iPhone)
  - Secure boot mechanisms are jailbreaking enemy number one
    - Jailbreaks rely on bypassing this mechanism, either by compromising a node in the chain-of-trust, or simply compromising an already fully booted system

DFU Protocol

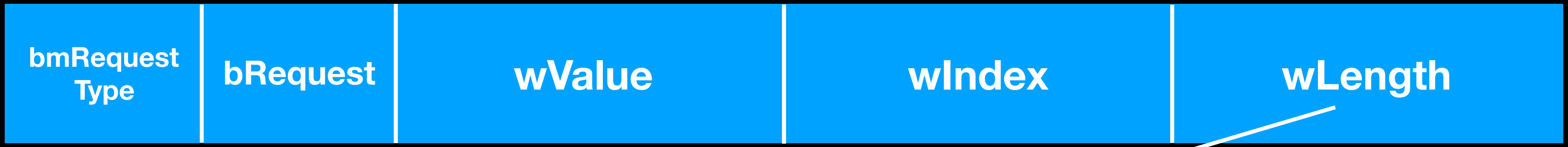
- DFU is a fairly simple protocol implemented over USB control transfers
    - Goal is to just allow the device to get data from host in a simplified way
- Data chunks are uploaded via request 0x21, 1
- Once the transfer is done, three status requests followed by a USB reset will exit DFU and attempt to boot the uploaded image
    - But DFU can be exited in other ways too, for instance via request 0x21, 4 (DFU abort)

USB Control Transfer

- Used to issue commands, send data from host to device or request data from device to host
  - Successful USB enumeration requires a few of these to be handled by the device
  - It starts with a setup packet, an 8 byte structure containing information on the transfer about to happen

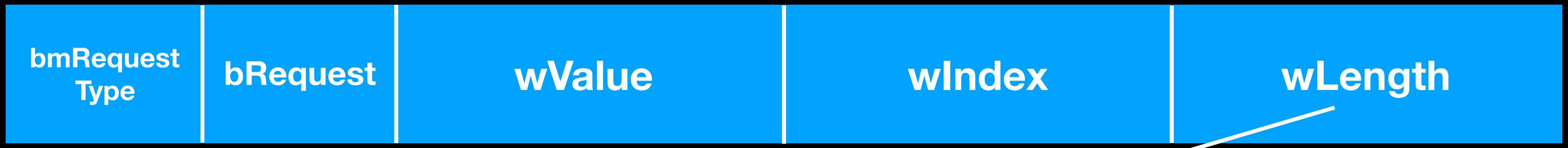
bmRequest Type	bRequest	wValue	wIndex	wLength

USB Control Transfer



If wLength is non-zero, the setup packet is followed by a “data phase”, which e.g. in DFU mode is how the data is transferred from host to device, although this can also be done device-to-host depending on the bmRequestType

USB Control Transfer



If wLength is non-zero, the setup packet is followed by a “data phase”, which e.g. in DFU mode is how the data is transferred from host to device, although this can also be done device-to-host depending on the bmRequestType

The “data phase” will chunk the data in packets of sizes ranging from 8 bytes to 64 bytes (depending on USB speed) and send it sequentially

USB Control Transfer

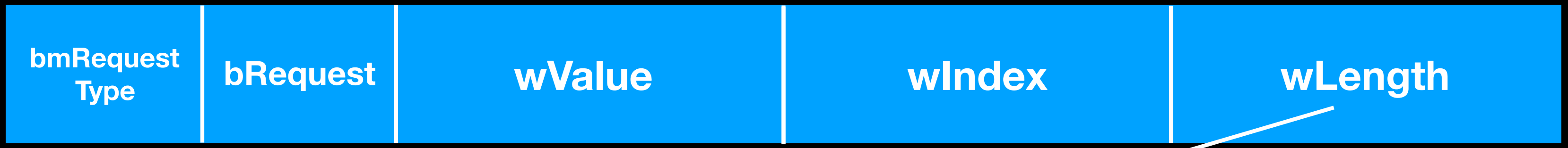
bmRequest Type	bRequest	wValue	wIndex	wLength

If wLength is non-zero, the setup packet is followed by a “data phase”, which e.g. in DFU mode is how the data is transferred from host to device, although this can also be done device-to-host depending on the bmRequestType

The “data phase” will chunk the data in packets of sizes ranging from 8 bytes to 64 bytes (depending on USB speed) and send it sequentially

Once the transfer is done, a “status phase” follows, which marks the end of the control transfer by sending a zero-length packet

USB Control Transfer



If wLength is non-zero, the setup packet is followed by a “data phase”, which e.g. in DFU mode is how the data is transferred from host to device, although this can also be done device-to-host depending on the bmRequestType

The “data phase” will chunk the data in packets of sizes ranging from 8 bytes to 64 bytes (depending on USB speed) and send it sequentially

In iBoot’s USB stack, a temporary buffer is *allocated during USB initialisation*, and data phase packets get memcpy’d into it once received by the device

Once the transfer is done, a “status phase” follows, which marks the end of the control transfer by sending a zero-length packet

USB and DFU

- The USB stack is turned on when entering DFU
    - Triggering the allocation of the buffer used for holding data temporarily during a control transfer’s data phase
- Once a control transfer is issued, a pointer to this buffer is copied into a global variable which the USB stack will use as destination for the data coming in via data phase
- … but upon DFU exit, the USB stack is turned off again
    - Triggering a free of the said buffer

The bug

- The USB stack is turned on when entering DFU
  - Triggering the allocation of the buffer used for holding data temporarily during a control transfer’s data phase
- Once a control transfer is issued, a **pointer to this buffer is copied into a global variable** which the USB stack will use as destination for the data coming in via data phase
- … but upon DFU exit, the USB stack is turned off again
  - **Triggering a free of the said buffer**

The bug

- The USB stack is turned on when entering DFU
    - Triggering the allocation of the buffer used for holding data temporarily during a control transfer's data phase
- Once a control transfer is issued, a **pointer to this buffer is copied into a global variable** which the USB stack will use as destination for the data coming in via data phase
- … but upon DFU exit, the USB stack is turned off again
    - **Triggering a free of the said buffer** $\rightarrow$ **The global variable is never NULL'd out**

The bug

- The USB stack is turned on when entering DFU
  - Triggering the allocation of the buffer used for holding data temporarily during a control transfer's data phase
- Once a control transfer is issued, **a pointer to this buffer is copied into a global variable** which the USB stack will use as destination for the data coming in via data phase

Use-After-Free

- … but upon DFU exit, the USB stack is turned off again
  - **Triggering a free of the said buffer**

The global variable is never NULL'd out

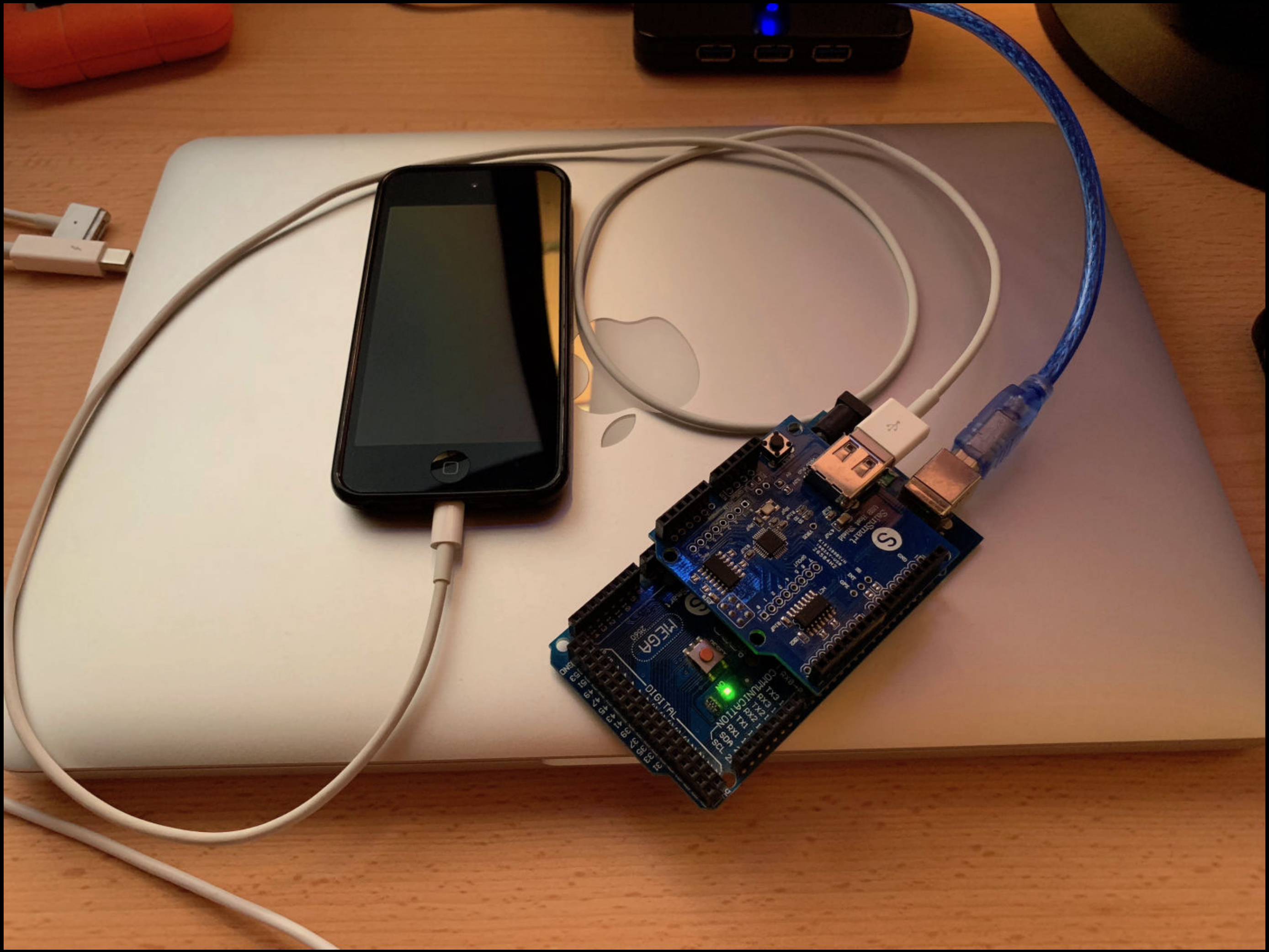
Trigger

- Start an USB control request transfer with a data phase
  - Interrupt transfer mid-way
- Issue a DFU abort (0x21, 4), causing the temporary buffer to be free’d
  - DFU is re-entered, reallocating the buffer
- Try to complete the previously-interrupted transfer by sending data phase packets once DFU is re-entered
  - The data will be memcpy’d on top of a free’d pointer

Practical Trigger

- Littlelailo and Siguza’s approach was to take an Arduino with a USB host shield in order to get full control over the host-side USB stack
  - Full control on the host-side USB stack is needed in order to be able to do partial control transfers, as most USB stacks won’t really let you do this without abusing tricks
- Axi0mX’s approach was to abuse the USB stack on macOS in order to have a transfer be aborted mid-way
  - Not deterministic, but multiple attempts can be made, and many USB stacks (eg. IOUSB, usbfs and WinUSB) provide feedback on how much data has been transferred in case of failed transfers, thus allowing a practically-deterministic implementation without custom hardware or drivers

Practical Trigger



From Siguza and littlelailo

SecureROM Exploitation

- If you attempted to follow these steps, nothing would happen on most devices
  - An exception is A8 and A9, due to a bug in the DFU abort functionality
    - Siguza and littlelailo analysed these two devices in particular, and noted that a task structure is leaked in the heap every time a DFU abort is done

SecureROM Exploitation (A8, A9)

- We are corrupting a task structure
  - Upon task yield, registers are saved to this structure, and upon scheduling they are restored off of it
    - UaF write happens on the currently-scheduled task, thus can't just overwrite registers, as those won't be used and will be overwritten
    - Tasks however have a linked list off of which the next task to schedule is chosen
      - Create a fake task structure and on next yield we get full control over the register state
        - No ROP/JOP needed: we achieve **direct code execution**

SecureROM Exploitation

- 32-bit ROMs (except S1 SoC), A7, A10, A10X, A11 and A11X can't use this strategy
  - There is no task structure leak to abuse
  - No crash is triggered whatsoever as the ROM is deterministic enough that the buffer is reallocated in the same place every time upon USB stack initialisation
    - Need a way to break this determinism in order to get a proper UaF scenario

SecureROM Exploitation

- Need a way to break this determinism in order to get an usable UaF scenario
  - The heap allocator in SecureROM returns the smallest possible hole available for a given allocation size
    - Heap feng shui
    - Axi0mX uses this in order to create an exploitable UaF condition
    - First of all, for this, we need controlled allocation primitives

SecureROM Exploitation

- In-flight USB transfers will have an associated structure allocated on the heap
  - Multiple USB transfers may be in-flight at the same time
    - For example, for requests that have data going device-to-host if the IN endpoint is stalled, the device is not going to be able to send data until the stall is cleared
      - Every setup packet for device-to-host requests sent in such a condition will end up staying around for a while

SecureROM Exploitation

- We have the ability to repeatedly malloc and delay the free temporarily, until stall conditions are cleared or the USB stack is shut down
    - The issue is that we need some allocations to persist across a DFU abort in order for our heap shaping to influence the next allocation of our buffer

SecureROM Exploitation

- A state machine bug in the USB stack is abused in order to have allocations that persist across USB stack destruction and creation
    - The standard device-to-host USB request destructor will send a zero-length packet once a request is fulfilled or the USB stack shuts down
        - The issue is that when the USB stack is shutting down, the zero-length packet is never sent and ends up being leaked
            - This leak can be triggered conditionally as only packets that match specific conditions will trigger this leak - for instance, requests that are not a multiple of 64 bytes won’t trigger the leak

SecureROM Exploitation

- We can use this heap leak in order to carefully craft the state of the heap and end up with an exactly sized hole, which will be the preferred place for malloc to put the buffer on the next allocation
    - Each USB request structure contains a function pointer to a callback, so we can now reallocate a few in place and perform the use-after-free write in order to get a controlled indirect branch upon stall clear or DFU exit
- This strategy can be successfully used on A9X, A10, A10X and A11

SecureROM Exploitation

- On older devices such as A7 this technique is not workable since the leak is always triggered for all in-flight USB requests
    - Can however have the heap almost run out, minus exactly the size of the buffer, thus creating a hole that is going to be preferred by malloc

Checkmate.

However, all we have now is code execution, we need to develop a bootkit of sorts to continue.

Bootkit ELI5

- We have code execution in SecureROM's DFU mode
- Our goal is then to boot the device normally and compromise each stage in the boot sequence
  - Eventually the bootloader will have the kernel loaded in memory and attempt to branch into it's entry point
    - Our goal is to patch the kernel at that point in time, as it's still fully read/write
- Turns out to be a bit tricky, especially when wanting to support multiple devices and versions without having to hardcode offsets
  - Resilient patchfinding strategies are required in order to keep such a project reasonably maintainable

Bootkit Development

- iBoot makes use of a “boot trampoline” which gets the CPU state back to an almost-reset one
  - Previous image is wiped (in case of ROM, access is disabled)
  - MMU is disabled
  - All registers set to zero
    - Including registers such as X18, which due to ABI constraints is never really used normally
      - Very easy to find the trampoline by looking for a mov x18, #0 instruction
        - Can write shellcode to dynamically locate this

Bootkit Development

- There is a chicken-and-egg issue with the trampoline
  - It will wipe the current-stage boot loader
    - So it has to be relocated off of the boot loader into some memory which is not wiped
    - A reserved region is used for this, however on some versions the trampoline is relocated just before trying to use it
      - This is problematic on SecureROM as it’s read only and we may get code execution before this happens

Bootkit Development

- We can relocate the SecureROM into SRAM and alter TTBR0_ELn in order to remap the ROM range
    - On some devices, a 512KB area can be used for this purpose
        - Whole ROM can be relocated there
    - On other devices, there is not much free SRAM we can abuse
        - Could use exotic strategies such as resizing the heap, reserving more L2 cache as addressable SRAM
        - But all we really need is to remap one single page in order to alter the trampoline

Bootkit Development

- We also need to find some area where to stash our shellcode
  - However we're still restricted to the available free SRAM
  - And we also need to do a normal boot, which itself makes use of some SRAM
  - Maybe we can use DRAM?
    - Not initialised while SecureROM is running; the first-level boot loader is in charge of DRAM initialisation

Bootkit Development

- We also need to find some area where to stash our shellcode
  - Maybe we can use DRAM?
    - Not initialised while SecureROM is running; the first-level boot loader is in charge of DRAM initialisation
      - The first level boot loader has a feature called SoftDFU that simulates DFU mode but has DRAM available at that point in time, allowing a larger image to be uploaded
        - We can abuse this to load a practically arbitrary amount of data in DRAM without having to bother initialising it ourselves

Bootkit Development

- Large portions of DRAM can be cleared by iBoot
  - Dirty workaround: hook bzero and detect wether the range overlaps with our shell code
    - If so, bzero is considered to be a NOP
    - Not the nicest way to pull it off, but it does work pretty well

Bootkit Development

- Once the trampoline hook and bzero hook are in place we are able to reach the next trampoline invocation
    - Re-execute these hooks in the next-stage boot loader if necessary
        - Devices before A10 use two stages, A10 and later only really use one, so this is not strictly required there

Bootkit Development

- Eventually the boot loader will have the kernel prepared and the trampoline will be used in order to invoke the entry point
  - On devices where EL3 is present, the Secure Monitor entry point is used instead, and the kernel arguments and entry point are passed to it in order for them to be loaded in EL1 by the SM
  - The hook we perform on the trampoline means we override the branch target into our own shellcode, and pass the original value in a register we prevent from being cleared

Bootkit Development

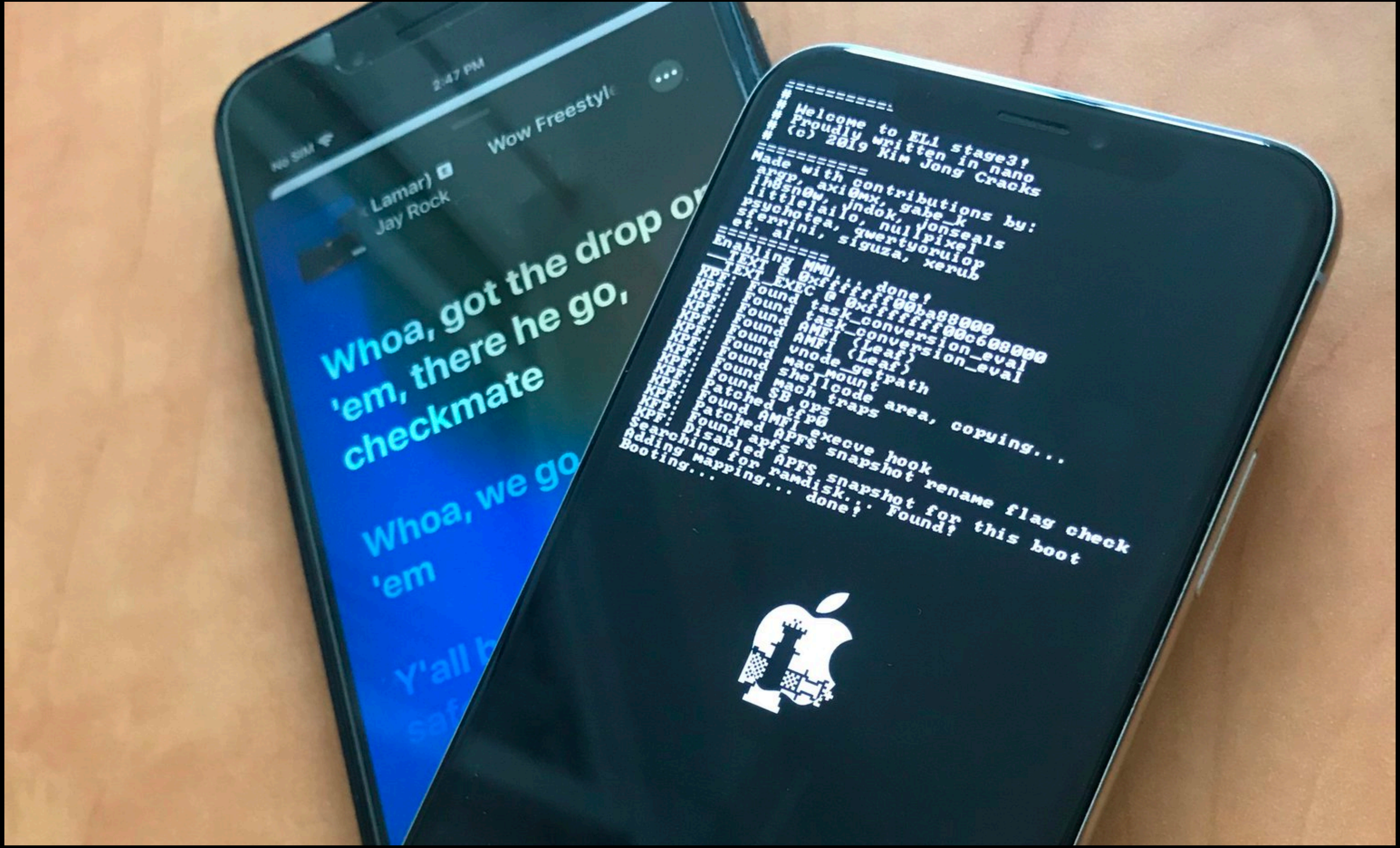
- We can now patch the kernel
    - However due to having to support arbitrary device/version pairs patchfinding is the only sane strategy to use here
        - Due to the way we patched the trampoline we gain code execution at a point in time with no MMU or caching enabled
        - This becomes really slow really fast due to the sheer amount of memory accesses we need to perform
            - We turn MMU back on with a set of TTEs that map a view of DRAM as cacheable for the patchfinder to work off of

Bootkit Development

- We apply a pretty old-school kernel patchset
  - KPP and KTRR are non-issues since these patches are performed before they get initialised at all
    - iOS 8 all over again
  - vm_map_protect, vm_fault_enter, AMFI trustcache validation, sandbox MACF hooks, mount, TFP0, apfs…

Bootkit Development

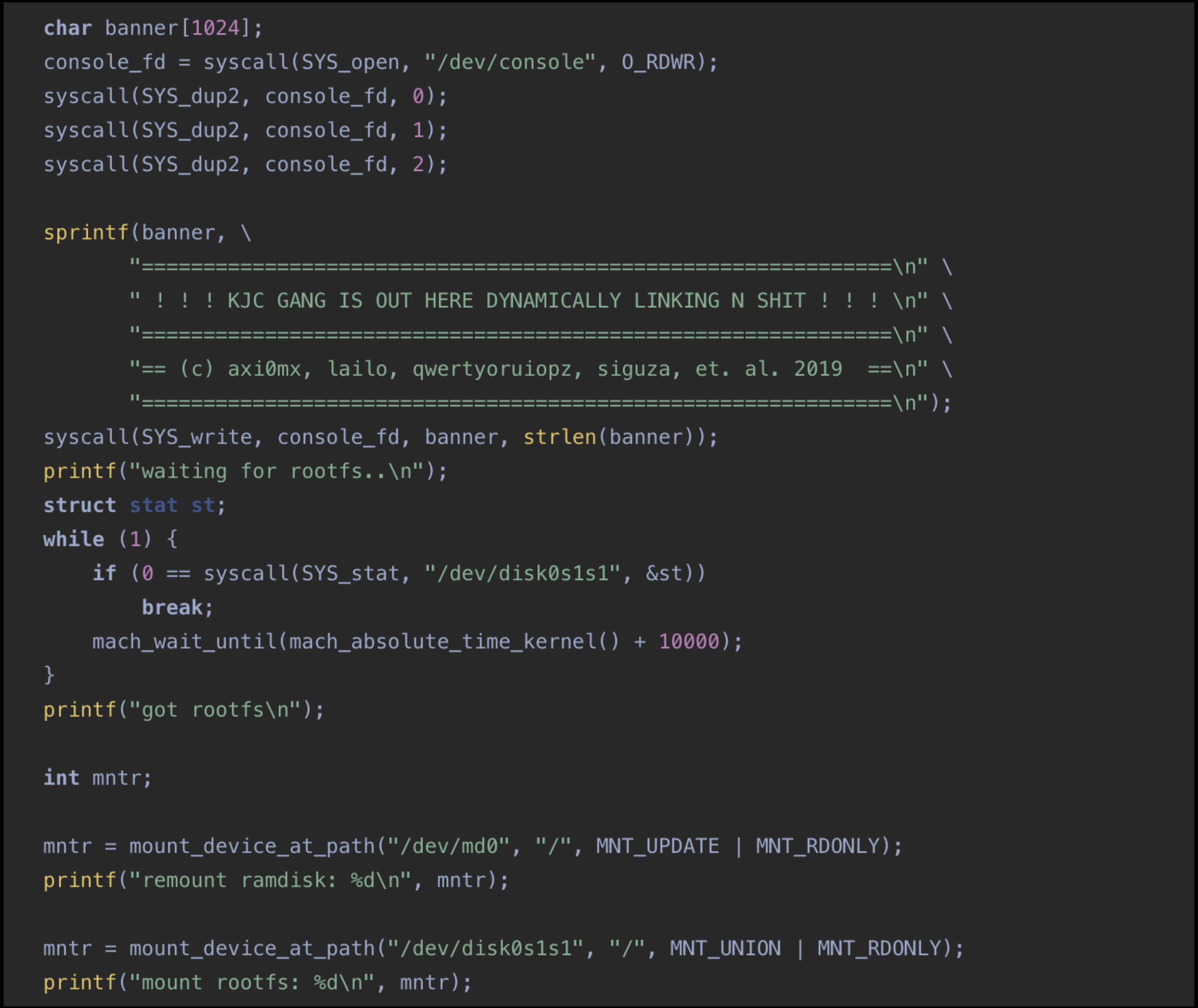
- Kernel patching alone is not enough
  - Once the kernel boots we need to retain code execution in usermode in order to prepare the jailbroken state
  - We can embed a tiny ramdisk in our shellcode in order to hijack EL0 code execution and patch the device tree and kernel boot arguments structure in order to map it in and have it used as root device



```
===========
# Welcome to EL1 stage3!
# Proudly written in nano
# (c) 2019 Kim Jong Cracks
===========
Made with contributions by:
argp, axi0mx, gabe_k
ih8sn0w, jndok, jonseals
littlelailo, nullpixel
psychotea, qwertyoruiop
sferrini, siguza, xerub
et. al.
===========
Enabling MMU... done!
__TEXT @ 0xfffffff00ba88000
__TEXT_EXEC @ 0xfffffff00c608000
KPF: Found task_conversion_eval
KPF: Found task_conversion_eval
KPF: Found AMFI (Leaf)
KPF: Found AMFI (Leaf)
KPF: Found vnode_getpath
KPF: Found mac_mount
KPF: Found shellcode area, copying...
KPF: Found mach traps
KPF: Patched SB ops
KPF: Patched tfp0
KPF: Found AMFI execve hook
KPF: Patched APFS snapshot
KFP: Found apfs snapshot rename flag check
KPF: Disabled APFS snapshot for this boot
Searching for ramdisk... Found!
Adding mapping... done!
Booting...
```

Jailbreak Development

- We started from SecureROM code execution
- We now got all the way to EL0 user-mode code execution
  - However our ramdisk has to be small due to performance: DFU transfers are not that fast, and having to wait several seconds on a data upload is not great
  - Copyright concerns also mean we should avoid shipping any Apple proprietary code with such a ramdisk
    - Problematic since we need a dynamic linker and at least a few user-mode libraries to link our code against



```c
char banner[1024];
console_fd = syscall(SYS_open, "/dev/console", O_RDWR);
syscall(SYS_dup2, console_fd, 0);
syscall(SYS_dup2, console_fd, 1);
syscall(SYS_dup2, console_fd, 2);

sprintf(banner, \
       "=============================================================\n" \
       " ! ! ! KJC GANG IS OUT HERE DYNAMICALLY LINKING N SHIT ! ! ! \n" \
       "=============================================================\n" \
       "== (c) axi0mx, lailo, qwertyoruiopz, siguza, et. al. 2019  ==\n" \
       "=============================================================\n");
syscall(SYS_write, console_fd, banner, strlen(banner));
printf("waiting for rootfs..\n");
struct stat st;
while (1) {
    if (0 == syscall(SYS_stat, "/dev/disk0s1s1", &st))
        break;
    mach_wait_until(mach_absolute_time_kernel() + 10000);
}
printf("got rootfs\n");

int mntr;

mntr = mount_device_at_path("/dev/md0", "/", MNT_UPDATE | MNT_RDONLY);
printf("remount ramdisk: %d\n", mntr);

mntr = mount_device_at_path("/dev/disk0s1s1", "/", MNT_UNION | MNT_RDONLY);
printf("mount rootfs: %d\n", mntr);
```

…. so we wrote our own Totally Legitimate dynamic linker

Jailbreak Development

- We use syscall(3) in order to mount the real root filesystem on top of /, using union mounting as to not destroy the vnode backing the code we’re running
  - Immediately shadows /usr/lib/dyld, giving us a real dynamic linker to work with, and provides a full dyld shared cache to link against
    - Can now run real code as PID 1, before launchd is ever executed

Jailbreak Development

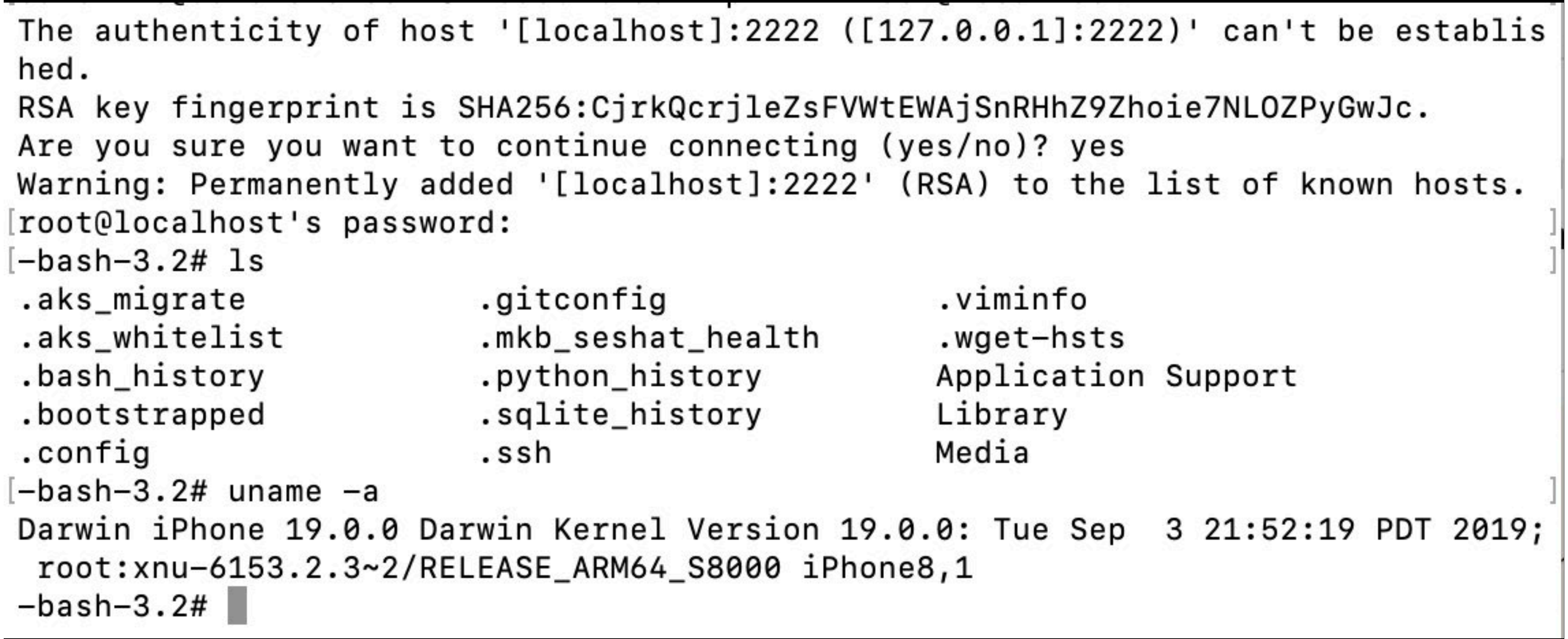
- The root filesystem at this point is still read-only, and we'd like to keep it that way until the user explicitly decides to change this
  - But we also need to drop extra files in order to get a shell and allow users to install a package manager if they so wish
    - So we need the /private/var partition to be available
    - Data protection needs to be initialised for this, and launchd is responsible for this

Jailbreak Development

- We need to hook execution of something early in the boot process, but late enough as to have data protection enabled
  - So we keep on union mounting, this time another tiny .dmg on top of /usr/libexec/
    - Can override any system daemon
      - We picked sysstatuscheck as it runs at the right point in time across multiple iOS versions
        - execve our way back into a vnode off the original ramdisk and forcefully umount the /usr/libexec dmg

Jailbreak Development

- We can now let launchd continue booting by simply execve'ing the original daemon
  - And we can fork(2) off before doing this as to run our code as the system boots up
    - Wait until usbmux turns on and wait for host to upload a .dmg containing all the required utilities for a basic shell to run, a ssh daemon and a fancy .app to let users install a proper jailbreak bootstrap on their root filesystem
      - We mount this on /binpack and kickstart the ssh daemon



```
The authenticity of host '[localhost]:2222 ([127.0.0.1]:2222)' can't be establis
hed.
RSA key fingerprint is SHA256:CjrkQcrjleZsFVWtEWAjSnRHhZ9Zhoie7NLOZPyGwJc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:2222' (RSA) to the list of known hosts.
root@localhost's password:
-bash-3.2# ls
.aks_migrate            .gitconfig              .viminfo
.aks_whitelist          .mkb_seshat_health      .wget-hsts
.bash_history           .python_history         Application Support
.bootstrapped           .sqlite_history         Library
.config                 .ssh                    Media
-bash-3.2# uname -a
Darwin iPhone 19.0.0 Darwin Kernel Version 19.0.0: Tue Sep  3 21:52:19 PDT 2019;
 root:xnu-6153.2.3~2/RELEASE_ARM64_S8000 iPhone8,1
-bash-3.2# 
```

and once SpringBoard runs…

iPod
09:3
Files
Find My
Tips
Voice
checkra1n
19:30
FaceTime
Podcasts
Calculator
Files
Shortcuts
Extras
checkra1n
Measu

Future Plans

- Ideally our aim for this project is to move from being a pure jailbreak to providing something like an iOS version of Clover
  - Custom on-boot kernel extension loading
    - Jailbreak patch set can be moved here to simplify maintenance and versioning
  - Dual booting
    - Very useful for research purposes
  - Linux on iPhone
    - :)