DE MYSTERIIS DOM JOBSIVS: MAC EFI ROOTKITS

SNARE
@ RUXCON
OCTOBER 2012

assurance

THIS GUY'S TOO TRUSTWORTHY

WHAT'S HIS ANGLE?

- Loukas/snare
  - From Melbourne, Australia
  - Principal Consultant at Assurance
  - We test pens and stuff
  - Keeping infosec metal
- Most \m/etal infosec company in Australia.

AGENDA

- Things I will talk about
  I. Introduction - goals, concepts & prior work
  II. EFI fundamentals
  III. Doing bad things with EFI
  IV. Persistence
  V. Evil maid attacks
  VI. Defence against the dark arts

I. INTRODUCTION

INTRODUCTION
I WANT A COOL BOOT SCREEN ON MY MAC

- Why are we here?
  - I wanted to mess with pre-boot graphics (seriously)
  - Minimal knowledge of firmware / bootloader
  - Did some research...
  - Wait a minute, backdooring firmware would be badass
  - But, of course, it's been done before...

INTRODUCTION
PRIOR ART

- Other work in this area
  - Old MBR viruses
  - ...
  - John Heasman @ Black Hat '07 (badass talk on EFI)
  - Core Security @ CanSecWest '09 (BIOS infection)
  - Invisible Things @ Black Hat '09 (Intel UEFI BIOS)
  - endrazine @ Black Hat 2012 (BIOS/Coreboot)
  - and more...

INTRODUCTION
GOALS

- Backdoor a machine
  - Preferably without evidence on-disk
  - Persist forever?
    - Across reboots, reinstalls, disk replacement, heat death of the universe
  - Patch the kernel at boot time
  - Work regardless of whole-disk encryption
- Sound hard?
  - Nah
  - (OK yeah, kinda)

II. EFI FUNDAMENTALS

WHAT'S AN EFI?
AND WHY DO I CARE?

- BIOS replacement
  - Initially developed at Intel
  - Designed to overcome limitations of PC BIOS
  - "Intel Boot Initiative"
  - Used in all Intel Macs - now I care
  - Used on lots of PC mobos as UEFI
    - With Compatibility Support Module (CSM) for BIOS emulation
- UEFI?
  - Handed over to Unified EFI Consortium @ v1.10
  - Apple's version reports as v1.10

EFI ARCHITECTURE
PUTTING THE "SUCK" IN "FUNDAMENTALS"!

- Modular
  - Comprises core components, apps, drivers, bootloaders
  - Core components reside on firmware
    - Along with some drivers
  - Applications & 3rd party drivers
    - Reside on disk
    - Or on firmware data flash
    - Or on option ROMs on PCI devices

EFI ARCHITECTURE
TERMINOLOGY

- Tables - pointers to functions & EFI data
  - System table
    - Pointers to core functions & other tables
  - Boot services table
    - Functions available during EFI environment - useful!
    - Memory allocation
    - Registering for timers and callbacks
    - Installing/managing protocols
    - Loading other executable images

EFI ARCHITECTURE
TERMINOLOGY

- Tables - pointers to functions & EFI data
  - Runtime services table
    - Functions available during pre-boot & while OS is running
    - Time services
    - Virtual memory - converting addresses from physical
    - Resetting system
    - Capsule management
    - Variables (we will use this)
      - NVRAM on the Mac - boot device is stored here
  - Configuration table
    - Pointers to data structures for access from OS
    - Custom runtime services

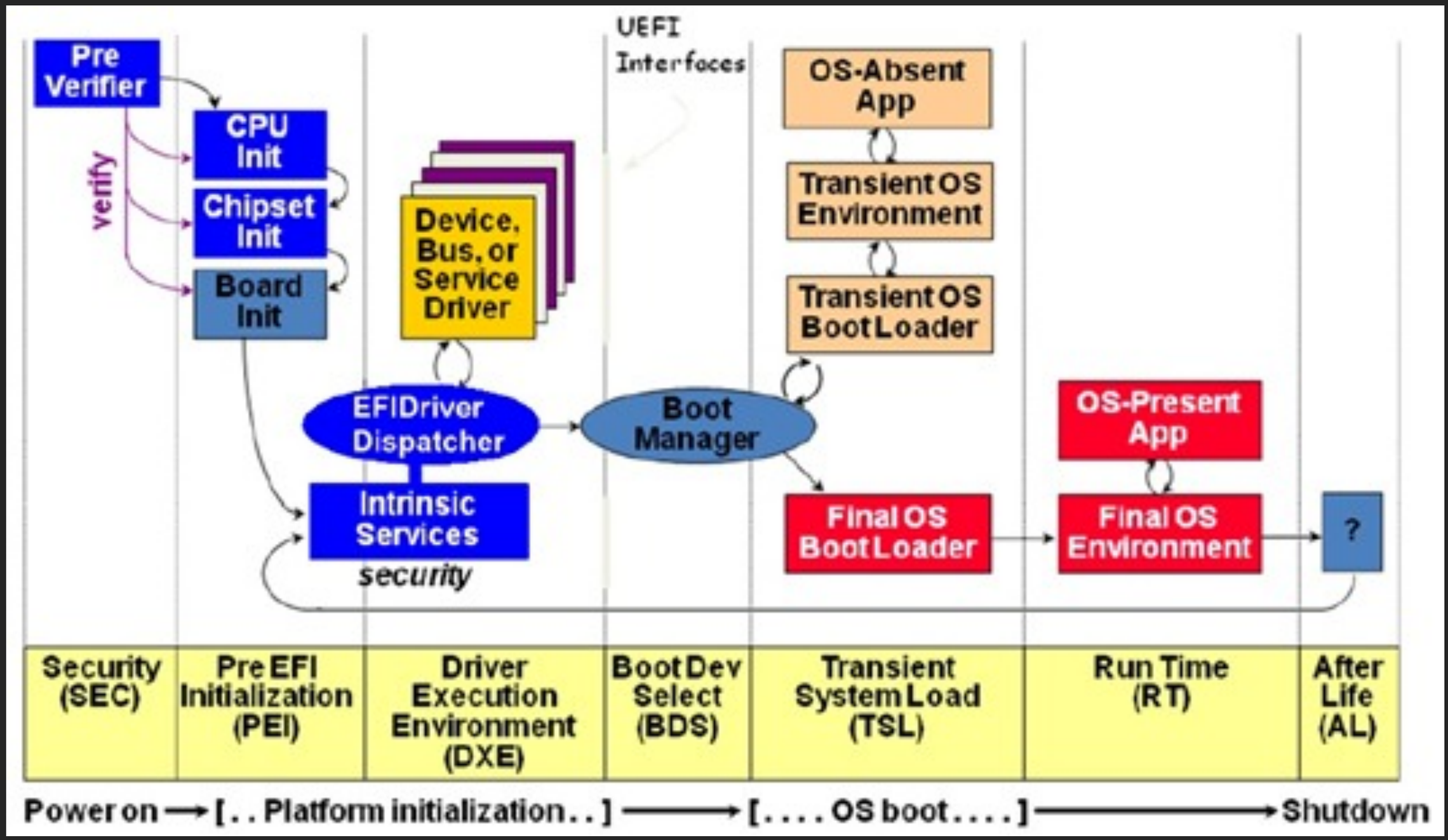
EFI ARCHITECTURE
DEVELOPING FOR EFI

- EDK2 - EFI Development Kit
  - Includes "TianoCore" - Intel's reference implementation
    - Most of what Apple uses
    - And probably most other IBVs
  - Written in C
    - Builds PE executables
    - >2mil lines of code in *.c/*.h
      - Compared to ~1.1mil in XNU
      - `find . \( -name "*.c" -o -name "*.h" \)|xargs cat|wc -l`
        - (not very scientific, whatever)
- Spec is 2156 pages long at v2.3.1

EFI ARCHITECTURE
STATS

- Some telling examples of defined protocols
  - Disk/filesystem access, console input/output
  - Graphics Output Protocol (graphical console)
  - Human Interface Infrastructure (UI forms!)
  - IPv4, IPv6, TCP, UDP, IPSEC, ARP, DHCP, FTP, TFTP
  - User management, SHA crypto, key management...
  - Heaps more
- Starting to sound like an entire OS

EFI ARCHITECTURE
BOOT PROCESS



Token shitty, low res diagram stolen from documentation

REVERSING EFI
KINDA SUCKS

- No dynamic linking/loading for EFI modules
- Entry point is passed pointers to tables
- All API functionality is provided through tables
- Protocols are accessed via Boot Services functions

- I wrote some helper scripts for IDA Pro
  - https://github.com/snarez/ida-efiutils

REVERSING EFI
KINDA SUCKS

```
loc_337F:
mov     [rbp+var_34], 0
mov     [rbp+var_38], 0
mov     [rbp+var_C8], 6
mov     rax, cs:qword_79A28
lea     rcx, [rbp+var_38]
mov     [rsp+1B0h+var_190], rcx
lea     rcx, aR             ; "R"
lea     rdx, dword_76014
lea     r9, [rbp+var_C8]
xor     r8d, r8d
call    qword ptr [rax+48h]
test    rax, rax
jns     short loc_33F5
```

```
mov     rax, cs:qword_79A28
mov     ecx, 0FFFFFF01h
mov     [rsp+1B0h+var_190], rcx
lea     rcx, aR             ; "R"
lea     rdx, dword_76014
mov     r8d, 6
mov     r9d, 6
call    qword ptr [rax+58h]
```

```
loc_337F:
mov     [rbp+var_34], 0
mov     [rbp+var_38], 0
mov     [rbp+var_C8], 6
mov     rax, cs:gRuntimeServices
lea     rcx, [rbp+var_38]
mov     [rsp+1B0h+var_190], rcx
lea     rcx, aR             ; "R"
lea     rdx, dword_76014
lea     r9, [rbp+var_C8]
xor     r8d, r8d
call    [rax+EFI_RUNTIME_SERVICES.GetVariable]
test    rax, rax
jns     short loc_33F5
```

```
mov     rax, cs:gRuntimeServices
mov     ecx, 0FFFFFF01h
mov     [rsp+1B0h+var_190], rcx
lea     rcx, aR             ; "R"
lea     rdx, dword_76014
mov     r8d, 6
mov     r9d, 6
call    [rax+EFI_RUNTIME_SERVICES.SetVariable]
```

REVERSING EFI
KINDA SUCKS

```
qword_76688     dq 4A3823DC9042A9DEh, 6A5180D0DE7AFB96h
                                        ; DATA XREF: sub_
                                        ; sub_8354+3E↑o .
qword_76698     dq 11D295625B1B31A1h, 3B7269C9A0003F8Eh
                                        ; DATA XREF: sub_
                                        ; sub_1EADB+52↑o
qword_766A8     dq 11D2954C56EC3091h, 3B7269C9A0003F8Eh
                                        ; DATA XREF: sub_
qword_766B8     dq 11D26459964E5B22h, 3B7269C9A000398Eh
                                        ; DATA XREF: sub_
                                        ; sub_171C4+1E3↑o
qword_766C8     dq 11D50B7531878C87h, 4DC13F2790004F9Ah
                                        ; DATA XREF: sub_
                                        ; sub_1EBC6+49↑o
qword_766D8     dq 41CBF4FA982C298Bh, 39B88F68AA7738B8h
                                        ; DATA XREF: sub_
                                        ; sub_1D1E3+71↑o
qword_766E8     dq 4C12012EF42F7782h, 21F70443F9495699h
                                        ; DATA XREF: sub_
qword_766F8     dq 444F506C7AA35A69h, 0C8716FF54B69AFA7h
                                        ; DATA XREF: sub_
qword_76708     dq 49016E735B213447h, 72A1B7F364B8F1A4h
                                        ; DATA XREF: sub_
                                        ; sub_1FC02+10↑o
```

```
gEfiGraphicsOutputProtocolGuid dq 4A3823DC9042A9DEh, 6A5180D0DE7AFB96h
                                        ; DATA XREF: sub_723F+2F↑o
                                        ; sub_8354+3E↑o ...
gEfiLoadedImageProtocolGuid dq 11D295625B1B31A1h, 3B7269C9A0003F8Eh
                                        ; DATA XREF: sub_2F0E+19↑o
                                        ; sub_1EADB+52↑o
gEfiLoadFileProtocolGuid dq 11D2954C56EC3091h, 3B7269C9A0003F8Eh
                                        ; DATA XREF: sub_2F0E+328↑o
gEfiSimpleFileSystemProtocolGuid dq 11D26459964E5B22h, 3B7269C9A000398E
                                        ; DATA XREF: sub_2F0E+135↑o
                                        ; sub_171C4+1E3↑o ...
gEfiSimplePointerProtocolGuid dq 11D50B7531878C87h, 4DC13F2790004F9Ah
                                        ; DATA XREF: sub_199D4+5E8↑o
                                        ; sub_1EBC6+49↑o ...
gEfiUgaDrawProtocolGuid dq 41CBF4FA982C298Bh, 39B88F68AA7738B8h
                                        ; DATA XREF: sub_723F+65↑o
                                        ; sub_1D1E3+71↑o ...
gEfiConsoleControlProtocolGuid dq 4C12012EF42F7782h, 21F70443F9495699h
                                        ; DATA XREF: sub_88C3+4E↑o
gEfiFirmwareVolumeDispatchProtocolGuid dq 444F506C7AA35A69h, 0C8716FF54
                                        ; DATA XREF: sub_C5F7+279↑o
qword_76708     dq 49016E735B213447h, 72A1B7F364B8F1A4h
                                        ; DATA XREF: sub_C5F7+87↑o
                                        ; sub_1FC02+10↑o
```

REVERSING EFI
KINDA SUCKS

```
00000C804 mov     rax, cs:gBootServices
00000C80B lea     rcx, [rbp+var_80]
00000C80F mov     [rsp+140h+var_120], rcx
00000C814 lea     rbx, gEfiFirmwareVolumeProtocolGuid
00000C81B xor     edi, edi
00000C81D lea     r9, [rbp+var_88]
00000C824 mov     ecx, 2
00000C829 mov     rdx, rbx
00000C82C mov     r13, rbx
00000C82F xor     r8d, r8d
00000C832 call    [rax+EFI_BOOT_SERVICES.LocateHandleBuffer]
00000C838 lea     rsi, [rbp+var_78]
00000C83C lea     r15, [rbp+var_5C]
00000C840 lea     r12, [rbp+var_5D]
00000C844 lea     r14, dword_7645C
00000C84B lea     rbx, [rbp+var_68]
00000C84F jmp     short loc_C854
```

```
000000000000C854
000000000000C854 loc_C854:
000000000000C854 cmp     rdi, [rbp+var_88]
000000000000C85B jnb     loc_C971
```

```
000000000000C861 mov     rax, [rbp+var_80]
000000000000C865 mov     rcx, [rax+rdi*8]
000000000000C869 mov     rax, cs:gBootServices
000000000000C870 lea     rdx, gEfiFirmwareVolumeDispatchProtocolGuid
000000000000C877 mov     r8, rsi
000000000000C87A call    [rax+EFI_BOOT_SERVICES.HandleProtocol]
```

III. DOING BAD THINGS WITH EFI

DOING BAD THINGS WITH EFI

WHAT CAN WE DO?

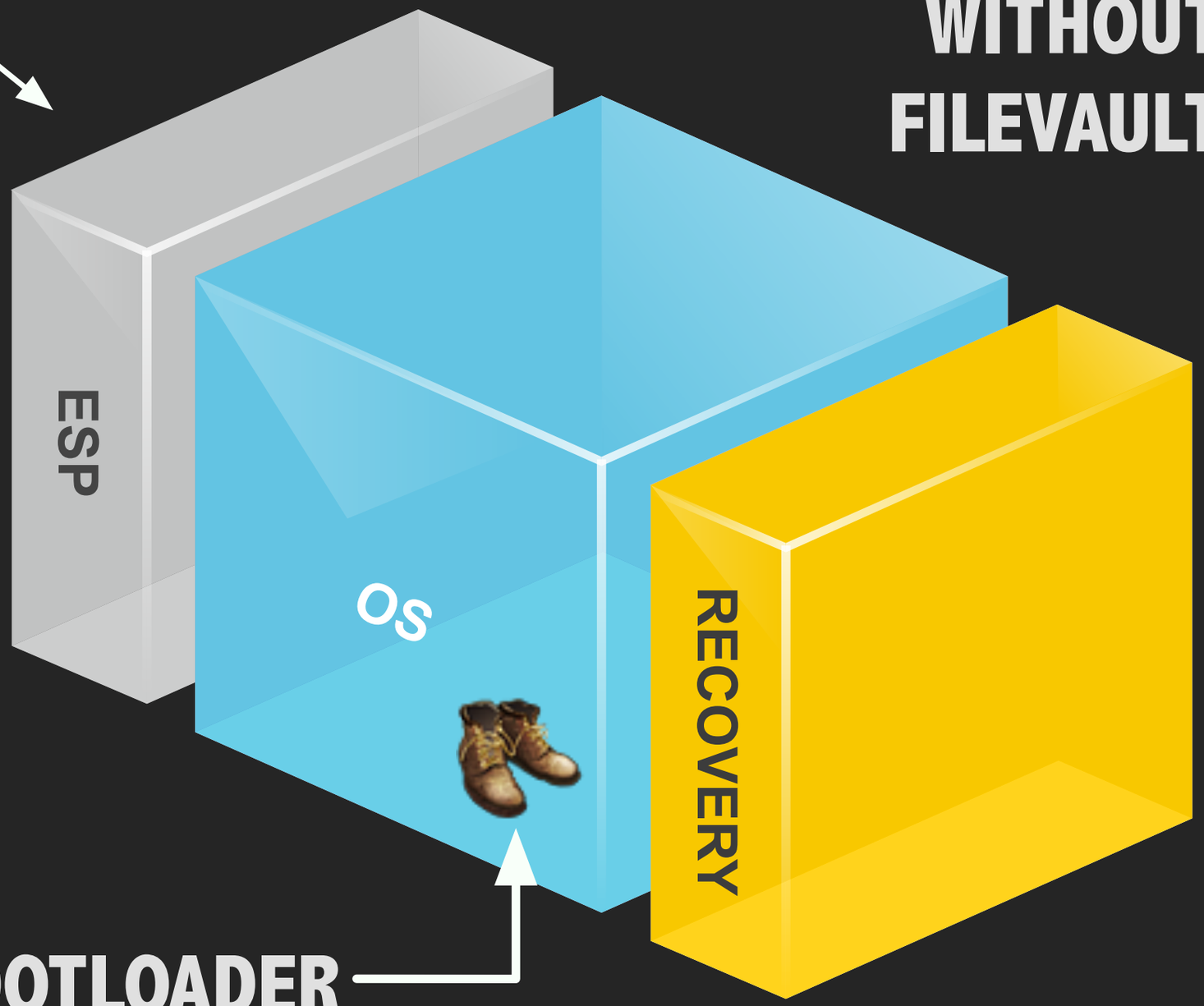
- Modularity & SDK makes it pretty easy
  - Build a rogue driver
  - Get loaded early on
  - Register callbacks
  - Hook Boot Services/Runtime Services
  - Hook various protocols
- No awful 16-bit real-mode assembly necessary
- Generic interface - minimal platform-specific stuff

DOING BAD THINGS WITH EFI

ATTACKING WHOLE-DISK ENCRYPTION

FANCY DIAGRAM OF HARD DISK

WITHOUT FILEVAULT

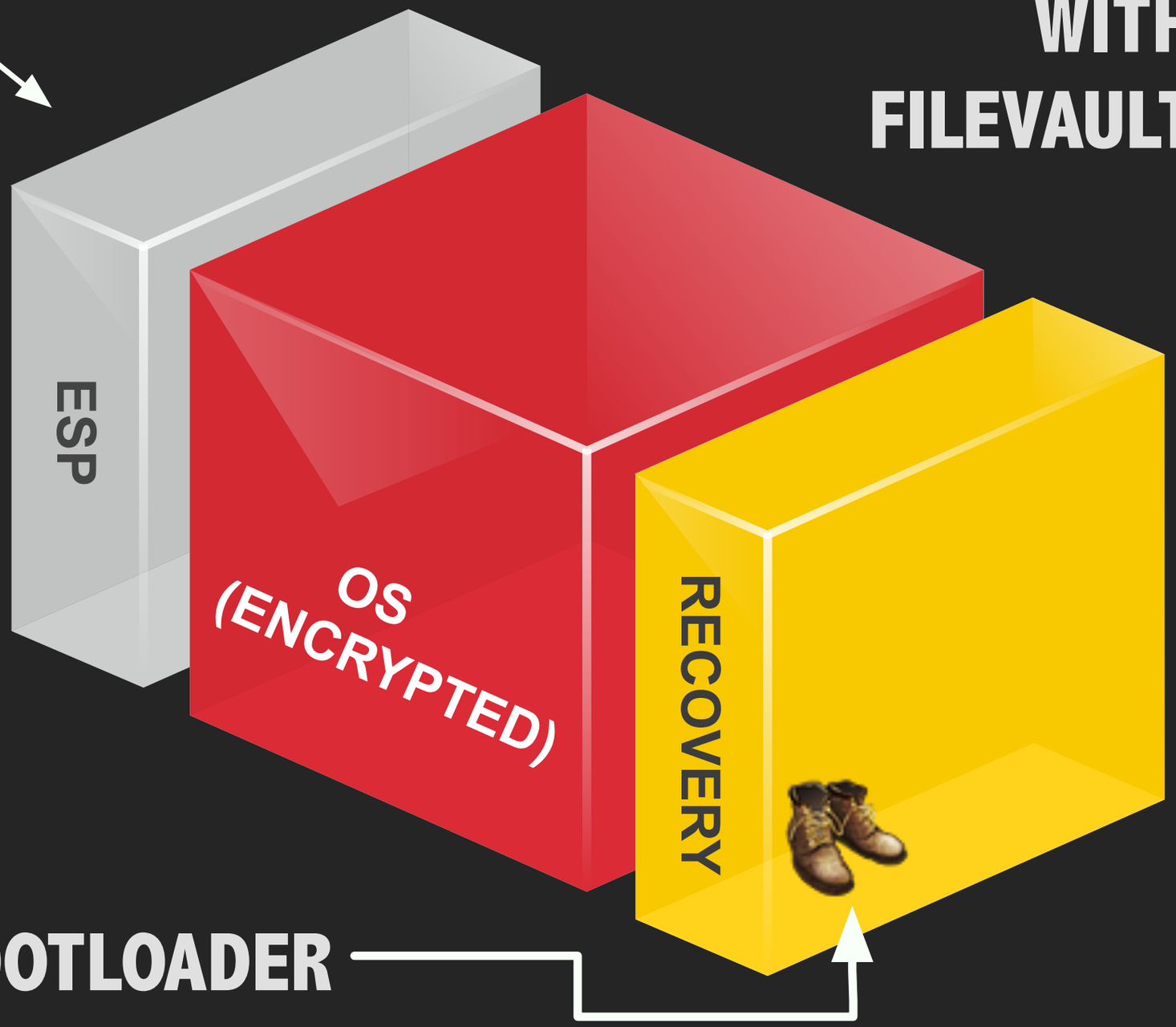


DOING BAD THINGS WITH EFI

ATTACKING WHOLE-DISK ENCRYPTION

FANCY DIAGRAM OF HARD DISK

WITH FILEVAULT



BOOTLOADER

DOING BAD THINGS WITH EFI
ATTACKING WHOLE-DISK ENCRYPTION

- Boot process with FileVault
  - Platform firmware inits
  - Loads bootloader from “recovery” partition
  - Bootloader prompts user for passphrase
  - Uses passphrase to unlock disk
  - Execute kernel

DOING BAD THINGS WITH EFI
ATTACKING WHOLE-DISK ENCRYPTION

- Stealing the user's passphrase
  - Keystroke logger!
  - Hook the Simple Text Input protocol
    - Specifically, the instance installed by the bootloader on the console device handle
    - Replace pointer to `ReadKeyStroke()` with our function
  - Every time a key is pressed, we get called
  - Record keystroke, call real `ReadKeyStroke()`

DOING BAD THINGS WITH EFI
ATTACKING WHOLE-DISK ENCRYPTION

- Steal the AES key
  - Hook `LoadImage()` function in Boot Services
  - Patch the bootloader when it is loaded
  - Shouldn't be toooooo hard...

```
aStartUnlockcor db 'Start UnlockCoreStorageVolumeKey',0
                                        ; DATA XREF: start+481↑o
                align 8
aEndUnlockcores db 'End UnlockCoreStorageVolumeKey',0
                                        ; DATA XREF: start+49F↑o
                align 8
```

(thanks for the debug logging, Apple)

THEY'RE GOING AFTER THE KERNEL!

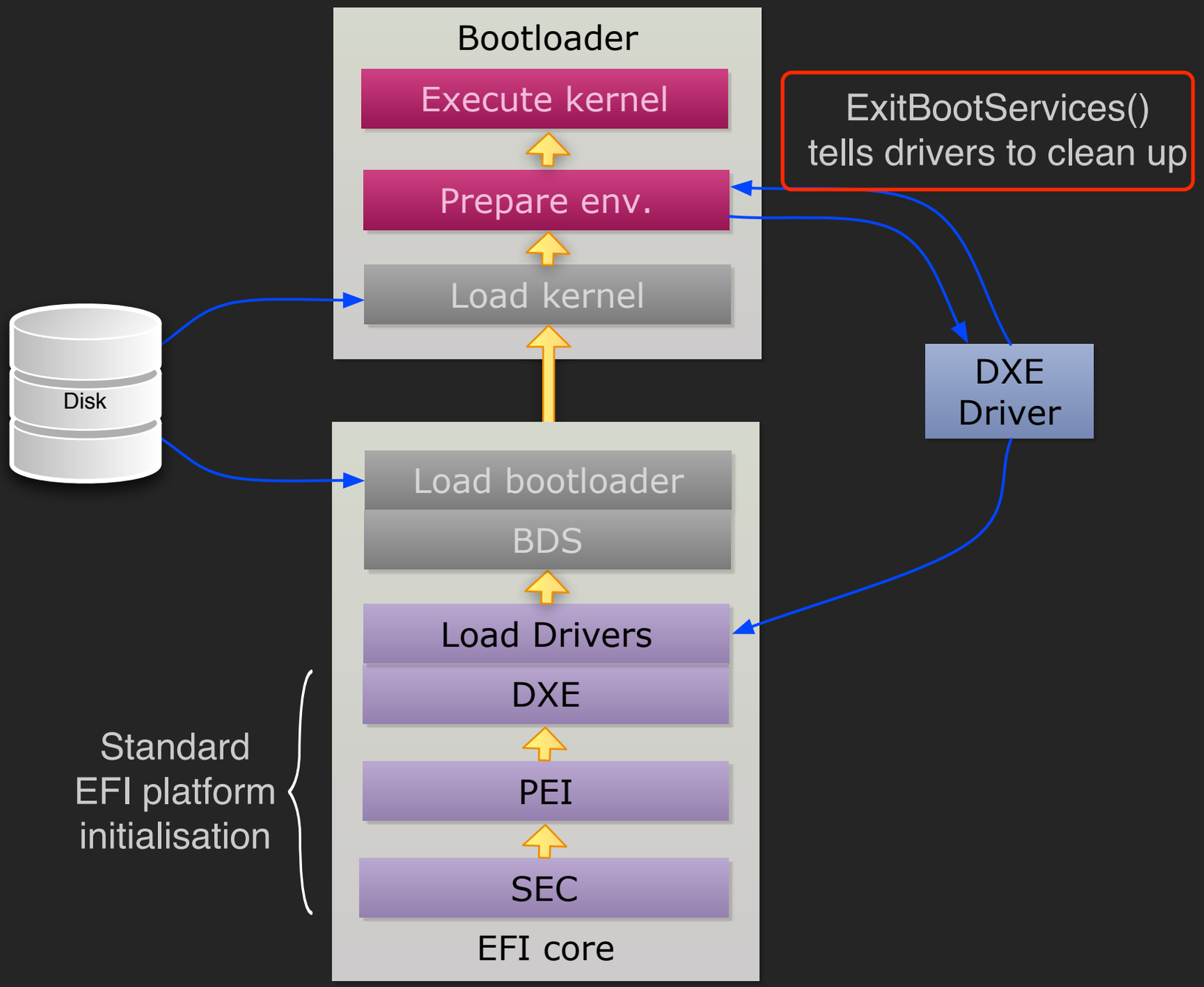
OTTERZ? IN *MY* KERNEL?

ATTACKING THE KERNEL
WHAT CAN WE DO?

- Patch the kernel from EFI
  - Find some place to put code
  - Hook some kernel functionality
  - Get execution during kernel init
  - Party
- It's not loaded when we get loaded
  - So how do we trojan the kernel?
  - Wait until it is loaded, then POUNCE
  - `ExitBootServices()`

ATTACKING THE KERNEL

EFI BOOT PROCESS



ATTACKING THE KERNEL

WHERE IS IT?

Start of kernel image is at `0xffffff8000200000`

```
$ otool -l /mach_kernel
/mach_kernel:
Load command 0
      cmd LC_SEGMENT_64
  cmdsize 472
  segname __TEXT
   vmaddr 0xffffff8000200000
   vmsize 0x000000000052e000

gdb$ x/x 0xffffff8000200000
0xffffff8000200000: 0xfeedfacf
```

First kernel segment VM load addr

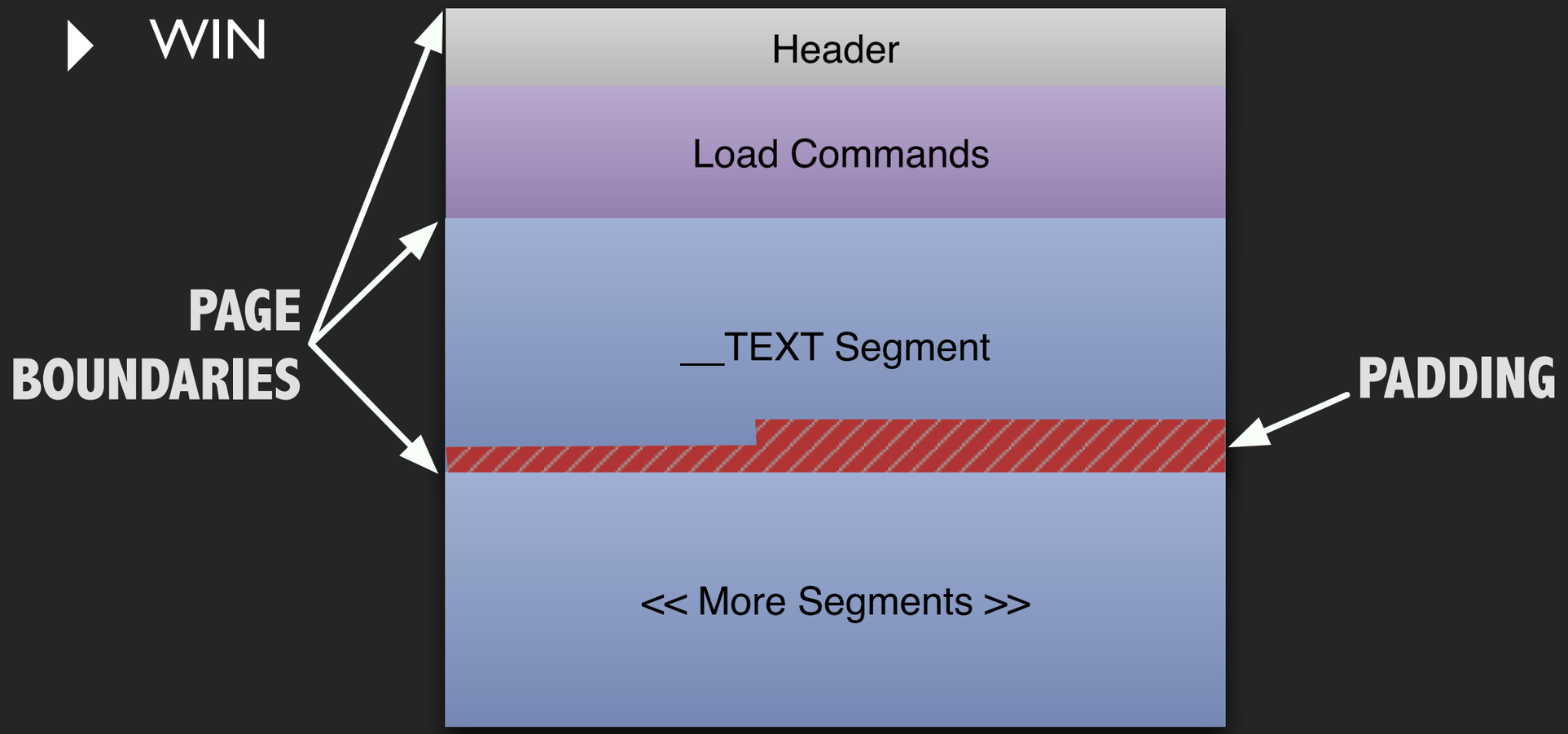
Mach-O header magic number (64-bit)

ATTACKING THE KERNEL
PATCHING THE KERNEL

- We know the kernel is at `0xffffff8000200000`
  - EFI uses a flat 32-bit memory model without paging
  - In 32-bit mode its at `0x00200000`
- What do we do?
  - Inject a payload somewhere
  - Patch a kernel function and point it at the payload
  - Trampoline payload to load bigger second stage?
    - From an EFI variable
    - From previously-allocated Runtime Services memory
    - Over the network

ATTACKING THE KERNEL
PATCHING THE KERNEL

- Where can we put our payload?
  - Page-alignment padding
    - End of the __TEXT segment
    - On the default 10.7.3 kernel, almost an entire 4k page
  - WIN



ATTACKING THE KERNEL
PATCHING THE KERNEL

- OK, so
  - We have been called by `ExitBootServices()`
  - We know where we can store a payload
    - And how much space we have
  - What do we put there?
  - And how do we get it called during kernel init?

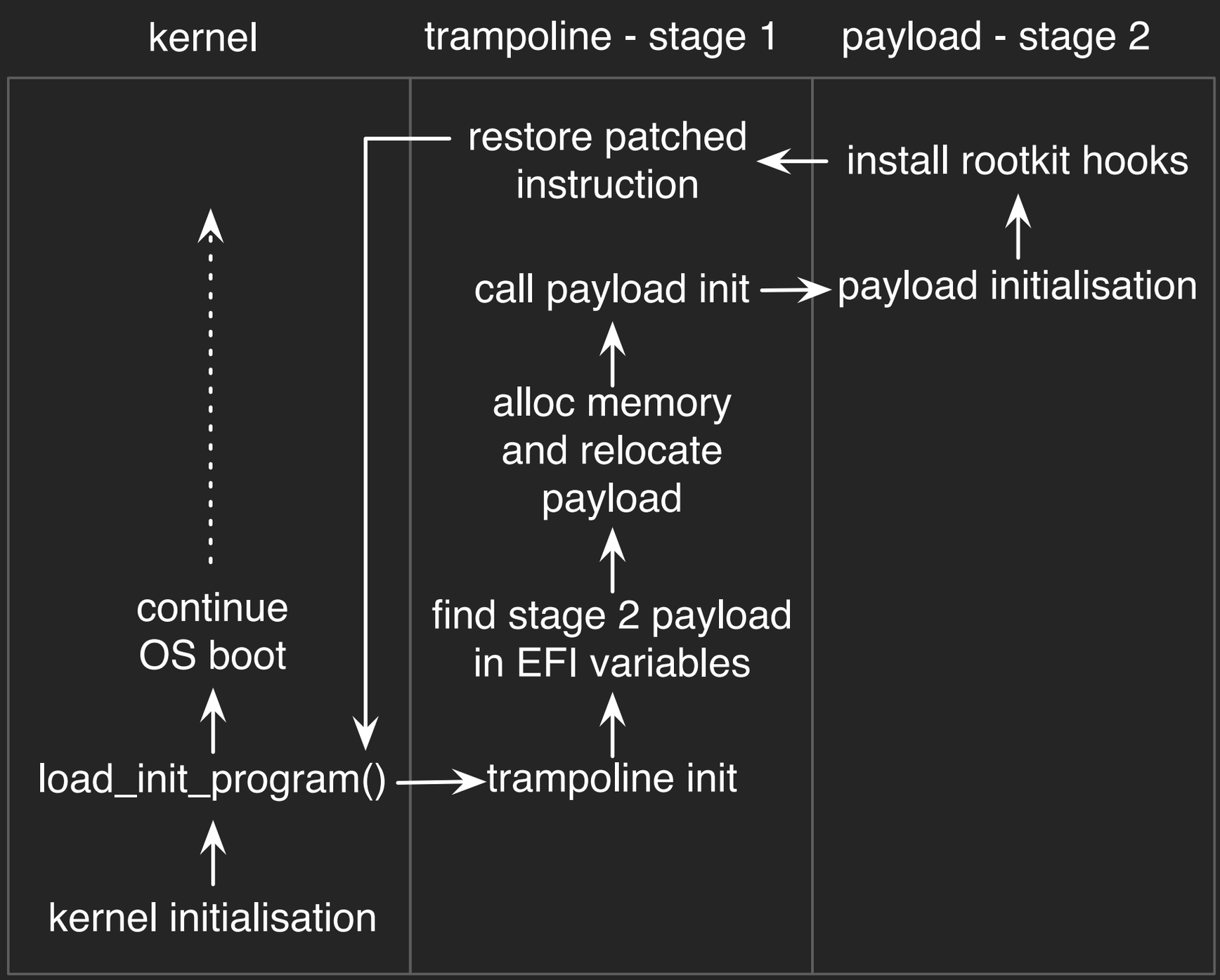
ATTACKING THE KERNEL
PATCHING THE KERNEL

- How do we get it called?
  - We patch a function in the kernel's boot process
  - `load_init_program()` is a good candidate
    - Kernel subsystems are mostly initialised
    - We're ready to exec the init process
  - Save the first instruction in the function, store in payload
  - Overwrite it with a jump to our payload

ATTACKING THE KERNEL
PATCHING THE KERNEL

- What's our payload? Tramampoline!
  - Save registers
  - Locate next stage payload
    - Stored in an EFI variable
  - Call next stage initialisation
  - Restore patched instruction
  - Restore registers
  - Jump back to patched func
  - Kernel continues booting

ATTACKING THE KERNEL
PATCHING THE KERNEL



ATTACKING THE KERNEL
PATCHING THE KERNEL

- Preparing our trampoline

```
/* We're going to patch the first instruction of load_init_program(), and
 * we need to jump back here */
tramp.patch_addr = find_kernel_symbol("_load_init_program");
DLOG(L"[+] patching load_init_program @ 0x%p\n", tramp.patch_addr);

/* Save the instruction data that we're going to overwrite. The tramp will
 * fix it up afterwards. */
tramp.patch_save = *((uint64_t *)tramp.patch_addr);
DLOG(L"[+] saved instructions: 0x%llx, \n", tramp.patch_save);

/* Overwrite the instruction with a jump to the trampoline shellcode */
jump.displacement = (uint32_t)sc_start - (uint32_t)tramp.patch_addr -
                    sizeof(jump);
*(uint64_t *)tramp.patch_addr = *(uint64_t *)&jump;
DLOG(L"[+] patched with instruction: 0x%llx\n", *(uint64_t *)&jump);
```

- Then we just copy it into the kernel

ATTACKING THE KERNEL
HALF-ASSED ROOTKIT HOOKS SLIDE

- What do we do once we're in the kernel?
  - Minimal detail here...
    - See my blog for previous talks on XNU rootkits, etc (http://ho.ax)
    - See fG's blog for more rad stuff (http://reverse.put.as)
  - Hook syscalls
  - Install NKE callbacks (socket/IP/interface filters)
  - Install TrustedBSD policy handlers
  - Patch things
  - ... and so on

ATTACKING THE KERNEL
OTHER HALF-ASSED ROOTKIT HOOKS SLIDE

- e.g. Hooking the `kill()` syscall
  - Demo will use this
  - Overwrite entry in `sysent` to point to our function
  - Our function...
    - Checks for a special condition (signal number == 7777)
      - Promotes the calling process to uid 0
    - Calls the original `kill()`

IV. PERSISTENCE

PERSISTENCE
OPTIONS?

- In ascending order of awesome
  - Patch/replace bootloader
  - EFI System Partition
  - PCI device expansion ROM
  - Firmware flash

Somewhat awesome

Pretty damn awesome

So awesome

PERSISTENCE
MESSING WITH THE BOOTLOADER

- /System/Library/CoreServices/boot.efi
- On-disk, why not just...
  - Patch the kernel
  - Install a kernel extension
- Somewhat useful for “evil maid” attacks
  - Even with FileVault, boot.efi is stored unencrypted
- Meh. 4/10.

PERSISTENCE
EFI SYSTEM PARTITION

- Not actually used by Apple's impl. by default
  - As far as I can tell
  - It is used to stage firmware updates
- Meh also. ~~1/10~~ 5/10
- I have revised my opinion on this
  - Possible to force Apple's EFI to use it
  - Set some NVRAM variables
  - Will demonstrate this

PERSISTENCE
PCI DEVICE EXPANSION ROMS

- Huh?
  - PCI bus is initialised
  - Devices are probed for expansion (“option”) ROMs
  - Found ROMs are mapped into memory
  - DXE phase loads any EFI drivers in ROMs
- Used for things like...
  - PXE on ethernet chipsets (hold that thought)
  - EFI/BIOS drivers for graphics hardware

PERSISTENCE
PCI DEVICE EXPANSION ROMS

- Hardware-specific
- Graphics cards in iMacs have them
  - MacBook Pros too
  - My old test MacBook - no dice
  - VMware's ethernet interfaces do - hurr (good for testing)
- Can write to them from the OS
  - Thanks, iMacGraphicsFWUpdate.pkg!
  - Probably with flashrom
- Pretty awesome. 7/10.

PERSISTENCE
FIRMWARE FLASH

- Hardware-specific, but it's always there
- Can modify everything
  - SEC, PEI, DXE, BDS, custom drivers, whatever
  - We can add/replace a driver in the volume
  - Re-flash it from the OS*/EFI with flashrom
- So awesome. 11/10 A++++ would buy again.
  - BUT...

PERSISTENCE
FIRMWARE FLASH

- Problems
  - Apple's boot ROM checks FV signature!
    - (Allegedly - this would explain my bricked test machine)
    - Might not be as easy as with PC hardware
  - Newer machines use WP flag on flash
    - Need to flash from early EFI stages (maybe SMM)
    - See Invisible Things Lab - "Attacking Intel BIOS"

PERSISTENCE
FIRMWARE FLASH

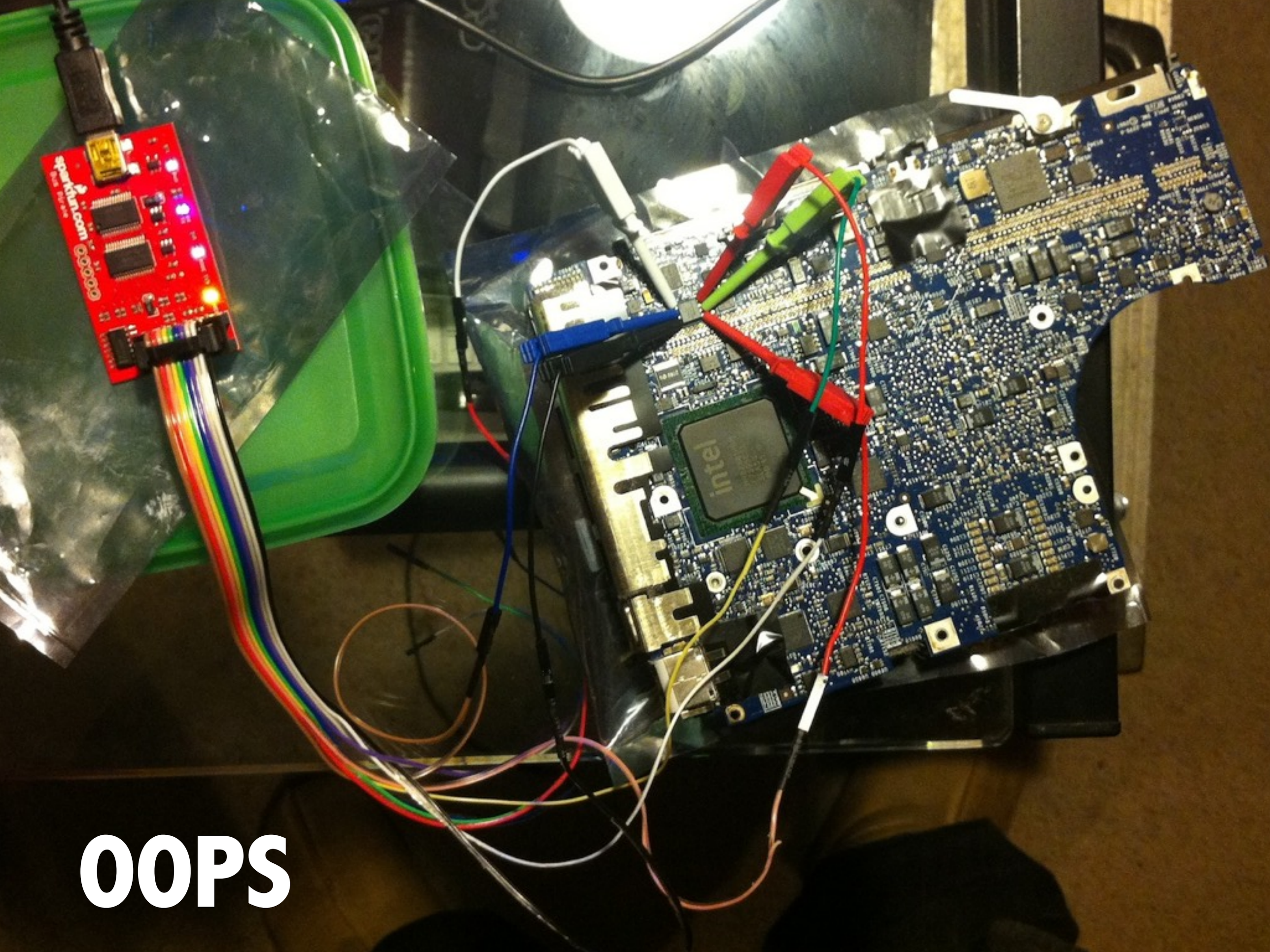
- Apple's firmware updates
  - Firmware updates are copied to ESP
  - Written to flash on reboot
  - Older machines use EFI Firmware Volumes (.fd files)
    - Volume is blessed with EfiUpdaterApp.efi
    - Writes to flash via SPI from EFI environment
  - Newer machines use EFI Capsules (.scap files)
    - EFI capsule mailbox stuff? (see the spec)

PERSISTENCE
FIRMWARE FLASH

- Manipulating firmware
  - Both capsules and firmware volumes are in the spec
    - http://download.intel.com/technology/framework/docs/Capsule.pdf
    - http://download.intel.com/technology/framework/docs/Fv.pdf
  - A capsule has a firmware volume inside
  - Inside the FV is a set of Firmware Filesystem “files”
    - http://download.intel.com/technology/framework/docs/Ffs.pdf
  - There are tools for manipulating Phoenix/AMI/etc BIOSes
    - Aimed at SLIC mods etc
  - I wrote my own in python
  - PS. Binaries are PE, remember? IDA understands them.

PERSISTENCE
FIRMWARE FLASH

```
[Firmware Volume]
  Offset = 0x0 (0)
  FileSystemGuid = 7a9354d9-0468-444a-81ce-0bf617d890df
  FvLength = 0x190000 (1638400)
  Signature = '_FVH'
  Attributes = 0xffff8eff
  HeaderLength = 0x48 (72)
  Checksum = 0xdefd (57085)
  Revision = 0x1 (1)
  [FvBlockMap]
    NumBlocks 25, BlockLength 65536
Files:
  11527125-78b2-4d3e-a0df-41e75c221f5a (EFI_FV_FILETYPE_PEIM)
  4d37da42-3a0c-4eda-b9eb-bc0e1db4713b (EFI_FV_FILETYPE_PEIM)
  35b898ca-b6a9-49ce-8c72-904735cc49b7 (EFI_FV_FILETYPE_DXE_CORE)
  c3e36d09-8294-4b97-a857-d5288fe33e28 (EFI_FV_FILETYPE_FREEFORM)
  bae7599f-3c6b-43b7-bdf0-9ce07aa91aa6 (EFI_FV_FILETYPE_DRIVER)
  b601f8c4-43b7-4784-95b1-f4226cb40cee (EFI_FV_FILETYPE_DRIVER)
  51c9f40c-5243-4473-b265-b3c8ffaff9fa (EFI_FV_FILETYPE_DRIVER)
---8<--snip--8<---
```



OOPS

V. EVIL MAID ATTACKS

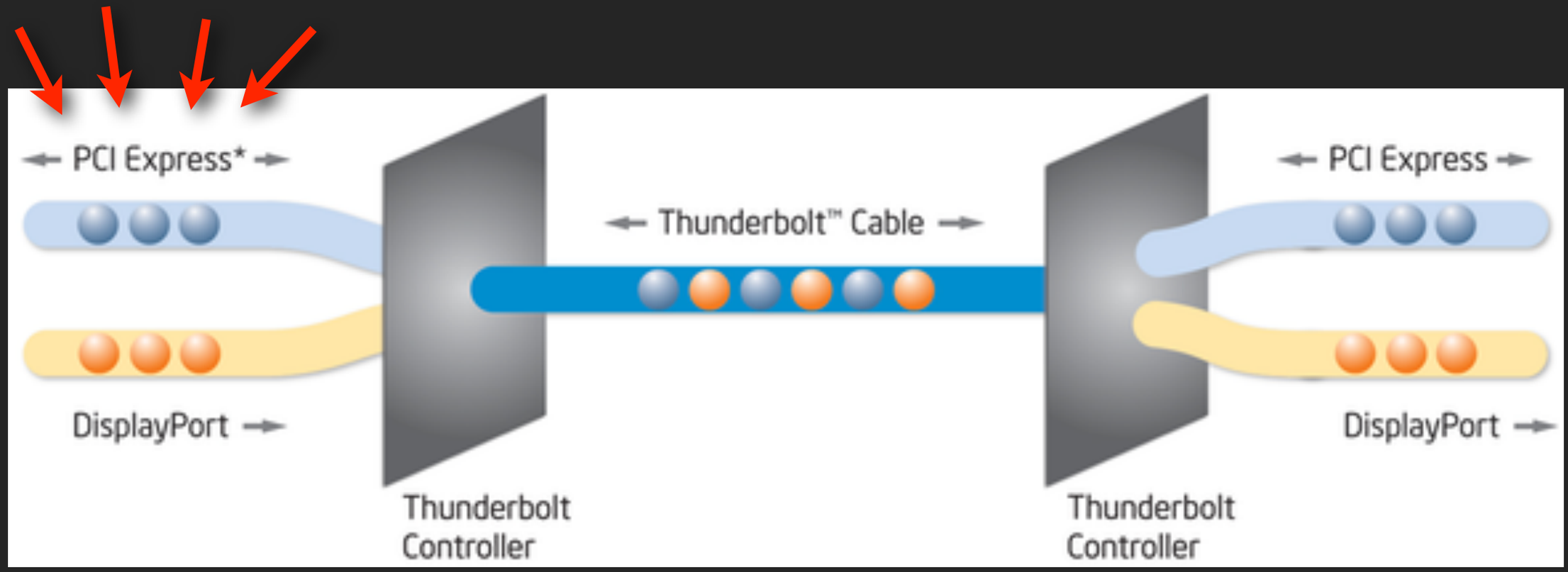
EVIL MAID ATTACKS
POSSIBILITIES?

- Change boot target
  - USB, Firewire, Network
  - Backdoor some things
- Remove disk and trojan
  - Patch bootloader
- Thunderbolt
  - ???

EVIL MAID ATTACKS

WHAT CAN WE DO WITH THUNDERBOLT?

MY GOOD FRIEND MARKETING DIAGRAM IS HERE TO EXPLAIN!



THUNDERBOLT DOES PCIe
PCIe DEVICES HAVE OPTION ROMS

EVIL MAID ATTACKS

IT'S MY JOB TO KEEP PUNK ROCK ELITE

Some ExpressCard SATA adapters have expansion ROMs

EVIL MAID ATTACKS
IT'S MY JOB TO KEEP PUNK ROCK ELITE

So does Apple's Thunderbolt to Ethernet adapter...

EVIL MAID ATTACKS
IT'S MY JOB TO KEEP PUNK ROCK ELITE

- Process
  - Attach Apple Thunderbolt to Gigabit Ethernet Adapter
  - Power on system
  - Driver is loaded from option ROM on adapter
  - Driver deploys payload
    - Drop shim bootloader and payload driver on disk
    - Flash payload driver to option ROM on video card
    - ...
  - All before the FileVault passphrase entry screen

EVIL MAID ATTACKS
THUNDERBOLT TO GIGABIT ETHERNET ADAPTER

```
08:00.0 Ethernet controller: Broadcom Corporation Device 1682
   Subsystem: Apple Computer Inc. Device 00f6
   Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr-
Stepping- SERR- FastB2B- DisINTx-
   Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort-
<MAbort- >SERR- <PERR- INTx-
   Latency: 0, Cache Line Size: 128 bytes
   Interrupt: pin A routed to IRQ 11
   Region 0: Memory at acb00000 (64-bit, prefetchable) [size=64K]
   Region 2: Memory at acb10000 (64-bit, prefetchable) [size=64K]
   Expansion ROM at acb20000 [disabled] [size=64K]
   --snip--
```

This guy right here, man

```
$ EfiRom -f 0x0001 -i 0x8003 -e defile.efi -o defile.rom
```

EVIL MAID ATTACKS

THUNDERBOLT TO GIGABIT ETHERNET ADAPTER

```
 Copyright(c) 2000-2011 Broadcom Corporation, all rights reserved.
 Broadcom NetXtreme/NetLink User Diagnostics 15.23 (12/16/11)
*******************************************************************************

C Brd    Rv    Bus    PCI Spd Base  Irq NVM(avl/max)   MAC          Boot Code  Config
- -------- ------- --- --- ----  --  ---------- ----------- ---------- -------
0 57762:A0 00:00:0 Ex1 250 ACD0  11    64k/  64k 406C8F35D639 57762-a1.10 WMp,auto

Checking IRQ.....................: passed
Checking NVRAM Content...........: passed
Programming PXE from defile.rom..: Updating PCI ROM (type 3) header with Ve
ndor ID = 0x14e4 Device ID = 0x1682
EFI Reading current NVRAM ... OK
Programming...      512
                                   passed
Checking Bond Id.................: passed
Manufacturing revision...........: D
Boot Code Version................: 57762-a1.10
Mac Address......................: 40-6C-8F-35-D6-39
NVRAM Size in KBytes.............: 64/0x40
TPM Size in KBytes...............: 0/0x0
Group A. Register Tests
 A1. Indirect Register Test_
```

DEMO

- Evil maid using emulated option ROM (VMware)
  - Assume we've already got privileged access
    - WebKit vuln + privesc to root?
    - From here we've flashed an option ROM on the graphics card?
  - Boot machine
  - Malicious driver is loaded from option ROM
    - For demonstration purposes it's on the virtual ethernet adapter
  - Driver registers for ExitBootServices()
  - Bootloader is executed, loads kernel
  - Driver gets called back by ExitBootServices() and patches the kernel
  - Kernel is booted

DEMO

- Evil maid using Thunderbolt
  - Connect Apple Thunderbolt to Gigabit Ethernet Adapter
  - Boot machine
  - Driver is loaded from adapter's PXE firmware & exec'd
  - Driver writes a shim bootloader & driver to the ESP
  - Configures NVRAM to point to that bootloader
  - On subsequent boots:
    - Shim bootloader is loaded
    - Loads payload driver, which registers ExitBootServices()
    - Shim execs normal bootloader
    - Driver patches kernel

DEMO

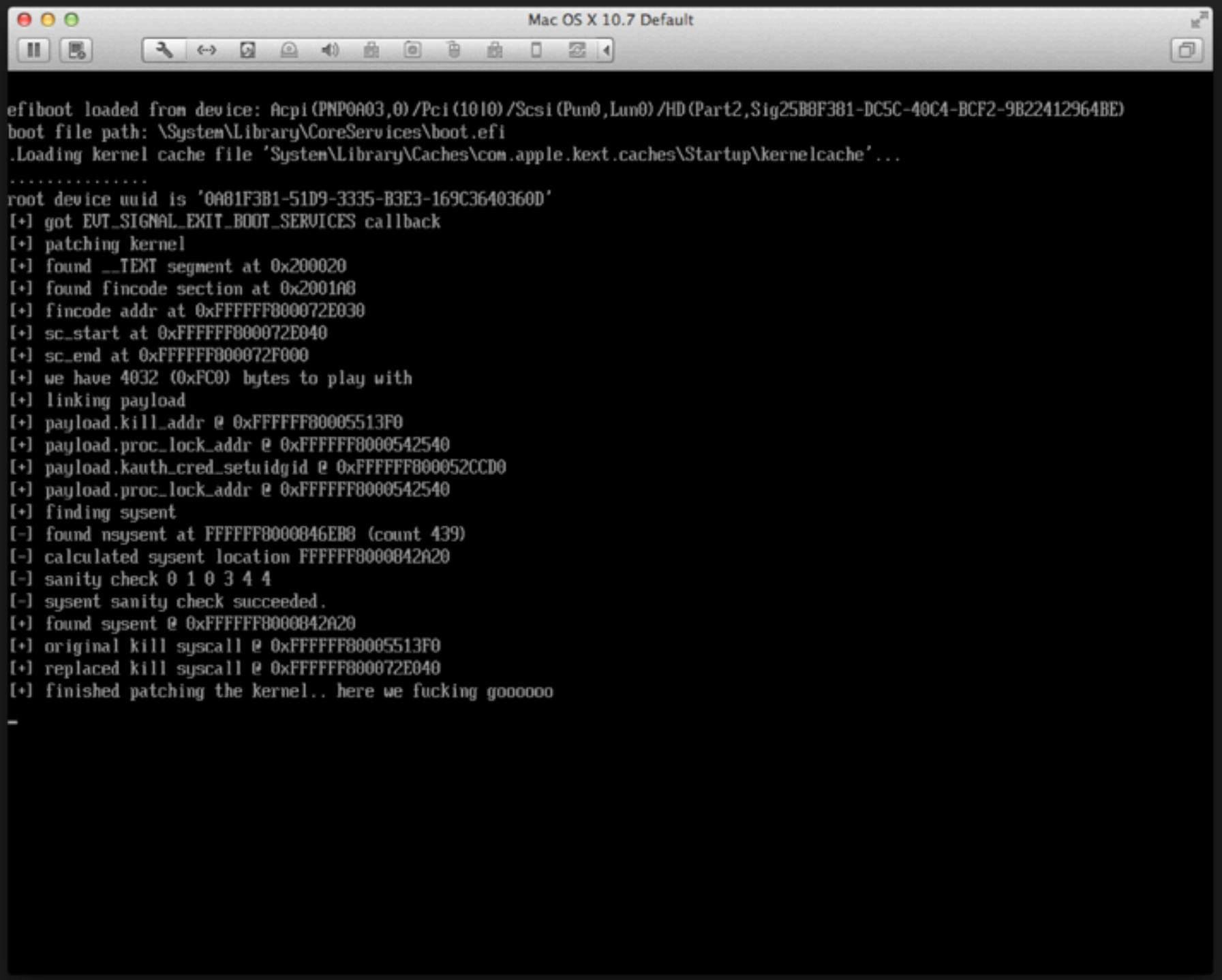

This is real.
System report:
Everything is fine. Nothing is ruined.
ok
COMPY 386

IN CASE MY DEMO BROKE

HERE'S SOME SCREENSHOTS

IN CASE MY DEMO BROKE

HERE'S SOME SCREENSHOTS



V. DEFENCE

DEFENCE
EFI FIRMWARE PASSWORD?

- Hahaha... :(
  - This will prevent some “evil maid” attacks
  - Stops you from changing the boot target
    - USB/Optical/Firewire/Network
  - That’s about it
  - Doesn’t prevent flashing the firmware/option ROMs
  - Doesn’t prevent loading drivers over Thunderbolt
  - There are ways to remove it on some machines
    - Weird RAM removal tricks
    - Not so much with current Macs with soldered-on RAM

DEFENCE
UEFI SECURE BOOT

- Part of the current UEFI spec
- Describes signing of EFI images (drivers/apps/loaders)
  - Platform Key (PK)
  - Key Exchange Key (KEK)
- DXE & BDS phases verify sigs of binaries

DEFENCE
UEFI SECURE BOOT

- Issues
  - *"The public key must be stored in non-volatile storage which is tamper and delete resistant."*
    - May not prevent evil maid attacks if NVRAM can be reset
    - Blank NVRAM == back to "setup" mode
    - Use the TPM
  - Signing needs to be enforced through the whole stack
  - More?

IN CONCLUSION...

I HAD FUN.

- So basically we're all screwed
  - What should you do?
    - Glue all your ports shut
    - Use an EFI password to prevent basic local attacks
    - Stop using computers, go back to the abacus
  - What should Apple do?
    - Implement UEFI Secure Boot (actually use the TPM)
    - Disable loading of option ROMs from devices on bus expansions
      - When an EFI password is set I guess?
    - Audit the damn EFI code (see Heasman/ITL)
    - Sacrifice more virgins

REFERENCES

- UEFI Spec
  - http://www.uefi.org/specs/
- EFI Development Kit II source & documentation
  - http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2
- Mac OS X Kernel Programming Guide
  - http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/
- Mac OS X Internals - Amit Singh
  - http://osxbook.com/
- Mac OS X Wars: A XNU Hope - nemo
  - http://www.phrack.com/issues.html?issue=64&id=11#article
- Runtime Kernel kmem Patching - Silvio Cesare
  - http://biblio.l0t3k.net/kernel/en/runtime-kernel-kmem-patching.txt
- Designing BSD Rootkits - Joseph Kong
  - http://nostarch.com/rootkits.htm
- Reverse Engineering Mac OS X - fG
  - http://reverse.put.as/
- Hacking The Extensible Firmware Interface - John Heasman
  - https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf
- Attacking Intel BIOS - Invisible Things Lab
  - http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

KTHXBAI \m/

twitter: @snare
blog: http://ho.ax

greetz:
y011, wily, fG! & #osxre, metlstorm, sharrow, nemo

special thanks to:
ruxcon dudes!
jesse for the test machine that I bricked (and resurrected)
baker & glenno for the brutal art

http://www.assurance.com.au