DE MYSTERIIS DOM JOBSIVS: MAC EFI ROOTKITS

SNARE
@ BLACK HAT USA
JULY 2012

assurance

Please complete the speaker feedback surveys
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.
    - Sup argp

De Mysteriis Dom Jobsivs - Black Hat USA
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
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...
Other work in this area

- Old MBR viruses
- ... (omitted content)
- 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, reinstall, 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 - this is very much ongoing research)
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 “SU CK” 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
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
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

De Mysteriis Dom Jobsivs - Black Hat USA
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
ESP
OS
RECOVERY
BOOTLOADER
WITHOUT FILEVAULT

De Mysteriis Dom Jovis - Black Hat USA
DOING BAD THINGS WITH EFI
ATTACKING WHOLE-DISK ENCRYPTION

FANCY DIAGRAM OF HARD DISK

ESP
OS (ENCRYPTED)
BOOTLOADER

WITH FILEVAULT
RECOVERY

De Mysteriis Dom Jobsivs - Black Hat USA
2012
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
  - 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 tooooo hard...

(thanks for the debug logging, Apple)
(also, that’s my one token IDA screenshot)
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

Bootloader
- Execute kernel
- Prepare env.
- Load kernel

Standard EFI platform initialisation

EFI core
- Load bootloader
  - BDS
- Load Drivers
  - DXE
  - PEI
  - SEC
- ExitBootServices() tells drivers to clean up

DXE Driver

Disk

De Mysteriis Dom Jovis - Black Hat USA
ATTACKING THE KERNEL
WHERE IS IT?

Start of kernel image is at \texttt{0xffffffff8000200000}

\$ otool -l /mach_kernel
/mach_kernel:
Load command 0
  cmd \texttt{LC\_SEGMENT\_64}
  cmdsize 472
  segname \texttt{\_TEXT}
  vmaddr \texttt{0xffffffff8000200000}
  vmsize \texttt{0x000000000052e000}

\texttt{gdb}\$ x/x \texttt{0xffffffff8000200000}
\texttt{0xffffffff8000200000}: \texttt{0xfeedfacf}

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

De Mysteriis Dom Jobsivs - Black Hat USA 2012
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? Trampoline!
  - 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

<table>
<thead>
<tr>
<th>kernel</th>
<th>trampoline - stage 1</th>
<th>payload - stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue OS boot</td>
<td></td>
<td>install rootkit hooks</td>
</tr>
<tr>
<td>load_init_program()</td>
<td>restore patched instruction</td>
<td>payload initialisation</td>
</tr>
<tr>
<td>kernel initialisation</td>
<td>call payload init</td>
<td></td>
</tr>
<tr>
<td></td>
<td>alloc memory and relocate payload</td>
<td></td>
</tr>
<tr>
<td></td>
<td>find stage 2 payload in EFI variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>trampoline init</td>
<td></td>
</tr>
</tbody>
</table>
Preparing our trampoline

```c
/* 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()`
PERSISTENCE OPTIONS?

- In ascending order of awesome
  - Patch/replace bootloader (Somewhat awesome)
  - EFI System Partition (Pretty damn awesome)
  - PCI device expansion ROM
  - Firmware flash (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 implementation
  - As far as I can tell
  - It is used to stage firmware updates
- Meh also. 1/10.
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
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
- Can be written to from the OS
  - flashrom
- So awesome. 11/10 A+++ would buy again.

De Mysteriis Dom Jobsivs - Black Hat USA 2012
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)
Manipulating firmware

- Both capsules and firmware volumes are in the spec

- A capsule has a firmware volume inside

- Inside the FV is a set of Firmware Filesystem “files”

- 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.
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<--
PERSISTENCE
FIRMWARE FLASH

- Manipulating firmware
  - We can add/replace a driver in the volume
  - Re-flash it from the OS with flashrom

- Problems
  - Apple’s boot ROM (pre-EFI) checks FV signature!
    - (Allegedly - this would explain my bricked test machine)
    - Might not be as easy as with commodity hardware
  - Newer machines use WP flag on flash
    - Need to flash from early EFI stages
    - See Invisible Things Lab - “Attacking Intel BIOS”
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
Some ExpressCard SATA adapters have expansion ROMs
So does Apple’s Thunderbolt to Ethernet adapter...
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]

This guy right here, man

$ EfiRom -f 0x0001 -i 0x8003 -e defile.efi -o defile.rom
<table>
<thead>
<tr>
<th>C Brd</th>
<th>Rv</th>
<th>Bus</th>
<th>PCI</th>
<th>Spd</th>
<th>Base</th>
<th>Irq</th>
<th>NVM (avl/max)</th>
<th>MAC</th>
<th>Boot Code</th>
<th>Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>57762-A0</td>
<td>08:00:0</td>
<td>Ex1</td>
<td>250</td>
<td>AC00</td>
<td>11</td>
<td>64k/64k</td>
<td>406C8F35D639</td>
<td>57762-a1.10</td>
<td>WMp, auto</td>
</tr>
</tbody>
</table>

Checking IRQ: passed
Checking NVRAM Content: passed
Programming PXE from defile.rom: passed
Updating PCI ROM (type 3) header with Vendor ID = 0x14e4 Device ID = 0x1682
EFI Reading current NVRAM ... OK
Programming... 512 passed

Checking Bond Id: passed
Manufacturing revision: 0
Boot Code Version: 57762-a1.10
Mac Address: 40-6C-8F-35-D6-39
NVRAM Size in KBytes: 64/0x48
TPM Size in KBytes: 0/0x0

Group A. Register Tests
A1. Indirect Register Test
Evil maid using USB

- Boot from USB flash drive
- Shim.efi loads malicious driver
- Driver registers for ExitBootServices()
- Shim.efi finds real bootloader via NVRAM
- Executes bootloader
- Driver gets called back by ExitBootServices() and patches the kernel
- Evil maid using Thunderbolt
- Connect Apple Thunderbolt to Gigabit Ethernet Adapter
- Boot machine
- Driver is loaded from adapter
- ... same as before
This is real.

System report:
Everything is fine. Nothing is ruined.

OK
IN CASE MY DEMO BROKE
HERE'S SOME SCREENSHOTS
IN CASE MY DEMO BROKE
HERE’S SOME SCREENSHOTS

De Mysteriis Dom Jobsivs - Black Hat USA
V. DEFENCE
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
- 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?
I HAD FUN.

So basically we’re all screwed

- 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  

- Mac OS X Kernel Programming Guide  

- 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  

- Attacking Intel BIOS - Invisible Things Lab  
  http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf
KTHXBAI \m/

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

Please complete the speaker feedback surveys

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

special thanks to:
the black hat crew for a great con/party
jesse for the test machine that I bricked (and resurrected)
baker & glenno for the brutal art

http://www.assurance.com.au