DE MYSTERIIS DOM JOBSIVS
Mac EFI Rootkits

Black Hat USA 2012

Loukas K (snare)
<loukask@assurance.com.au>

assurance

Table of Contents

Introduction

Attacks against PC firmware have been a threat since the early days of malware, beginning with the venerable MBR virus and quickly moving on to more advanced attacks. In 1998 we saw the CIH/Chernobyl malware infect many systems around the world, rendering some systems completely unbootable by corrupting the system's BIOS. In more recent times we have seen proof-of-concept rootkits (such as IceLord and Rakshasa[1]), and malware in the wild (such as Mebromi) that are able to overwrite the BIOS with a malicious version that enables the malware to persist in the system and interfere with the boot process. This type of malware can persist solely in the BIOS EEPROM on the motherboard, without requiring the storage of any files on the system's internal hard disk. This means that the malware can persist across operating system reinstalls, disk formats, and even the replacement of the hard disk.

With the advent of the Extensible Firmware Interface (EFI), malware developers are given new opportunities to infect a wide range of new systems. A detailed specification, common reference implementation upon which most vendor implementations are based, and a full-featured development kit enable both legitimate firmware developers and malware developers alike to build cross-platform code with much greater ease than developing for the legacy BIOS.

Apple was one of the earliest adopters of the EFI firmware when they utilised it in their range of Intel-based Macs beginning in 2006. Apple's EFI implementation includes support for a number of common hardware components used in Mac systems, and Mac-specific features like the HFS+ filesystem, but is still based on the same specification and reference implementation as other vendors. More recently, EFI has been implemented by a number of PC motherboard vendors to replace the legacy PC BIOS, further highlighting the possibility for attacks against EFI.

This paper discusses the current state of EFI-based malware, and how it may be implemented in order to attack Apple Mac systems. In the presentation accompanying this paper, proof-of-concept attacks will be demonstrated that utilise a number of the techniques discussed herein.

Background

What is EFI?

In 1998, Intel began a project initially known as the Intel Boot Initiative to develop a specification for a replacement for the PC BIOS, in an attempt to address some of its limitations. This project was eventually renamed EFI (Extensible Firmware Interface) and was developed by Intel until 2005 (EFI version 1.10), at which point it was handed over to a community group, the Unified EFI Consortium, and renamed UEFI (Unified Extensible Firmware

Interface). Alongside the development of the specification, Intel developed a reference implementation called the *Intel Platform Innovation Framework*, codenamed "Tiano", and also known as “the Framework”. Tiano is the "preferred" implementation according to Intel, and it is the basis on which most IBVs (independent BIOS vendors) build their own implementation of the specification.

When Apple began manufacturing hardware using the Intel x86 CPU architecture in 2006, they also adopted EFI in favour of OpenFirmware, which they were previously using on their PowerPC-based hardware. Apple's EFI implementations are based on version 1.10 of the EFI specification, and presumably the same version of Intel's reference implementation.

EFI architecture & boot process

An EFI environment comprises a number of components – EFI core modules (SEC, PEI, DXE and BDS), drivers, applications and bootloaders. Generally, an EFI firmware image contains the core modules and a set of drivers for supporting at least the core hardware on the motherboard. It may also contain other common drivers, or applications such as the EDK Shell, a command shell for interacting with the EFI pre-boot environment. Apple's EFI implementations differ, as expected, from machine to machine depending on the hardware used in each type of system.

When an EFI system is powered on, the SEC (Security) phase of EFI is the first code that is executed within EFI. This phase serves as a root of trust for the system and handles platform reset events, among other things. The SEC phase hands off to the PEI (Pre-EFI Initialisation) phase, which is responsible for initialising the CPU and main memory, before handing execution off to the DXE phase.

The DXE (Driver eXecution Environment) phase is where the majority of the system initialisation takes place. First, the DXE core produces a set of Boot and Runtime Services. Boot Services provide drivers, applications and bootloaders that run within the EFI environment with a number of services such as allocating memory and loading executable images. Runtime Services provide services such as converting memory addresses from physical to virtual during the handover to the kernel, and resetting the CPU, to code running within the EFI environment or within the OS kernel once it has taken control of the system. Once these services have been established, the DXE dispatcher discovers and executes drivers from the firmware volume, expansion ROMs on devices connected to the PCIe bus, and connected disks.

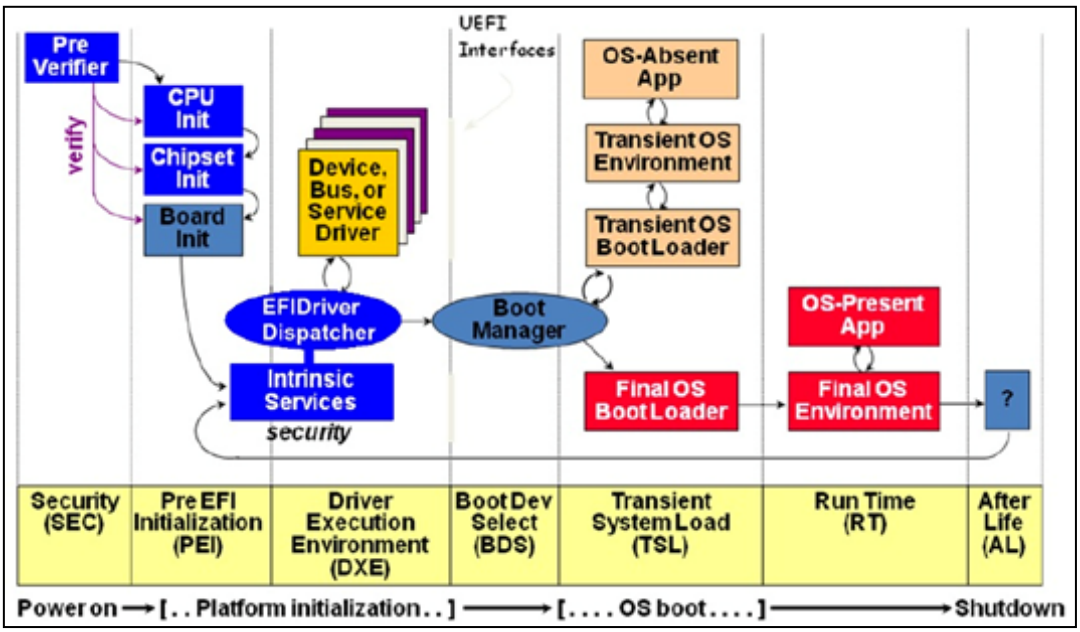
When drivers are initialised they register “protocols”, which are blocks of pointers to functions and data structures that serve as the interface to the driver. The UEFI specification defines a number of core protocols that provide some of the main services like console input and output (Simple Text Input Protocol, Simple Text Output Protocol and Graphics Output Protocol), media access (Simple File System Protocol, EFI File Protocol, Disk I/O Protocol, etc), PCI bus support (PCI Root Bridge I/O Protocol, PCI I/O Protocol, etc),

USB support (USB2 Host Controller Protocol and USB I/O Protocol), a series of network-related protocols, and many more. See the UEFI Specification[2] for a complete list of these protocols and the detail of their implementation.

Drivers can register for a number of notifications of events that occur within the EFI pre-boot environment. For example, a driver can request that it be notified whenever new protocols are installed on device handles, or it can request to be notified when the `ExitBootServices()` function is called to prepare the environment for the execution of the kernel.

Once the DXE phase has loaded and executed all the necessary drivers, it hands off execution to the BDS (Boot Device Selection) phase. This phase is responsible for discovering the possible boot devices, selecting one to boot from, loading the bootloader and executing it. On a Mac, when a boot device is selected as the default to boot from, the device path is stored in the system's NVRAM. When the BDS phase is executed, it locates the disk using this NVRAM data, locates the bootloader using the HFS+ volume header, and executes it.

The bootloader is responsible for loading the kernel and executing it. Prior to executing the kernel, the bootloader calls the `ExitBootServices()` function from the Boot Services table, which informs EFI that it should prepare the environment for the kernel to take over control of the system. During this preparation, drivers who have registered for the `ExitBootServices()` event are notified so that they can free unnecessary memory and perform any other clean up tasks prior to the kernel's execution.



The EFI boot process.

Developing for EFI

The open source part of Tiano is the EFI Development Kit (EDK), which contains the framework's foundation code and some sample drivers. The current version of the EDK is EDK2 and is available for download from

SourceForge[3]. EDK2 can be used in conjunction with a standard development toolchain to build drivers, applications and bootloaders for execution within an EFI environment. The majority of the EDK2 is written in C, with some assembly language components for various platforms, and some additional tools written in Python, Bash and other languages.

EFI components are developed in C, whereas modifications for the legacy PC BIOS generally need to be written in assembly language. A number of platforms can be targeted from this code, often without resorting to a great deal of low-level, platform-specific implementation (obviously this depends on the particular application). This process is further assisted by the EFI Byte Code format (EBC), which can be run on any EFI implementation. This is helpful to malware developers as they can easily deploy universal malware to target various different platforms.

EFI uses a modified subset of the PE32+ format for its executable images, which is a common executable format used by Microsoft Windows that many tools can generate and parse. This is also helpful for reverse engineering efforts, as common tools used for reverse engineering can understand and parse this format. IDA Pro, an advanced tool for disassembling binary images, can parse PE32+ and disassemble the EBC format.

Attacks using EFI

EFI's flexibility is a boon to driver and malware developers alike, as it makes building modular code that can be loaded and executed on a wide variety of EFI implementations much simpler than targeting the traditional PC BIOS. The simplest way to deploy malicious code for execution within the EFI environment is to build an EFI DXE driver that attacks the system, rather than supporting hardware. Once it is loaded into the system, such a driver can interfere with the boot process by hooking various protocols in the pre-boot environment, or by gaining execution within the context of the bootloader and patching the kernel prior to its execution.

Attacking FileVault

Deploying Apple's full-disk encryption implementation, FileVault, slightly changes the boot process. On a non-FileVault system, the bootloader (*boot.efi*) is stored on the main OS partition in */System/Library/CoreServices*. On a FileVault-encrypted system, the main OS partition is encrypted and cannot be accessed by the early stages of EFI in order to load the bootloader. As such, when FileVault is enabled the bootloader is relocated to the "recovery" partition, which is a partition at the end of the partition table otherwise used to boot the system into recovery mode for troubleshooting or reinstallation. When the system is booted, the bootloader is loaded from the recovery partition and presents the passphrase entry screen. The bootloader then uses the passphrase to "unlock" the CoreStorage volume on the primary OS partition and continue the boot process. An in-depth analysis of FileVault and the CoreStorage format has recently been undertaken by researchers[4].

It is recommended that this be referred to for more information on the internals of FileVault.

One way in which FileVault can be attacked from an EFI driver is by employing a traditional key-logging technique in order to capture the passphrase as it is entered by the user. Keystrokes are processed in the EFI pre-boot environment by the Simple Text Input protocol, which defines a function called `ReadKeyStroke()`:

```
_EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
   EFI_INPUT_RESET Reset;
   EFI_INPUT_READ_KEY ReadKeyStroke;
   EFI_EVENT WaitForKey;
} EFI_SIMPLE_TEXT_INPUT_PROTOCOL;
```

The relevant instance of this protocol is the one installed on the console device handle, and referred to by the EFI System Table's `ConIn` variable:

```
typedef struct {
   EFI_TABLE_HEADER Hdr;
   CHAR16 *FirmwareVendor;
   UINT32 FirmwareRevision;
   EFI_HANDLE ConsoleInHandle;
   EFI_SIMPLE_TEXT_INPUT_PROTOCOL *ConIn;
   EFI_HANDLE ConsoleOutHandle;
   EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *ConOut;
   EFI_HANDLE StandardErrorHandle;
   EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;
   EFI_RUNTIME_SERVICES *RuntimeServices;
   EFI_BOOT_SERVICES *BootServices;
   UINTN NumberOfTableEntries;
   EFI_CONFIGURATION_TABLE *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

The instance of the Simple Text Input Protocol that is initially installed and assigned to `ConIn` is replaced by the bootloader when the passphrase entry screen is called. As such, we need a way to update this new instance of the protocol before the passphrase is entered. In order to do so we can register for notifications when new protocols are installed on device handles, using the `RegisterProtocolNotify()` boot service. When the event is triggered and our driver is notified, we can save a pointer to the `ReadKeyStroke()` function in the Simple Text Input protocol instance, and overwrite the pointer with a pointer to our own function. When our `ReadKeyStroke()` function is called, we simply save the key that was pressed into a buffer and call the real `ReadKeyStroke()`. When the driver is unloaded, or the `ExitBootServices()` function is called and the malicious driver is notified, it can write the key buffer to a file or transmit it over the network to the waiting attacker.

Patching the kernel

Pre-boot malware typically interferes with the loading of the OS kernel, and patches it before it is executed, in order to modify the kernel's behaviour once it is in control. In order to do this from a malicious EFI driver we need to wait until the kernel has been loaded, as the kernel is not in memory at the point that the EFI driver is loaded and initialised. The notification for the

`ExitBootServices()` function happens to be triggered once the kernel is in memory, so it is an opportune time at which to patch the kernel image.

Before we can patch the kernel in memory we need to locate it. Inspecting the kernel Mach-O binary image informs us of its virtual memory location once it is loaded:

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

We can see that the first segment of the kernel image is loaded at the VM address `0xffffff8000200000`. If we inspect this memory location on a booted Mac OS X system using GDB we can see that the value at this address is the magic number that corresponds to a 64-bit Mach-O binary:

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

EFI uses a flat, 32-bit memory model, rather than the 64-bit memory model with canonical upper and lower halves that the OS kernel uses, so in the EFI environment the kernel image is located at `0x00200000`.

Since the kernel is not executing at this point and we do not have the facility to allocate memory in the kernel's memory map, we need a location in the kernel image in which we can store our payload. In the proof-of-concept rootkit implemented by the author, the age-old technique of storing a payload in the page-alignment padding between binary segments[5] is used.

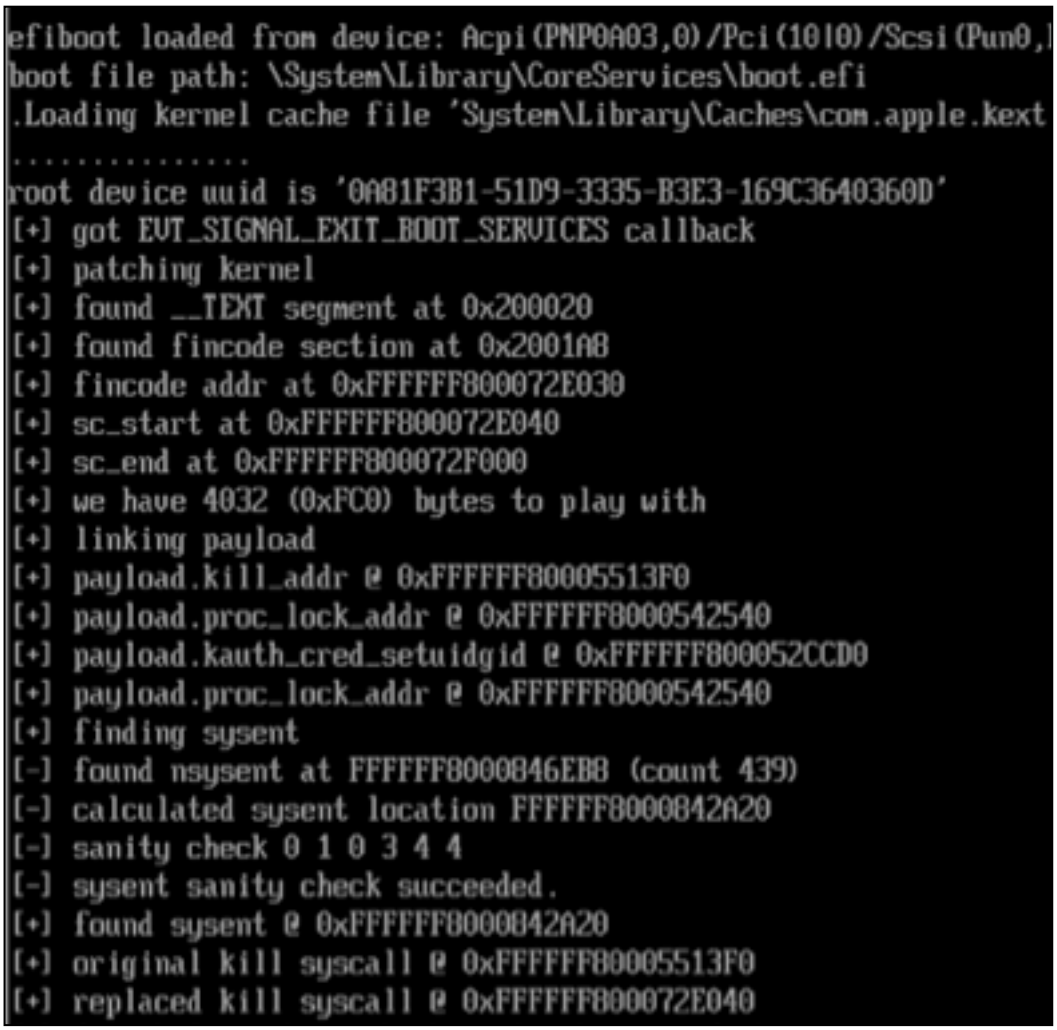
The simplest proof-of-concept implemented to demonstrate patching the kernel from EFI is a basic syscall-hooking technique, as follows:

1. Inject a binary payload into the page-alignment padding
2. Locate the sysent table within the kernel image
3. Overwrite the address in the sysent table of the `kill()` syscall with the address of the payload

The payload is called when the `kill()` syscall is called by the kernel, and performs the following operations:

1. Call the original `kill()` implementation
2. Check the parameters for a trigger value
3. If the trigger value is present, promote the calling process to uid 0

Below is an example of a driver patching the kernel at boot time to deliver this type of payload:



Proof-of-concept rootkit "Defile" patching the kernel at boot.

The main limitation of this approach is the limited space in which we can store a payload. Mach-O binary segments are aligned on page boundaries, and pages are 4096 bytes in size. This means we have an absolute maximum of 4KB in which to store our payload, but in practice we have less due to the actual kernel code encroaching on our buffer space. There are various payload storage options available to the malware developer to solve this problem; however, the author chose to store the payload in the system's NVRAM in the proof-of-concept implementation. We could also store the second stage payload in Runtime Services memory, or load it over a network connection.

Prior to patching the kernel, the EFI driver stores the second-stage payload in NVRAM using the `SetVariable()` Boot Services function. In order to access the second-stage payload in NVRAM, a small first-stage payload is injected into the page-alignment padding as discussed above. In the proof-of-concept implementation, the author chose to "hook" the execution of the kernel early in its initialisation stages in order to load the second stage payload from NVRAM before the user or the kernel has much opportunity to detect and/or interfere with the payload. This was achieved by overwriting the first instructions of the `load_init_program()` function in the XNU kernel with a jump to the first-

stage payload located in the page-alignment padding. The first-stage payload performs the following operations:

1. Save the state of the CPU
2. Locate the NVRAM device via IOKit
3. Locate the second-stage payload within NVRAM
4. Call the second-stage payload initialisation
5. Restore the patched instructions at the beginning of `load_init_program()`
6. Restore the state of the CPU
7. Jump back to the patched function, `load_init_program()`

In the proof-of-concept implementation, the second-stage payload's initialisation process performs the following tasks:

1. Allocate some memory in the kernel memory map
2. Copy the hooked `kill()` syscall payload used previously to this memory
3. Locate the sysent table
4. Overwrite the `kill()`syscall in the sysent table with a pointer to our function

This may seem like a convoluted design to deploy such a simple payload like a hooked syscall, but it demonstrates the possibilities that could be implemented for a larger, more complex rootkit payload.

Persistence & loading drivers

There are a number of locations in which EFI-based malware can be stored in order to persist on a system – the primary boot device (ie. HDD or SSD), expansion ROMs on PCI devices, and the EEPROM containing the EFI firmware itself.

Boot device

The obvious place for malware to persist is the system's boot device. EFI-based malware, unlike malware targeting the OS kernel or applications, has fairly limited options for infecting the Mac OS X boot device. The EFI specification defines a partition at the beginning of the partition table called the EFI System Partition (ESP). This partition is to be used to store drivers and bootloaders for various platforms and operating systems. Unfortunately for the malware developer, Apple's implementation does not use this partition for its intended purpose. Instead, the ESP is used to stage firmware updates (see below).

The most useful option for infecting the boot device with EFI-based malware is patching or replacing the Mac OS X bootloader – *boot.efi*. In much the same way that a replacement bootloader, such as rEFIt[6] is installed, a

malicious bootloader can be installed onto the drive and assigned as the live bootloader using the *bless* utility (or a replication of its functionality). This method has been discussed previously[7] and was not explored extensively in this research.

Furthermore, simple “evil maid” attacks can be carried out on systems not protected by an EFI firmware password by using the BDS phase’s boot menu (holding down the Option key at boot) to boot from an external USB mass storage device (such as a USB flash disk), FireWire disk, or network boot source.

PCI expansion ROMs

Attacks utilising PCI expansion (or "option") ROMs have been considered for some time now. John Heasman discussed the possibilities for option ROM-based attacks in his 2007 paper, *Implementing and Detecting a PCI Rootkit* [8], and it is the author's opinion that the threat has not diminished since then.

Modern Macs use a PCI Express (PCIe) bus to connect on-board peripherals such as the graphics card to the system. PCIe is also used to connect external peripherals to the system via the Thunderbolt expansion port. When the EFI firmware initialises the PCIe bus in the early stages of platform initialisation, it enumerates devices on the PCIe bus and executes drivers it finds in expansion ROMs connected to these devices. This operation is performed as a part of the normal initialisation of the system, as the firmware contained in the primary EFI flash chip on the logic board may not necessarily contain appropriate drivers to interact with all connected devices in the pre-boot environment. For example, when booting from a SATA adapter connected to the PCIe bus, the firmware needs to be able to interact with the SATA controller in order to read disks connected to this adapter.

In the same way that they are used to store legitimate drivers to support hardware, expansion ROMs can be used to store malicious EFI drivers. Addendums to the EFI specification provide details on how PCI option ROM images are to be structured in order to contain EFI DXE drivers. It is also possible for an option ROM image to contain both an EFI driver and a traditional BIOS driver – allowing for cross-platform payloads that can be used to attack legacy BIOS and EFI systems alike.

A number of current Mac systems utilise on-board PCIe devices that contain option ROMs. For example, some MacBook Pro systems contain an expansion ROM on the higher-performance video card. The video cards used in iMacs also contain expansion ROMs, as do some Ethernet chipsets used in various models of Macs. These are all very stealthy locations for malware to be stored, where it is unlikely to be detected.

Expansion ROMs can be written to from Mac OS X using a kernel-space driver, like *DirectHW.kext*[9]. The *flashrom*[10] utility communicates with various chipsets using the *DirectHW* driver to read and write the attached EEPROM or flash chips via the SPI protocol. Many vendors, for example Broadcom and ATI, also provide utilities for flashing the expansion ROMs on their devices.

Expansion ROMs on external devices

As described previously, PCIe devices are enumerated during the early stages of EFI initialisation, and any drivers discovered are loaded and executed. This applies to on-board devices and devices connected via PCIe bus expansions such as ExpressCard and Thunderbolt.

The author has implemented proof-of-concept "evil maid" attacks utilising the recently-released Apple Thunderbolt to Gigabit Ethernet Adapter and an ExpressCard SATA adapter as payload delivery mechanisms. To prepare the delivery mechanism for such an attack, we need to generate an option ROM image from our malicious EFI driver. This can be achieved using the *EfiRom* utility, which is part of the *BaseTools* package in EDK2. The chipset in question is a Broadcom BCM57762, the PCI vendor ID for which is 0x14E4, and the device ID is 0x1682, but the Broadcom utility seems to want 0x0001 as the PCI vendor ID and 0x8003 is the PCI device ID, so that's what we'll use:

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

We can also use the *EfiRom* utility to inspect the ROM image we've created:

```
$ EfiRom -d defile.rom
Image 1 -- Offset 0x0
  ROM header contents
    Signature              0xAA55
    PCIR offset            0x001C
    Signature               PCIR
    Vendor ID               0x0001
    Device ID               0x8003
    Length                  0x001C
    Revision                0x0003
    DeviceListOffset        0x00
    Class Code              0x000000
    Image size              0x4E00
    Code revision:          0x0000
    MaxRuntimeImageLength   0x00
    ConfigUtilityCodeHeaderOffset 0x00
    DMTFCLPEntryPointOffset 0x00
    Indicator               0x80   (last image)
    Code type               0x03   (EFI image)
  EFI ROM header contents
    EFI Signature          0x0EF1
    Compression Type       0x0000 (not compressed)
    Machine type           0x8664 (unknown)
    Subsystem              0x000B (EFI boot service driver)
    EFI image offset       0x0038 (@0x38)
```

Once we have the malicious driver in an option ROM image we can boot into a FreeDOS system with the adapter connected and use the Broadcom *B57UDIAG.EXE* utility to flash it to the expansion ROM on the Thunderbolt to Gigabit Ethernet Adapter as its PXE firmware:

```
C:\B57UDIAG\> b57udiag.exe –ppxe defile.rom
```

Once the option ROM image has been written, we can boot the machine into Mac OS X with the adapter connected and the device driver will be loaded at

boot time, and the kernel patched at the `ExitBootServices()` callback as previously described.

Initially this attack was developed using an ExpressCard SATA adapter connected to the Mac via an ExpressCard to Thunderbolt adapter, pictured below.

The first incarnation of the "evil maid" attack apparatus.

This attack was then adapted to utilise the Apple Thunderbolt to Gigabit Ethernet Adapter, resulting in a much stealthier payload delivery mechanism.

The second incarnation of the "evil maid" attack – pretty stealthy.

The output of lspci shows the presence of an expansion ROM on the 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--
```

EFI firmware flash

The ultimate goal for this type of malware is to persist within the EFI firmware itself. If this is done successfully, as it has been done in the past on some PC motherboards[11], the attacker can modify everything within the firmware volume – the core phases of EFI and all other executables contained within the firmware image.

Exploring firmware volumes

The firmware image format used on Apple Mac systems is the same format specified by Intel's documentation[12][13][14].

While investigating the EFI firmware volume format, the author developed a tool in Python to disassemble firmware volumes. This tool has not been released; however, other tools have been released for doing similar disassembly. Example output from this tool showing some of the EFI drivers contained within the MacBook's firmware image:

```
[Firmware Volume]
  Offset                        0x0 (0)
  FileSystemGuid                7a9354d9-0468-444a-81ce-0bf617d890df
  FvLength                      0x190000 (1638400)
  Signature                     '_FVH'
  Attributes                    0xffff8eff (4294938367)
  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)
  53bcc14f-c24f-434c-b294-8ed2d4cc1860 (EFI_FV_FILETYPE_DRIVER)
  ca515306-00ce-4032-874e-11b755ff6866 (EFI_FV_FILETYPE_DRIVER)
  9f455d3b-2b8a-4c06-960b-a71b9714b9cd (EFI_FV_FILETYPE_DRIVER)
  a62d933a-9293-4d9f-9a16-ce81994cc4f2 (EFI_FV_FILETYPE_DRIVER)
  1c6b2faf-d8bd-44d1-a91e-7321b4c2f3d1 (EFI_FV_FILETYPE_DRIVER)
  f1efb523-3d59-4888-bb71-eaa5a96628fa (EFI_FV_FILETYPE_DRIVER)
  a6f691ac-31c8-4444-854c-e2c1a6950f92 (EFI_FV_FILETYPE_DRIVER)
  07a9330a-f347-11d4-9a49-0090273fc14d (EFI_FV_FILETYPE_DRIVER)
  e424c009-cd92-4fec-8029-d79d3f1cf3de (EFI_FV_FILETYPE_DRIVER)
  79ca4208-bba1-4a9a-8456-e1e66a81484e (EFI_FV_FILETYPE_DRIVER)
  45424d0c-e6af-4af2-ad99-fa77168742d1 (EFI_FV_FILETYPE_DRIVER)
  378d7b65-8da9-4773-b6e4-a47826a833e1 (EFI_FV_FILETYPE_DRIVER)
  28df6de0-188f-4200-9959-46fefe971362 (EFI_FV_FILETYPE_DRIVER)
  8d460379-bf70-41c9-9a23-1808cdbbe8cd (EFI_FV_FILETYPE_DRIVER)
  6db75c4a-5e6c-4fc8-a234-f5bb27d5c2d5 (EFI_FV_FILETYPE_DRIVER)
  2daaa7f4-3167-4883-8a06-6c14f08515c7 (EFI_FV_FILETYPE_DRIVER)
  1e843ad6-e237-42fc-bda2-de78542e16dd (EFI_FV_FILETYPE_DRIVER)
  4c862fc6-0e54-4e36-8c8f-ff6f3167951f (EFI_FV_FILETYPE_DRIVER)
  cbd2e4d5-7068-4ff5-b462-9822b4ad8d60 (EFI_FV_FILETYPE_DRIVER)
--snip--
```

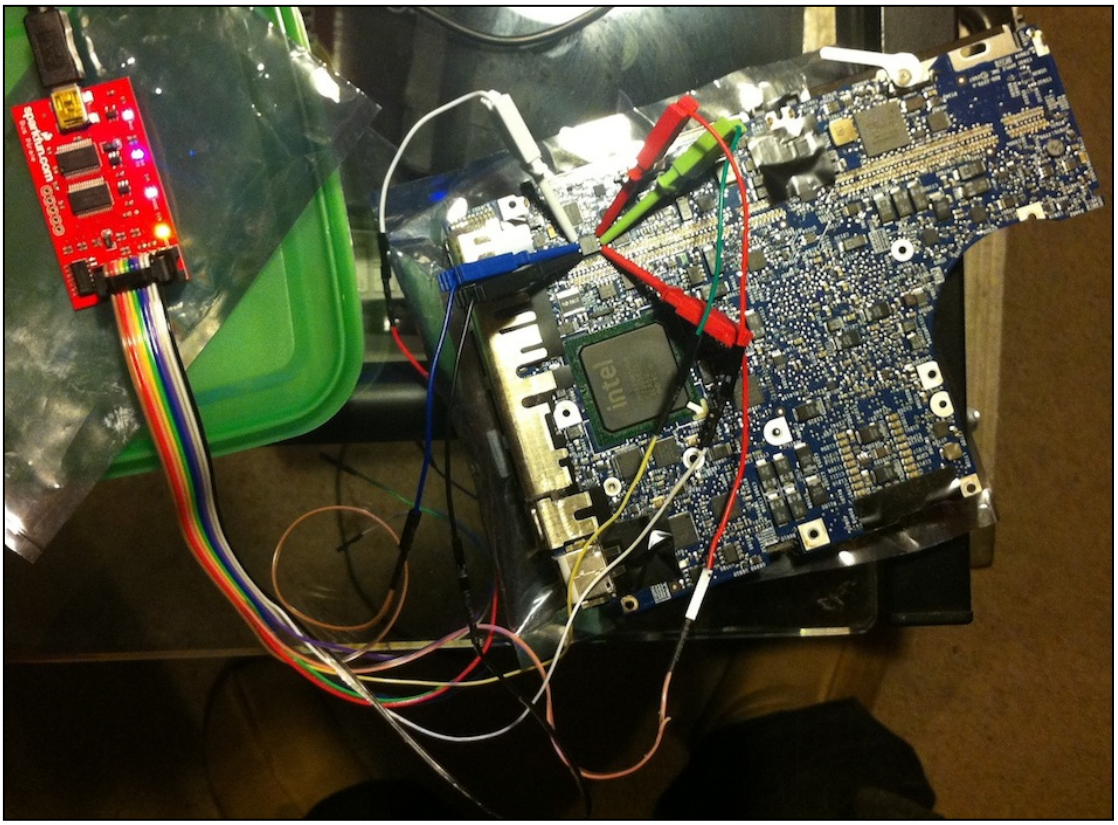
Writing to firmware flash

It is possible to communicate with the flash chip containing the EFI firmware on many Mac systems, and overwrite the EFI flash image. For example, we can communicate with the Intel ICH8M chipset on this MacBook with *flashrom*, and see the SST 25VF016B flash chip containing the EFI firmware:

```
# flashrom
flashrom v0.9.5-r1504 on Darwin 11.3.0 (x86_64), built with libpci
3.1.7, LLVM Clang 3.1 (tags/Apple/clang-318.0.54), little endian
flashrom is free software, get the source code at
http://www.flashrom.org

Calibrating delay loop... OK.
Mapping low megabyte at 0x00000400, unaligned size 0xffc00.
Mapping low megabyte, 0xffc00 bytes at unaligned 0x00000400.
sh: dmidecode: command not found
dmidecode execution unsuccessful - continuing without DMI info
Found chipset "Intel ICH8M". Enabling flash write... BBAR offset is
unknown on ICH8!
OK.
Found SST flash chip "SST25VF016B" (2048 kB, SPI) at physical
address 0xffe00000.
No operations were specified.
```

When the author overwrote the firmware flash with a new (valid) firmware image containing the original firmware and a malicious driver, the machine ceased to boot. Manual intervention was required to re-flash the original firmware by using an external flashing tool (a Bus Pirate by Dangerous Prototypes) to communicate directly with the flash chip via SPI:



Anecdotal evidence has indicated that Mac systems also contain a “boot ROM”, which is executed before the EFI firmware and verifies the integrity of the firmware image including its cryptographic signature at the end of the firmware volume. If the firmware image is not deemed to be valid, the system generates the “S.O.S.” beep sound (literally “S O S” in Morse code) and refuses to boot. The author has not explored this any further; however, it may be a future area of research.

Defense

There are a number of defensive measures that can be taken against this kind of attack – some by the user, and some by the vendor.

Firmware password

Apple has implemented password-protection on the BDS (Boot Device Selection) phase of the EFI firmware in order to prevent "evil maid" attacks where an attacker has gained physical access to a system and can interfere with the boot process. This mechanism prevents attackers from executing malicious EFI drivers and applications from devices connected to the USB, FireWire and network interfaces, but does not protect the user from malicious drivers loaded from devices connected directly to the PCIe bus via ExpressCard or Thunderbolt.

Furthermore, there have been a number of examples where the firmware password protection has been bypassed by techniques involving removing memory from the system. Newer Mac notebooks do not have removable memory, so these attacks may not be applicable to them.

Despite the attacks against the EFI firmware password protection, and the fact that this mechanism does not protect the user from drivers loaded from PCI devices, it is recommended that users apply this setting to their systems to mitigate the risk of simple “evil maid” attacks.

Secure Boot

The UEFI 2.3.1 specification defines a process for authenticating executable images to be executed by the EFI environment, known as “Secure Boot”. Approved vendors sign their drivers, bootloaders and applications with a cryptographic key. A database of allowed vendor keys is stored in secure, non-volatile storage, and these keys are used to verify the signatures within executables that are to be loaded and executed. Executables that are not signed by approved vendors are refused execution. A successful implementation of this process would mitigate the risk of many attacks described herein.

Previous generations of Mac systems have included a Trusted Platform Module (TPM) on the logic board, which was, to the knowledge of the author, unused. In an implementation of Secure Boot by Apple, the TPM would be used to store and generate cryptographic keys used in the Secure Boot process.

Attacks against, and the implementation of, Secure Boot have not been explored thoroughly in this research, however it is suggested that Apple implement Secure Boot in a future version of their EFI firmware.

Conclusion

This paper has demonstrated that there are a number of ways in which the EFI firmware used in modern Macs and other systems can be used in attacks

against the operating system and the user. These attacks can be undertaken against a wide range of hardware and software configurations with a great deal of ease, compared to similar attacks against the legacy PC BIOS, due to the standardised and cross-platform nature of the EFI specification and its supporting technologies.

References

1. Hardware Backdooring is Practical – Jonathan Brossard & Florentin Demetrescu, Hackito Ergo Sum 2012

   http://2012.hackitoergosum.org/blog/wp-content/uploads/2012/04/HES-2012-jbrossard_fdemetrescu-Hardware-Backdooring-is-pratical.pdf

2. UEFI Specification

   http://www.uefi.org/specs/

3. EFI Development Kit II

   http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2

4. Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption

   http://eprint.iacr.org/2012/374.pdf

5. Runtime Kernel kmem Patching – Silvio Cesare

   http://althing.cs.dartmouth.edu/local/vsc07.html

6. rEFIt

   http://refit.sourceforge.net/

7. Hacking the Extensible Firmware Interface – John Heasman, 2007

   https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf

8. Implementing and Detecting a PCI Rootkit – John Heasman, 2007

   http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf

9. DirectHW, part of the CoreBoot project

   http://www.coreboot.org/DirectHW

10. Flashrom

    http://flashrom.org/

11. Attacking the Intel BIOS – Invisible Things Labs, 2009

    http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

12. Intel Platform Innovation Framework for EFI – Firmware Volume Specification

http://download.intel.com/technology/framework/docs/Fv.pdf

13. Intel Platform Innovation Framework for EFI – Capsule Specification

    http://download.intel.com/technology/framework/docs/Capsule.pdf

14. Intel Platform Innovation Framework for EFI – Firmware File System Specification

    http://download.intel.com/technology/framework/docs/Ffs.pdf

About the author

Loukas is the Principal Consultant at Assurance Pty Ltd in Melbourne, Australia. Assurance is a specialist, vendor-neutral consultancy providing security and mobility services for critical infrastructure, financial services and government.