# Thunderstrike:
# EFI firmware bootkits for Apple MacBooks

Trammell Hudson and Larry Rudolph

Two Sigma Investments

trammell.hudson@twosigma.com, larry.rudolph@twosigma.com

## Abstract

There are several flaws in Apple's MacBook firmware security that allows untrusted modifications to be written to the SPI Flash boot ROM of these laptops. This capability represents a new class of persistent firmware rootkits, or 'bootkits', for the popular Apple MacBook product line. Stealthy bootkits can conceal themselves from detection and prevent software attempts to remove them. Malicious modifications to the boot ROM are able to survive re-installation of the operating system and even hard-drive replacement. Additionally, the malware can install a copy of itself onto other Thunderbolt devices' Option ROMs as a means to spread virally across air-gap security perimeters. Apple has fixed some of these flaws as part of CVE 2014-4498, but there is no easy solution to this class of vulnerability, since the MacBook lacks trusted hardware to perform cryptographic validation of the firmware at boot time.

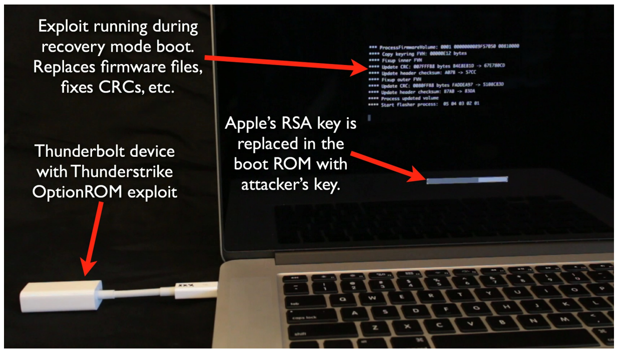
***Categories and Subject Descriptors*** D.4.6 [*Operating Systems*]: Security and Protection

***General Terms*** Invasive software, Cryptographic controls

***Keywords*** Apple, Firmware, BIOS, EFI, UEFI, rootkit, bootkit, Flash, ROM, Virus, Thunderbolt, Expansion ROM

## 1. Introduction

We have found that it is possible for untrusted code to be written to the motherboard boot ROM on Apple's MacBook laptops. Once flashed into the ROM, this new code controls the system from the very first instruction that the CPU executes upon booting. It is able to steal disk encryption passwords, install backdoors into the OS X kernel, evade detection and resist all software attempts to remove it. It is also possible for this code to infect the Option ROMs of other Thunderbolt attached devices as a viral means of crossing air-gap security measures.


Exploit running during recovery mode boot. Replaces firmware files, fixes CRCs, etc.
Apple's RSA key is replaced in the boot ROM with attacker's key.
Thunderbolt device with Thunderstrike OptionROM exploit

```
*** ProcessFirmwareVolume: 0001 000000089F57050 00810000
**** Copy keyring FVH: 00000E12 bytes
**** Fixup inner FVH
**** Update CRC: 007FFFB8 bytes 84E8E81D -> 67E780CD
**** Update header checksum: A07B -> 57CC
**** Fixup outer FVH
**** Update CRC: 0080FFB8 bytes FADDEA97 -> 5108C83D
**** Update header checksum: 87AB -> 83DA
**** Process updated volume
**** Start flasher process:  05 04 03 02 01
```

**Figure 1.** Untrusted firmware is written to the MacBook boot ROM, using Apple's own EFI flash update routine. The Thunderbolt Gigabit Ethernet adapter's Option ROM contains the Thunderstrike exploit, which circumvents Apple's EFI firmware's cryptographic signature verification of the firmware update during a recovery mode boot.

Thunderstrike, our proof-of-concept exploit shown in Figure 1, is the first publicly announced persistent boot ROM firmware rootkit, or 'bootkit', for Apple's Intel-based Macintosh systems using the Extensible Firmware Interface (EFI) (Hudson 2014). Other BIOS firmware rootkits and bootkits have focused on commodity PC hardware (Ortega and Sacco 2009; Giuliani 2011; Bulygin 2013; Kallenberg et al. 2014a; Loucaides 2014). Previously published Mac EFI bootkits were limited to writing their exploit to the EFI hard-drive partition, which allowed them to be removed by software or a re-installation of the operating system (Snare 2012).

The results of this paper can be viewed in the larger context of Option ROMS and boot time security. Although there are advantages to having a device driver built in to a new peripheral, such as a driver for a primary display device, executing external code during the boot process is dangerous. Layered security is important, but a weakness

*SYSTOR '15*, May 26–28, 2015, Haifa, Israel.
Copyright is held by the owner/author(s).
ACM ACM 978-1-4503-3607-9/15/05.
http://dx.doi.org/10.1145/10.1145/2757667.2757673

in one layer presents opportunities to attack other layers. Indeed, the execution of untrusted Option ROMs exposes hidden weakness which are normally harder to exploit.

Several security flaws in Apple's EFI firmware and boot process generate this vulnerability. The first weakness that our proof-of-concept exploits is the absence of cryptographic validation of the boot ROM at boot. Given access to the flash chip, either via a hardware in-system programmer during an 'evil-maid' attack or via software with the flash ROM chip unlocked, an attacker can install malicious code into the boot ROM that will be executed on all subsequent boots.

The second weakness is that Apple's EFI firmware loads and executes Option ROMs from external Thunderbolt devices at every boot. This flaw is still present in the current MacBook Pro models with Thunderbolt, more than two years after the issue was first reported (Snare 2012). The native x86 or x86-64 machine code in the Option ROM can inject untrusted code into the boot process that will execute in ring 0 and can circumvent many software protections. The Thunderbolt port also exposes the trusted PCIe bus to the outside world and an malicious Thunderbolt device is able to launch DMA attacks against a running system (Sevinsky 2013).

A physically proximate attacker can initiate a 'recovery mode boot', during which the Serial Peripheral Interface (SPI) flash ROM containing the boot code is unlocked and writable. The attacker can use circumvent Apple's firmware update signature verification system and leverage Apple's own update routines to replace the firmware in the boot ROM. Since the same physical port is also used for external displays and supports daisy chaining, an 'evil conference organizer' can trick the computer owner into plugging in a malicious Mini-display port video adapter that can take control of the system to install malware (FitzPatrick and Crabill 2014b,a).

It is possible for a malicious firmware bootkit to spread virally since the PCIe Option ROMs on many Thunderbolt-attached devices are also writable. Shared devices are a common way for malware to cross air-gap security perimeters, and this bootkit can spread by writing itself to Ethernet or storage adapters that are used by multiple MacBook laptops.

Lastly, fixing these problems might not be possible with software fixes and might require hardware modifications. The current systems have no trusted hardware to verify the authenticity of the firmware during boot nor to check the signatures of firmware updates. If malicious software is able to inject malware into the boot process – e.g., via buffer overflow, flash misconfigurations or other failures – it can write its untrusted code into the boot ROM.

The remainder of the paper is organized as follows. Section 2 covers related work in the area of boot-time security, Section 3 disclose the checks that Apple performs at boot time and how to perform in-system programming of the boot ROM. Section 4 discusses the details of Thunderbolt and PCIe Option ROMs. Section 5 has an overview of Apple's firmware update and signature process, and Section 6 describes the Thunderstrike proof-of-concept attack on the boot ROM. Finally, Section 7 considers some approaches to mitigate these specific problems, as well as other research into related exploits.

**Figure 2.** Photo from Edward Snowden's documents that allegedly shows an NSA Tailored Access Operation (TAO) installing a backdoor into a Cisco router. (Greenwald 2014).

## 2. Related work

There has been extensive research into boot-time security due to the risks posed by malware in the boot process. It potentially controls the system from the first instruction executed, it can interfere with the chain of trust and it can introduce a wide range of infection modes that are difficult to detect or remove. Unlike the older BIOS firmwares that were unique to each machine type, the 'Unified EFI' (UEFI) in many modern systems is built from the same Intel reference sources and provides a monoculture for attackers. Additionally, the UEFI project has very ambitious goals and consists of millions of lines of C code that are executed in ring 0. This code has significant new functionality compared to the straightforward 16-bit BIOSes of earlier systems and presents a very large attack surface for researchers.

In 2007 Heasman showed how to use internal PCI and PCIe expansion card Option ROMs as an attack vector to inject malicious code into the boot loader (Heasman 2007). This attack required physical access to the internal bus of the system or a local exploit with sufficient permission to write to the I/O space of a hardware expansion card already installed. A similar technique was used in the NSA's IRONCHEF Tailored Access Operation (TAO) that installed bootkits into the ROMs of devices installed in HP servers (Schneier 2014; Spiegel 2013) or Cisco routers (Greenwald 2014).

In 2009 Ortega and Sacco demonstrated a firmware attack that infected Linux and Windows systems, although it only worked on systems that did not use signed firmware updates or that left the flash ROM unlocked (Ortega and Sacco 2009). Since most modern systems lock the write-protect bit on the flash ROM and require signed BIOS updates,

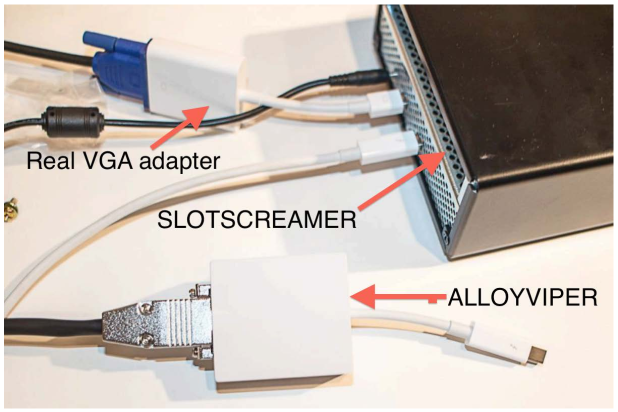


**Figure 3.** The SLOTSCREAMER and ALLOYVIPER active Thunderbolt DMA attack devices have full read/write access to system memory (FitzPatrick and Crabill 2014b,a).

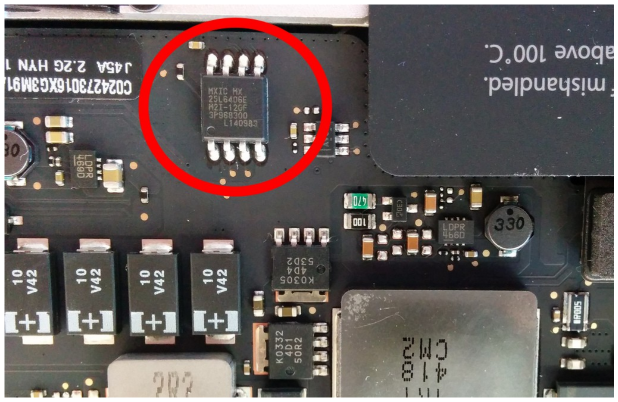


**Figure 4.** The SPI flash ROM chip is typically external from the CPU and is often a 8-SOIC. This is the MX25L6406E ROM chip on a MacBook Pro Retina 15" main logic board.

Tereshkin and Wojtczuk's attack later in 2009 circumvented firmware signature checks by exploiting a BIOS bitmap parser bug (Tereshkin and Wojtczuk 2009a). They were able to install custom firmware into certain Intel motherboards, and their bootkit used System Management Mode (SMM) to hide from antivirus software (Tereshkin and Wojtczuk 2009b). It wasn't until 2011, however, that the first reported BIOS bootkit in the wild was detected (Giuliani 2011). This bootkit was only able to attack machines using the Award BIOS and was suspected to be based on IceLord's BIOS proto-bootkit that targeted similar systems (Ice Lord 2007).

In 2009 Rutkowska demonstrated booting a system from a USB drive and installing a malicious bootloader on the internal hard-drive. Her attack was dubbed the 'evil-maid' attack since it could easily be carried out by hotel staff given a few minutes alone with a business traveler's laptop (Rutkowska 2009). Physical access attacks are also a concern while crossing international borders, since government officials have unattended access to travelers' laptops and no legal restrictions (Perlroth 2012). Rutkowska's example evil-maid bootkit installed a keylogger that stole full-disk encryption passwords when the unsuspecting user typed in the decryption key on a subsequent boot, and a more complex payload could also allow remote access to the computer or engage in other malicious activity. To prevent this particular attack, Apple and many other system vendors now require the password to be entered before allowing the system to boot from an external device if the user has configured a firmware password (Kessler 2012).

Analysis of the Stuxnet, Flame and Agent.btz worms in 2010 and 2011 showed how they were able to cross air-gaps through removable media (Nakashima 2011; Karnouskos 2011; Symantec 2012). All of these pieces of malware were believed to passively use USB flash drives and various 'zero-day attacks' against the host operating systems. The BadUSB proof of concept in 2014 demonstrated how the active firmware on the USB controllers could circumvent host security in a variety of ways, including updating the BIOS on some systems (Nohl et al. 2014).

In 2012 Snare streamlined Heasman's Option ROM attack to make it suitable for evil maids by using the externally accessible Thunderbolt port on modern Apple MacBook laptops (Snare 2012). The Apple's EFI boot ROM loads and executes the Option ROM from remote PCIe devices connected to the Thunderbolt port, even if a firmware password has been set, allowing an easy avenue to inject code into the boot process. This bug remains unfixed two years later, despite multiple Radar bugs and reports. A brief discussion of this vulnerability is in Section 4.

System security relies on a critical chain of steps in the boot process, as noted by Bulygin in 2013: The BIOS must be protected against SPI writes, BIOS updates must be signed, the signing keys must be protected, the 'boot block' code must be updated securely, the SPI flash descriptor must be programmed correctly and protected against writes, the SPI controller configuration must be locked and firmware writers must "*not introduce a single bug in all of this, of course*" (Bulygin 2013). Bugs have been found in every step of this chain on multiple motherboards (Kallenberg et al. 2014a; Bulygin et al. 2013; Kallenberg and Kovah 2015).

In 2014 FitzPatrick and Crabill demonstrated DMA vulnerabilities via the Thunderbolt port (FitzPatrick and Crabill 2014b). Their rogue hardware device, shown in Figure 3, routed all DEFCON presenters' laptops through their device and had the capability to "*read memory, bypass software and hardware security measures, and directly attack other hardware devices in the system*". Snare and Collinson showed similar capabilities at SysCon'14, including an analysis of the IOMMU/VT-d protection that Apple has started to use on more recent MacBooks (Snare and Collinson 2014).

In 2014 Loucaides presented attacks against secure boot provided an overview of the current state of BIOS and firmware security (Loucaides 2014). The presentation cov-

```
FvHeader:
 < UINT8 ZeroVector[16] >
 < EFI_GUID FileSystemGuid >
 < UINT64 FvLength >
 < UINT32 Signature >
 < EFI_FVB_ATTRIBUTES Attributes >
 < UINT16 HeaderLength >
 < UINT16 Checksum >
 < UINT8 Reserved[3] >
 < UINT8 Revision >
```

**Figure 5.** Definition of the `FvHeader` (Firmware Volume Header) structure in the EFI specification (Intel 2003b). The `Signature` field is the constant value '`_FVH`', not a cryptographic signature, and the `Checksum` field only covers the volume header, not the contents of the firmware volume.

ered issues relating to SPI flash protection mistakes, BIOS update bugs, DMA attacks against SMI and SMRAM, platform keys and many other problems. The presentation also introduced the `chipsec` toolkit to help diagnose common misconfigurations (Intel 2014).

## 3. Apple EFI boot ROM checks

We have discovered that there is no cryptographic verification of the firmware at boot time, that the Intel ME does not have any role in verification of the ROM, and nor is there any Trusted Platform Module (TPM) hardware on the motherboard to assist with these sorts of checks. Previously reported efforts to rewrite the boot ROM on Apple MacBooks had been unsuccessful, leading to conjecture that Apple's EFI firmware or the Intel Management Engine (ME) might "*verify the integrity of the firmware image including its cryptographic signature at the end of the firmware volume*" on every boot (Snare 2012). This section discusses our analysis of the additional validation that Apple does beyond the normal EFI header validation.

The MacBook Pro Retina 15" main logic board shown in Figure 4 uses a MX25L6406E 8 pin SOIC SPI flash ROM chip with 8 MB of storage to contain the EFI firmware. The BIOS region of the flash ROM consists of multiple EFI Firmware Volumes (FV) structures (Intel 2003c), the layout of which is described in Figure 5. All valid EFI firmware volumes share the same GUID in the `FileSystemGuid` member to identify the structure type, i.e. `EFI_FIRMWARE_FILE_SYSTEM_GUID`[1]. Additionally, the `Signature` field is not a cryptographic signature, but merely the constant 32-bit value `0x5f465648` ('`_FVH`'). Finally, the specification does not define any validity checks on the contents of the firmware volume, but does define the `Checksum` field as "*a 16-bit checksum of the firmware volume header.*" (Intel 2003b). This is not a cryptographic check and is only intended to protect the firmware volume header against accidental corruption.

[1] GUID `7a9354d9-4a04-ce81-ce0bf617d890df`

| Offset | Name | Description |
|---|---|---|
| 0x00 | `unknown0` | Varies, unknown purpose |
| 0x04 | `unknown1` | Always `0x0` |
| 0x08 | `crc32` | CRC32 of firmware volume data |
| 0x0c | `FreeSpace` | Byte offset to free space in volume |

**Figure 6.** Apple's modifications to the EFI Firmware volume header's 16-byte `ZeroVector` structure, reverse engineered from one version of Apple's EFI 1.10-derived boot ROM.

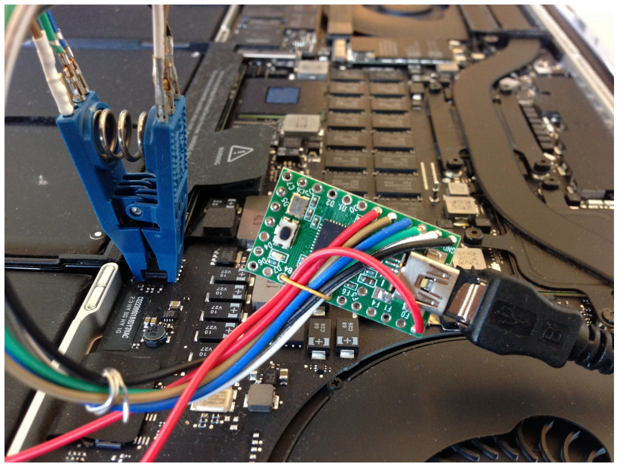


**Figure 7.** SPI Flash reader/writer on a MacBook Pro Retina SPI flash chip using an 8-SOIC chip-clip (Hudson 2012).

We have determined that Apple has repurposed the 16-byte `ZeroVector` at offset `0x0` of the firmware volume header with four of their own fields, the details of which are shown in Figure 6. The use of the first two fields is unknown; the portions of the boot ROM reverse engineered do not appear to read them. The third quad-word is a CRC32 of the contents of the firmware volumes. The boot ROM contains code that computes the CRC of each firmware volume and, if the CRC does not match, halts the boot process. Much like the 16-bit `Checksum`, this test is not a cryptographic check and only serves to verify that the ROM sections were written correctly during a firmware update and that they have not been accidentally corrupted. The offset to the free space does not appear to be used in any of the firmware functions examined.

The CRC32 is the only additional test of validity that Apple performs on the ROM image. There are no cryptographic signatures, which means that if an attacker is able to rewrite the ROM and fix up the undocumented CRC in the header, the system will boot and execute the attacker's code. One way to write code directly into the ROM is to use an in-system programmer, such as the one shown in Figure 7.

While performing an in-system programming attack by opening up the computer and directly reprogramming the ROM with a hardware device is certainly within the realm of a state-sponsored TAO or during a border crossing (Greenwald

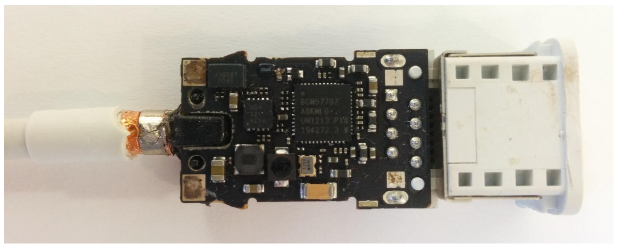
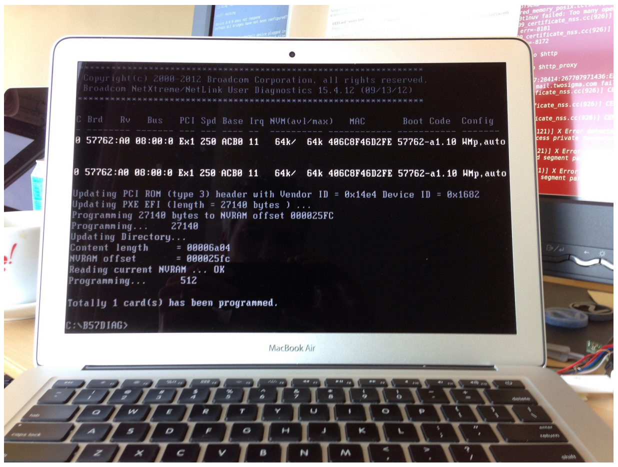


**Figure 8.** Apple's Thunderbolt Gigabit Ethernet adapter, based on a Broadcom NIC. This device contains a PCIe Option ROM with 64 KB of space and is a popular Thunderbolt Option ROM exploit device due to its low cost and ubiquity.



```
************************************************************
 Copyright(c) 2000-2012 Broadcom Corporation, all rights reserved.
 Broadcom NetXtreme/NetLink User Diagnostics 15.4.12 (09/13/12)
************************************************************

C Brd      Rv    Bus     PCI Spd Base Irq NVM(avl/max)    MAC          Boot Code    Config
- -------- ------- --- --- ---- -- ---------- ------------ ----------- ------
0 57762:A0 08:00:0 Ex1 250 ACB0 11   64k/  64k 406C8F46D2FE 57762-a1.10 WMp,auto

0 57762:A0 08:00:0 Ex1 250 ACB0 11   64k/  64k 406C8F46D2FE 57762-a1.10 WMp,auto

Updating PCI ROM (type 3) header with Vendor ID = 0x14e4 Device ID = 0x1682
Updating PXE EFI (length = 27140 bytes ) ...
Programming 27140 bytes to NVRAM offset 000025FC
Programming...    27140
Updating Directory...
Content length      = 00006a04
NVRAM offset        = 000025fc
Reading current NVRAM ... OK
Programming...       512

Totally 1 card(s) has been programmed.

C:\B57DIAG>
```

**Figure 9.** Successful flash of an exploit to the Option ROM on the Apple Thunderbolt Gigabit Ethernet adapter.

2014; Schneier 2014; Spiegel 2013; Perlroth 2012), such an involved attack might be a high bar for an evil maid.

## 4. Thunderbolt exploits

Apple's Thunderbolt connector brings the PCIe bus to an easily accessed external port, which can be used for both Option ROM (Snare 2012) and PCIe DMA attacks (FitzPatrick and Crabill 2014b; Snare and Collinson 2014). The PCIe bus is normally an internal bus and has no security of its own. Devices on the bus are trusted to write to their own memory-mapped regions and have the ability to put arbitrary source-routed packets on the bus.

Option ROMs are a legacy feature used by PCI and PCIe expansion cards to provide boot-time initialization routines to the BIOS or EFI system. For instance, a RAID controller might provide routines to read the OS kernel from the disk array, allowing the system to boot from a non-standard device. Unfortunately these ROMs contain unsigned native x86 or x86-64 machine code that will be executed in ring 0. Previous researchers have used Option ROMs as a vector to install bootkits into operating systems (Heasman 2007; Snare 2012). The UEFI 2.x specification has added driver signing and other checks related to secure booting (Wilkins and Richardson 2013; Bobzin 2011), but Apple has not upgraded their EFI implementation past 1.10 and does not enforce any signatures on device drivers or Option ROMs.

Without any verification of cryptographic signatures on the Option ROMs, a malicious attacker can flash a customized attack vector onto the PCIe device. The Thunderbolt port is ideal for evil-maid attacks, since it brings the PCIe bus to an outside connector and does not require access to the inside of the computer chassis. Ironically, the common device used in many Thunderbolt exploits is Apple's own Gigabit Ethernet adapter shown in Figure 8. This Ethernet adapter is sufficiently common to avoid suspicion and not very expensive if the attacker needs to discard it.

Since Apple has used the same connector for both Mini DisplayPort and Thunderbolt, a rogue device can appear to be an innocuous video adapter, while actually connecting the MacBook to an active attack system. One example of this is the ALLOYVIPER (FitzPatrick and Crabill 2014a), a decoy MiniDisplayPort-to-VGA adapter that connects to a PCIe card cage with a SLOTSCREAMER (FitzPatrick and Crabill 2014b) device that can read and write from anywhere in the laptop's RAM. This pair of devices was easily built with off-the-shelf components, and a motivated attacker could build a self-contained version. In addition to the active DMA attacks, this sort of system can be used to launch the firmware update exploit described in Section 6.

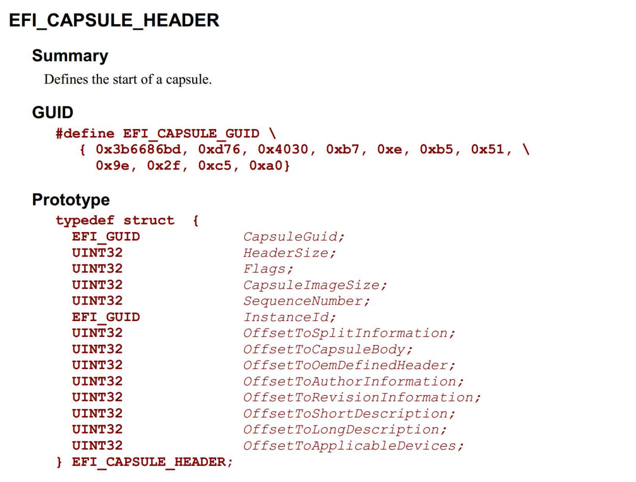
The re-programability of the Option ROMs in external devices enables a self-replicating attack that can cross air-gaps. Figure 9 shows an exploit being flashed into an Apple Gigabit Ethernet adapter from software. An infected machine can write the attack vector into the Option ROM of a clean Thunderbolt device, which can then infect further machines in turn. The limiting factor is the small amount of space in most Option ROMs to use for the attack, typically less than 64 KB. While small, this is sufficient for many payloads; MITRE's Tick malware is only 51 bytes (Butterworth et al. 2013), and many bootkits are able to reduce their payload size by making use of the extensive code and libraries already present in the EFI firmware.

## 5. Firmware update process

Firmware needs to be modifiable to fix bugs and support new hardware. However it is vital that firmware updates be signed and that the signing key be protected by the vendor to prevent malicious firmware from being installed (Loucaides 2014). Apple has implemented a public key-based firmware signature system for their firmware update files, called 'SCAP' for 'Secure Capsule' format. In this section we disclose the file format and Apple's signature system.

The SCAP file has three parts: a UEFI capsule header[2] (Figure 10), a clear-text EFI firmware volume described

[2] GUID `3b6686bd-0d76-4030-b70eb5519e2fc5a0`



**EFI_CAPSULE_HEADER**

**Summary**

Defines the start of a capsule.

**GUID**

```
#define EFI_CAPSULE_GUID \
  { 0x3b6686bd, 0xd76, 0x4030, 0xb7, 0xe, 0xb5, 0x51, \
    0x9e, 0x2f, 0xc5, 0xa0}
```

**Prototype**

```
typedef struct  {
  EFI_GUID          CapsuleGuid;
  UINT32            HeaderSize;
  UINT32            Flags;
  UINT32            CapsuleImageSize;
  UINT32            SequenceNumber;
  EFI_GUID          InstanceId;
  UINT32            OffsetToSplitInformation;
  UINT32            OffsetToCapsuleBody;
  UINT32            OffsetToOemDefinedHeader;
  UINT32            OffsetToAuthorInformation;
  UINT32            OffsetToRevisionInformation;
  UINT32            OffsetToShortDescription;
  UINT32            OffsetToLongDescription;
  UINT32            OffsetToApplicableDevices;
} EFI_CAPSULE_HEADER;
```

**Figure 10.** The UEFI capsule header structure (Intel 2003a). In a normal SCAP file from Apple, most of these fields are zero, except for the `CapsuleImageSize` and `OffsetToCapsuleBody` members.

```
typedef struct {
  EFI_GUID  HashType;
  UINT8     PublicKey[256];
  UINT8     Signature[256];
} EFI_CERT_BLOCK_RSA_2048_SHA256;
```

**Figure 11.** The `EFI_CERT_BLOCK_RSA_2048_SHA256` structure is used in Apple's SCAP files to provide cryptographic authentication of the contents of the firmware volume. The structure includes an unused RSA2048 public key in raw little-endian format and the 256 bytes of a RSA2048 signature, also in raw little-endian format, on the SHA256 cryptographic hash of the firmware volume.

in more detail below, and an `EFI_CERT_BLOCK_RSA_2048_SHA256`[3] structure at the end of the file (Figure 11). The certificate trailer consists of an RSA2048 public key, and an RSA2048 signature. Apple has also added an additional GUID[4] to the very end of the file that is checked by the SCAP validation code.

The firmware volume inside the SCAP file contains three main pieces: a Portable Executable format (PE) program that performs the write to the SPI flash, a configuration file, and multiple compressed firmware volumes that are to be written to the boot ROM. The configuration file details the regions of the SPI flash that are to be overwritten. There are three different update types used in this file: Type 3 is a simple copy, type 4 preserves some of the data already in the ROM and type 5 whose purpose is not known. The last update region in this sample configuration file shown Figure 12, `0xFFFF0000` of type 4, saves the final `0x80` bytes of the ROM that stores machine serial number and recovery hash.

```
[Head]
MinVersion = 0x0009
NumOfUpdate = 15
Update0 = 0xFFE70000
Update1 = 0xFF800000
...
[0xFF990000]
UpdateType = 3
FileGuid = 77AD7FDB-DF2A-4302-[...]
FvBaseAddress = 0xFF990000
Length = 0x001A0000
...
[0xFF801000]
UpdateType = 5
FileGuid = F1143A53-CBEB-4833-[...]
FvBaseAddress = 0xFF801000
Length = 0x0018F000
...
[0xFFFF0000]
UpdateType = 4
FileGuid = 77777777-E825-441E-[...]
FvBaseAddress = 0xFFFF0000
Length = 0x00010000
VersionFileGuid = C3E36D09-8294-4b97-[...]
SaveStart0 = 0xFFFFFF00
SaveSize0 = 0x80
```

**Figure 12.** Excerpt from the configuration file for the MacBook Pro EFI firmware `MBP101.00EE.B02` SCAP file.

```
sudo /usr/sbin/bless \
  --recovery         \
  --verbose          \
  -mount /           \
  -firmware firmware.scap
```

**Figure 13.** Command line to invoke `bless` on an SCAP file containing a firmware update using the undocumented `-firmware` option. Without an in-system programming device, it is possible to 'brick' a MacBook by running this command on an invalid file.

Apple's `/usr/sbin/bless` tool copies the SCAP file to the EFI partition and writes the filename to the protected `efi-apple-recovery-data` variable in NVRAM to initiate a firmware update. Attempts to set the NVRAM variable directly with `/sbin/nvram` fail. The syntax for the `bless` command is shown in Figure 13.

Very early during the boot process, the EFI firmware checks the NVRAM variable to see if it should initiate a recovery mode boot instead of a normal one. During a normal boot, the firmware instructs the Intel ME to lock down the flash regions and configuration before loading Option ROMs or the operating system kernel, which prevents software writes to the boot ROM. During a recovery mode boot,

[3] GUID `a7717414-c616-4977-9420844712a735bf`

[4] GUID `b0709b22-b940-45db-8a78fe8852b01a3f`

```
void
scap_validate(
    EFI_FV * fv,
    SCAP * scap,
    RSA2048 * pubkey
) {
    // Check the RSA2048 signature on the
    // SHA256 hash of the contents of the
    // firmware volume in the SCAP file.
    uint8_t hash[64] = sha256(fv);
    uint8_t new_sig[256] =
        = RSA_encrypt(pubkey, hash);
    if (new_sig != scap->sig)
        halt();

    // "Mount" the firmware volume
    if (!gDXE->ProcessFirmwareVolume(fv))
        halt();

    // Invoke the executable in the
    // firmware volume, which will
    // start the flashing process
    if (!gDXE->Dispatch())
        halt();
}
```

**Figure 14.** Pseudo-code of the cryptographic signature validation on the SCAP firmware update file performed by Apple's EFI boot ROM during a recovery mode boot.

however, the firmware leaves the flash unlocked so that the flasher program can update the ROM.

Before jumping into the flasher code, the boot firmware verifies the signature on the firmware volume in the SCAP file. Reverse engineered pseudo-code for the cryptographic verification used to validate the signature on the SCAP file is shown in Figure 14.

The boot firmware contains a keyring with five RSA public keys stored as raw bytes in little-endian format, although only the first one in the ROM is used to validate the SCAP file signature. The raw keys in the ROM do not contain the RSA exponent, but experimental tests have determined that they are using the normal `0x10001` value.

The SCAP signature check consists of computing the SHA256 checksum of the SCAP file's firmware volume data section, encrypting the hash with the first RSA public key stored in the ROM and comparing the result to the signature on the end of the file. If they do not match, the system is halted with no easy way to recover. This makes experimentation with firmware difficult since the system will continue to attempt recovery mode boots as long as the NVRAM parameters are still present, which effectively 'bricks' the MacBook until the NVRAM is cleared by flashing the ROM via an in-system programmer.

If the signature does match, the firmware then 'mounts' and 'executes' the SCAP file's firmware volume by calling the `ProcessFirmwareVolume` and `Dispatch` methods in the Device Execution Environment (DXE) Services Table (Intel 2004). The PE firmware flasher file in the SCAP file reads the configuration file in the volume and follows its instructions to flash multiple firmware regions into the still-unlocked boot ROM. After flashing, the NVRAM variables are cleared and the system reboots normally with the new boot ROM image.

We have determined that the Intel ME is not involved in validating the cryptographic signatures at all during a firmware update. This was tested by replacing the RSA and SHA256 function calls in the firmware with NOP instructions, and 'blessing' a SCAP file with an invalid signature. In other words, there is no hardware validation of the cryptographic signatures and, furthermore, if the recovery mode boot can be subverted it is possible to flash untrusted firmware to the SPI flash ROM. The next section describes one such way to take control of a recovery mode boot.

## 6. Exploiting Apple's firmware update

We have built a proof-of-concept exploit named Thunderstrike that combines Snare's 2012 Thunderbolt attack (Snare 2012) with a recovery mode boot to allow unsigned firmware to be written to the Apple boot ROM by an evil maid with only external access to the laptop. Our exploit is the first public disclosure of such a vulnerability and represents a new class of firmware 'bootkits' for the MacBook systems. Once the attack's firmware has been written to the SPI flash ROM chip, it will be executed without question, since there is no hardware verification of the boot ROM contents. The bootkit is in control of the system from the very first instruction executed upon booting, which allows it to conceal itself and resist removal attempts.

Our proof-of-concept attack 'hooks' the DXE Services Table's `ProcessFirmwareVolume` function pointer by replacing it in the table with its own function pointer. This new function can replace or modify the SCAP file's firmware volume image after it has been validated by the boot ROM software, but before it is mounted and executed. The significant security weakness here is that the PCIe Option ROMs are loaded during PCIe bus enumeration, even though the system should not require any external device drivers during a recovery mode boot.

The pseudo-code for our proof of concept is shown in Figure 15 and the full Thunderstrike exploit is shown executing in Figure 1. Unfortunately, the Option ROM on Apple's Thunderbolt Gigabit Ethernet adapter is designed to only store a small PXE boot ROM, and there is insufficient space to store an entire firmware replacement. Instead, the exploit uses only a small amount of code and a new 256-byte RSA2048 public key that is under our control. When the Option ROM's `efi_main` is invoked, it walks the system table to find the DXE Services Table and hooks the `ProcessFirmwareVolume` function pointer with our own implementation.

```
int (*old_ProcessFirmwareVolume)(EFI_FV *);

// if the firmware volume contains the RSA keyring
// file identified by GUID, then replace the
// file with our own 256-byte RSA2048 key.
int
our_ProcessFirmwareVolume
    EFI_FV * fv
) {
    EFI_FILE * file
        = find_file(fv, rsakey_guid);

    // replace the file in place and fixup the
    // various volume crc32 and header csum
    if (file) {
        memcpy(file, our_rsakey, 256);
        fixup_fvh(file);
        fixup_fvh(fv);
    }

    // use the original function to mount
    // and activate the firmware volume
    return old_ProcessFirmwareVolume(fv);
}

int
efi_main(
    EFI_SYSTEM_TABLE * system_table
) {
    gDXE = find_dxe(system_table);

    // Cache the old functon pointer
    old_ProcessFirmwareVolume
        = gDXE->ProcessFirmwareVolume;
    gDXE->ProcessFirmwareVolume
        = our_ProcessFirmwareVolume;

    return EFI_SUCCESS;
}
```

**Figure 15.** Pseudo-code of our Thunderbolt Option ROM proof-of-concept exploit that replaces Apple's keyring with our own RSA2048 public key during a firmware update recovery mode boot.

This proof-of-concept attack requires the `bless` command to be run to initiate a recovery mode boot on a valid SCAP file. It is possible to do these steps directly from a malicious Option ROM, which is a necessary development for this exploit were to be used by an actual malicious agent.

During the recovery mode boot, the SCAP validation routine will validate the SCAP file using the official Apple RSA2048 key in the boot ROM and call the DXE function pointer `ProcessFirmwareVolume`. The exploit version of this function walks the firmware volume to see if it contains a file with the GUID of the RSA keyring, and if found, copies our key into the memory image of the file inside the SCAP image. The exploit then updates the CRC32 and volume header checksum on the file volume header that contains it before repeating the two-step process on the SCAP volume header. Finally it calls the original `Process-FirmwareVolume` function pointer on the modified image. Since the public key signatures are not consulted again, the EFI firmware will mount the modified volume without complaint.

The official boot ROM code then calls `Dispatch`, which will execute the flasher in the now mounted firmware volume that contains our public key file instead of the official Apple one. The flasher will write the our key into the ROM and the exploit can then use Apple's own validation routines to install an entire firmware image signed with our key on the next boot.

Once installed into the boot ROM, this exploit is persistent and can resist attempts to remove it since it is in control of the software executed during a firmware update. The upshot: Once an attack has replaced the RSA2048 key in the ROM, it will also effectively prevent future updates from being installed unless they are signed with the attacker's private RSA key. The only way to remove or replace the infected firmware is with an in-system hardware programmer.

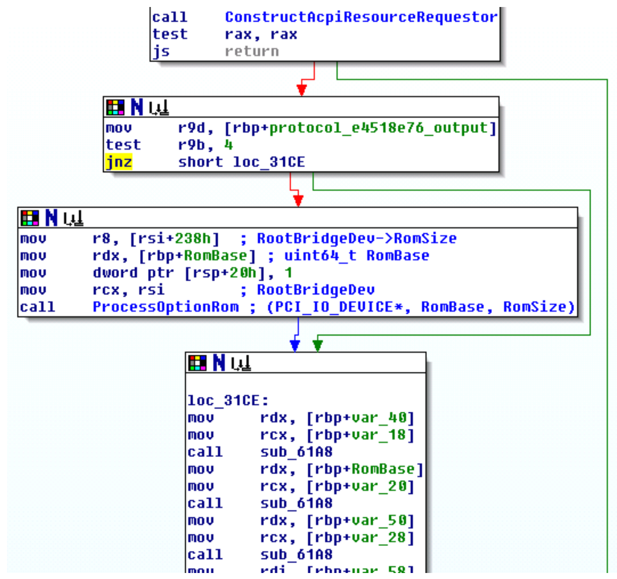
## 7. Mitigation

We disclosed this vulnerability to Apple, who assigned it CVE 2014-4498 and released a modified boot ROM firmware beginning with OS X Yosemite version 10.10.2 (Apple 2015; MITRE 2015). Their updated firmware no longer loads Option ROMs during firmware updates and will not allow downgrading to vulnerable versions of the firmware. While this prevents the specific Thunderstrike attack, it still leaves the Macbooks exposed to other boot time vulnerabilities.

One such vulnerability is that Apple's EFI firmware continues to load untrusted Option ROMs from external devices. This means that the system is vulnerable to evil-maid attacks that backdoor the OS X kernel and write a bootkit to the hard-drive similar to Snare's 2012 vulnerability (Snare 2012). The firmware also load Option ROMs from internal devices such as video cards and WiFi adapters, creating the possibility of a remotely installable exploit.

This class of vulnerabilities can be mitigated by simply not loading PCIe Option ROMs during normal boot. In many cases this does not affect the usability of the device: the modern MacBook boot ROMs include drivers for many devices, including the Gigabit Ethernet adapter used in the Thunderstrike exploit. The diassembled firmware code in Figure 16 can be patched with two bytes to change the `JNZ` instruction to a `JMP` so that the `ProcessOptionRom()` function is never invoked. The Thunderstrike proof-of-concept applies this patch to prevent its removal via a similar exploit.

A second remaining vulnerability is that the PCIe bus is still exposed on the Thunderbolt port, allowing an attacker to initiate active DMA attacks against the system using a



```
call    ConstructAcpiResourceRequestor
test    rax, rax
js      return

mov     r9d, [rbp+protocol_e4518e76_output]
test    r9b, 4
jnz     short loc_31CE

mov     r8, [rsi+238h]   ; RootBridgeDev->RomSize
mov     rdx, [rbp+RomBase] ; uint64_t RomBase
mov     dword ptr [rsp+20h], 1
mov     rcx, rsi         ; RootBridgeDev
call    ProcessOptionRom ; (PCI_IO_DEVICE*, RomBase, RomSize)

loc_31CE:
mov     rdx, [rbp+var_40]
mov     rcx, [rbp+var_18]
call    sub_61A8
mov     rdx, [rbp+RomBase]
mov     rcx, [rbp+var_20]
call    sub_61A8
mov     rdx, [rbp+var_50]
mov     rcx, [rbp+var_28]
call    sub_61A8
mov     rdi, [rbp+var_58]
```

**Figure 16.** Excerpt of disassembly showing call to `Process-OptionRom` in `PciHostBridgeResourceAllocator`. The `JNZ` instruction can be changed to a `JMP` to avoid calling `ProcessOptionRom`.

device like the SLOTSCREAMER (FitzPatrick and Crabill 2014b). Disabling Thunderbolt's PCIe support would provide better physical security, but has a significant drawback that it prevents the use of any external Thunderbolt devices. Modifying the firmware to remove the PCIe functions would be a fairly extensive reverse-engineering effort for anyone outside of Apple and is left as an area of future work. Newer models of the MacBook no longer have Thunderbolt ports and are not vulnerable to the Thunderstrike proof-of-concept, although it remains to be seen if Apple has enabled PCIe as an Alternate Mode on the USB-C port (Geuss 2014).

Software-only attacks against the flash similar to the Dark Jedi Coma (Wojtczuk and Kallenberg 2014) or against the flash update process are also possible (Kallenberg et al. 2014b; CERT 2014). Additionally, a physically proximate attacker is still able to flash a malicious ROM image through an in-system programing attack. Since there is no trusted hardware involved in the booting process it very difficult to defend against these sorts of attacks. A newer versions of the MacBook motherboard could incorporate Intel's Boot Guard hardware (Intel 2013). Initial reports indicate that it has moved the root of trust sufficiently far into the CPU to detect modifications to the boot ROM (Richardson 2015), possibly to the detriment of Free Software (Garrett 2015).

As a result of these software issues, some security guidelines go as far as suggesting physically disabling external ports like Thunderbolt and hard wiring the Write-Protect pin on the SPI flash chip (Stuge 2013). This is a radical approach and also prevents it from being used to change the NVRAM variables that are updated frequently, such as boot device, audio volume, etc. Other devices, such as the Google Chromebook, use a physical write-protect switch on the ROM and erase the device when it is toggled to prevent 'evil-maid' attacks (Google 2010).

## 8. Conclusions

We have shown that it is possible for persistent firmware modifications to be written to the SPI flash ROM chip on unpatched Apple's MacBook and MacBook Pro systems and to virally transmit these modifications across air-gaps via external Thunderbolt devices. Our proof-of-concept implementation is suitable for surreptitious 'evil-maid' attacks since it uses Apple's commonly available Gigabit Ethernet Thunderbolt adapter, does not require access to the inside of the system, and can be installed in less than a minute.

Malicious code loaded and executed from the Option ROM can circumvent Apple's firmware update process to flash unsigned code to the boot ROM. Additionally, because there is no hardware cryptographic verification of the new firmware image being written, any attacker that can take control of the system before the SPI flash is locked will be able to write its own bootkit to the ROM.

OS X Yosemite 10.10.2 and newer versions no longer load Option ROMs during firmware updates (Apple 2015; MITRE 2015), which mitigates the boot ROM persistence vulnerability, but does not remove the ability of an Option ROM attack to write a bootkit to the hard-drive during a normal boot (Snare 2012). Older OS X systems remain vulnerable to both types of attack.

Preventing malware from writing to the boot ROM is important since the firmware in the boot ROM controls the system from the first instruction the system executes upon boot up. This privileged position allows the attacker to inject exploits into the operating system kernel before it starts, circumventing any further security measures. It can also insert code into System Management Mode or use virtualization, allowing it to hide from attempts to detect it.

The code stored in the boot ROM also controls the firmware update process, and malware in the ROM can resist attempts to replace the bootkit through software. Reinstallation of the operating system or even swapping the hard-drive would be insufficient to remove this sort of firmware bootkit, since it does not depend on any software installed on the drive. Removing a firmware bootkit requires a hardware in-system programmer to rewrite the SPI flash ROM.

Finally, since the PCIe Option ROMs are writable from the EFI firmware, it is possible for the bootkit to flash a copy of itself to new Thunderbolt devices when they are plugged in. This allows the bootkit to spread to new devices and potentially cross air-gapped security measures.

## References

Apple. About the security content of OS X Yosemite v10.10.2 and Security Update 2015-001, 2015.

J. Bobzin. Implementing a Secure Boot Path with UEFI 2.3.1, 2011.

Y. Bulygin. The evil maid just got angrier. *CanSecWest*, 2013.

Y. Bulygin, A. Furtak, and O. Bzahaniuk. A Tale of One Software Bypass of Windows 8 Secure Boot. *BlackHat*, 2013.

J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. *BlackHat*, 2013.

CERT. UEFI EDK2 Capsule Update vulnerabilities, 2014.

J. FitzPatrick and M. Crabill. Stupid PCIe Tricks. *DEFCON 22*, 2014a.

J. FitzPatrick and M. Crabill. NSA Playset: PCIe SLOTSCREAMER. *BlackHat*, 2014b.

M. Garrett. Intel Boot Guard, Coreboot and user freedom, 2015.

M. Geuss. Reversible, tiny, faster: Hands-on with the USB Type-C plug, 2014.

M. Giuliani. Mebromi: the first BIOS rootkit in the wild, 2011.

Google. Cr-48 Chrome Notebook Developer Information: Entering developer mode, 2010.

G. Greenwald. *No place to hide*. Metropolitan Books, 2014.

J. Heasman. Implementing and Detecting a PCI Rootkit. *BlackHat*, 2007.

T. Hudson. spiflash source, 2012.

T. Hudson. Thunderstrike: EFI firmware rootkits for MacBooks. *31C3*, 2014.

Ice Lord. BIOS Rootkit:Welcome home,my Lord!, 2007.

Intel. `chipsec` framework, 2014.

Intel. Intel Hardware-based security technologies for intelligent retail devices, 2013.

Intel. Platform Innovation Framework for EFI: Capsule Specification, 2003a.

Intel. Platform Innovation Framework for EFI: Firmware Volume Block Specification, 2003b.

Intel. Driver Execution Environment Core Interface Specification (DXE CIS), 2004.

Intel. UEFI Specification, 2003c.

C. Kallenberg and X. Kovah. How Many Million BIOSes Would you Like to Infect? *CanSecWest*, 2015.

C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell. All your boot are belong to us. *CanSecWest*, 2014a.

C. Kallenberg, X. Kovah, J. Butterwoth, and S. Cornwell. Extreme Privilege Escalation On Windows 8/UEFI Systems. *BlackHat*, 2014b.

S. Karnouskos. Stuxnet Worm Impact on Industrial Cyber-Physical System Security. *IEEE Industrial Electronics Society*, 2011.

T. Kessler. EFI firmware proteciton locks down newer Macs, 2012.

J. Loucaides. BIOS and Secure Boot Attacks Uncovered. *RUXCON*, 2014.

MITRE. CVE-2014-4498: Thunderstrike vulnerability , 2015.

E. Nakashima. Cyber-intruder sparks response, debate. *The Washington Post*, 2011.

K. Nohl, S. Kriler, and J. Lell. BadUSB - On accessories that turn evil. *PacSecWest*, 2014.

A. Ortega and A. Sacco. Persistent BIOS Infection. *CanSecWest*, 2009.

N. Perlroth. Traveling Light in a Time of Digital Thievery. *New York Times*, 2012.

B. Richardson. The Tricky World of Securing Firmware, 2015.

J. Rutkowska. Evil Maid goes after TrueCrypt, 2009.

B. Schneier. IRONCHEF: NSA Exploit of the Day, 2014.

R. Sevinsky. Funderbolt: Adventures in Thunderbolt DMA Attacks. *BlackHat*, 2013.

Snare. De Mysteriis Dom Jobsivs – Mac EFI Rootkits. *BlackHat*, 2012.

Snare and S. Collinson. Thunderbolt and Lightning:Very very frightening. *SysCon*, 2014.

Spiegel. Inside TAO: Documents Reveal Top NSA Hacking Unit, 2013.

P. Stuge. Hardening hardware and choosing a #goodBIOS. *30C3*, 2013.

Symantec. Flamer: Highly Sophisticated and Discreet Threat Targets the Middle East, 2012.

A. Tereshkin and R. Wojtczuk. Attacking Intel BIOS. *BlackHat*, 2009a.

A. Tereshkin and R. Wojtczuk. Introducing Ring -3 Rookits. *BlackHat*, 2009b.

R. Wilkins and B. Richardson. UEFI SECURE BOOT IN MODERN COMPUTER SECURITY SOLUTIONS, 2013.

R. Wojtczuk and C. Kallenberg. Attacks on UEFI Security. *31C3*, 2014.