Thunderbolt: Exposure and Mitigation

CS838-1 - Fall 2013
Final Project Report

Saul St. John
University of Wisconsin - Madison

Abstract

The Thunderbolt interface, similarly to its quasi-predecessor, FireWire, exposes to external peripherals direct access to the system bus. What's more, Thunderbolt exacerbates said previously existing vulnerability by exposing additional platform features to connected devices. In this paper, we begin by describing and exploring these vulnerabilities. We then discuss mitigation tactics against these attacks, both well-known and novel. Subsequently, we consider countermeasures against these techniques to mitigate a Thunderbolt-borne threat. Finally, we discuss the construction of a prototype of a device that appears to behave maliciously, and show that no mitigation techniques suggested are sufficient to prevent a multi-vector Thunderbolt-born attack in the usual case.

1 Introduction

In 2011, Apple and Intel released the first commercially available implementation of their new Thunderbolt (ne Light Peak) peripheral device interface. Promoted as "the most advanced I/O ever," Thunderbolt "supports high-resolution displays and high-performance data devices through a single, compact port." It is able to accomplish this feat by virtue of its architectural construction: as a logical extension of the PCIe bus, Thunderbolt affords peripheral devices the same access to system functionality that any internal PCIe expansion card might consume.

One of the primary facilities offered by the PCIe bus architecture is the ability for devices to directly transact with system memory, bypassing the CPU, via Direct Memory Access (DMA). Through this interface, devices might compromise the integrity of a running system or exfiltrate system data. As such, the Thunderbolt interface exposes the system to infiltration by malicious individuals with physical access to the machine and in possession of a specially-constructed, purpose specific malicious device.

The contributions of this paper are two-fold: first, we demonstrate the practicality of this attack using an inexpensive, commonly available Thunderbolt adapter and widely-available software to construct a malicious device capable of exploiting a system via the Thunderbolt interface. Secondly, we demonstrate that commonly suggested mitigation tactics are ineffective in the face of this attack.

The next section describes the Thunderbolt architecture in greater detail, with a focus on the implementation that exists on Apple MacBook computers. The subsequent section discusses a prototype implementation of a malicious Thunderbolt device, and several potential future enhancements to effect persistence and stealth. Next, mitigation tactics are presented and discussed. Finally, those tactics are analyzed, and their effectiveness evaluated.

2 Background

There are three key subsystems within a modern personal computer that are responsible for the broad attack surface exposed by the Thunderbolt interface. They are described herein.

2.1 Thunderbolt Controller

Thunderbolt is, fundamentally, a hardware-agnostic serial protocol for multiplexing multiple external signals. It achieves hardware-agnosticism by means of requiring active components in all interconnect cables. It thereby abstracts itself from the need to define a physical-layer signaling protocol by requiring all interconnect devices – including simple patch cables – to provide the necessary functionality in order transmit over a given medium.

Thunderbolt itself does not define the protocols which it supports multiplexing over its own media– however, at this time, the only known use-cases for Thunderbolt is transmission of DisplayPort signals and extension of the PCIe bus. As an extension of the PCIe bus, it acts as a *transparent* bridge– to the PCIe Root Complex, a Thunderbolt-attached device appears no differently than any other PCIe device on a secondary bus. As such, a Thunderbolt connected device can invoke all system functionality ordinarily allowed to a PCIe bus-connected device, such as participation in the system boot process, peer-to-peer transactions over the PCI bus, and Direct Memory Access.

2.2 Extensible Firmware Interface

The Extensible Firmware Interface (EFI) is the logical successor to the traditional IBM PC architecture's Basic Input/Output System (BIOS). Unlike BIOS, EFI was designed from the ground-up to provide a modern runtime environment with well-defined and standardized interfaces to the system components which participate in the boot process.

2.3 I/O Memory Management Unit

An I/O Memory Management Unit (IOMMU) is a device akin to a traditional MMU such as might live in the North Bridge of a traditional Intel x86 architecture system. Rather than translating virtual addresses to physical addresses, however, it offers a CPU-programmable mechanism to facilitate the translation of *bus* addresses to physical addresses. On the Intel architecture, where supported, IOMMU functionality is built into the system Platform Controller Hub, and branded "VT-d."

In the Intel architecture, VT-d is positioned as a feature for accelerating the I/O performance of hypervisor-hosted virtual machines, as it allows guest virtual machines to directly participate in DMA with host devices– the IOMMU, in this case, is programmed so as to translate the addresses of guest-bound DMA to host-physical addresses. However, it has often been suggested for use in providing security to systems that expose interfaces allowing DMA through external connectors (such as all Intel-architecture Apple systems going back at least a decade.)

3 Exposure

The Thunderbolt interface, by virtue of its ability to couple external devices so closely to the system bus, exposes the system to a pair of attacks, described herein, each of which can result in complete system compromise.

3.1 DMA attack

Direct Memory Access is a feature of the PCI bus that was introduced to lessen the involvement of the system CPU in I/O operations over the system bus. Prior to DMA, the process for transmitting data over the bus required the CPU to chunk data to be transmitted into blocks acceptable for transmission over the bus and for receipt by the device, and then send each such chunk individually. On the reverse path, the system must periodically poll the device for fresh data, and then retrieve it similarly.

On a system supporting DMA, devices on the PCIe bus can be flagged as 'Bus Masters.' A so-flagged device is allowed to "drive" the system bus– to directly issue read and write commands to system memory, unsolicited. The intention is that the system notifies the device of the memory address where it would like data to be written to or transmitted from, and the device becomes responsible for satisfying the entirety of the request. However, a malicious device need not so-operate.

DMA attacks against Apple systems are not unique to those systems with Thunderbolt interfaces; the FireWire interface previously used was also vulnerable to such an attack. As was shown in previous work, such as Inception [1], this sort of attack can fully compromise a running system and exfiltrate arbitrary data.

3.2 Option ROM attack

In order to allow devices to participate in the system boot process, the PCI specification defines a mechanism by which a device may expose an 'Option ROM'– a memory device which may extend the functionality of the system firmware so as to initialize itself, and provide services to the platform firmware. Under BIOS systems, this sort of facility was utilized to allow add-on graphic devices to display messages from the system during initialization, and to expose network devices' ability to provide a system boot image, such as by PXE.

Under EFI, the Option ROM interface has become much more general. All devices on the PCIe bus may expose an Option ROM, and said ROM may contain arbitrary data. If such data is an EFI driver image appropriate for the execution platform, it is loaded into memory and executed during the EFI boot process's DXE (Driver eXecution Environment) phase. As an EFI driver image differs from an EFI executable image– such as an operating system bootloader– only by a typecode in the image's PE header, an EFI driver, when initialized, can perform any of the same operations an OS bootloader or a platform driver can. In other words, it executes in an unrestricted environment with full access to system components.

As a result, an Option ROM can be used as the launching-point for several types of attack.

3.2.1 SMM attack

On Intel-architecture systems, the System Management Mode (SMM) is a processor state where operating system and userland execution is suspended, and platform-specific code executes to effect platform-specific ends (such as power management, its original designed use-case.). The code executed in SMM is hidden from inspection by the operating system by virtue of being located at memory-addresses traditionally reserved for the VGA graphics aperture. As this code is "invisible" from the perspective of an initialized system, it is difficult to validate or check for malware. Code running in the SMM, however, has full access to system memory.

Prior to system initialization, though, the SMM memory window is made available for write access to all applications at a different memory address. It is therefore possible for any code that executes prior to operating system initialization to alter the SMM executable code, thereby providing a transparent mechanism for achieving runtime persistence transparent to operating system security mechanisms. Such an attack has been demonstrated previously, as in [7].

What's more, the EFI provides a standardized mechanism for runtime drivers to integrate themselves into the SMM code, and register for callbacks upon receipt of certain platform-level events, such as system management interrupts. It is therefore possible to extend the SMM functionality, rather than replacing or hot-patching it, as was required under legacy systems. We are unaware of any published EFI-specific SMM attacks at this time.

3.2.2 ACPI attack

In order to provide an interface power-management that is more descriptive and less divorced from the operating system than SMM, the Advanced Configuration and Platform Interface was developed and standardized. The ACPI allows device manufacturers to describe the construction and functionality of the platform to the operating system in the form of extensible "tables" with a standardized interface. Previous work has shown that these tables can provide a mechanism for malicious pre-boot software to integrate itself into the execution environment of a system post-boot [4]. Similar to as in the SMM attack described previously, the EFI offers a standardized interface to append, replace, and destroy tables in the ACPI to executing images, thereby substantially easing the implementation-burden of ACPI-borne malware. We are unaware of any published EFI-specific ACPI attacks at this time.

3.2.3 EFI runtime services table poisoning

The EFI provides a rich set of functionality to images executing in the pre-boot environment, called the Boot Services. Subsequent to system boot, however, those services are no longer available. Instead, the EFI exposes a much more limited set of services to an executing operating system, known as the Runtime Services Table. This table contains pointers to firmware-provided executable code, located in memory and exposing a standardized call-specification, which the operating system may call to invoke platform services. This interface allows the operating system to, for example, access platform NVRAM; however, it does not offer the rich set of functionality available to boot services, such as memory allocation (as, subsequent to boot, the responsibility for memory management belongs to the operating system, not the EFI firmware.)

However, the Runtime Services are also exposed to pre-boot applications and drivers, as is a mechanism to register for an callback function to be executed immediately prior to the system entering the booted state. A malicious application or driver might use said event as an opportunity to replace functionality within the Runtime Services table, and thereby hijack control-flow from the system when platform services are invoked. We are unaware of any published such attacks.

3.2.4 Bootloader compromise

Rather than compromise the Runtime Services table, a malicious application or driver might simply wait for the boot services terminating event, and compromise the operating system bootloader, which will have been by that point loaded into memory. This sort of attack would be much more specific to an individual operating system and version, but could potentially offer a greater likelihood of success than those described previously. We are unaware of any published such attacks.

3.3 Persistence

Finally, we note that, while these techniques can be leveraged to achieve control over a running system, they do not offer any means by which such control can be retained subsequent to device removal and system reset. We describe three such mechanisms herein.

3.3.1 Capsule Update

The EFI provides a standardized mechanism by which it may, itself, be updated to newer versions. The process, effectively, operates as follows:

1. Booted system loads update image (essentially, a filesystem containing an update payload) into memory.
2. System stores memory address of update into NVRAM.
3. System initiates reset that does not wipe physical memory.
4. EFI firmware retrieves address of update from NVRAM, mounts it, performs system-dependent authentication, and executes payload.

While the system-dependent authentication might conceivably present an impediment from abusing the Capsule Update EFI facility to achieve persistence, previous work has shown the update process, itself, to be vulnerable to exploitation. As a result, software might achieve persistence by installing itself directly into the system ROM. [5] [8]

3.3.2 PCH flash

In addition to the Capsule Update functionality, system firmware may be directly altered by software utilizing the interface provided by the system's Platform Controller Hub (on Intel-architecture systems) to flash the system ROM. Such functionality is already available in existent software packages such as the open-source "flashrom" package (part of the "coreboot" suite.) The implementation details are omitted for brevity, and the interested reader is instructed to consult the source code for "flashrom" for details.

3.3.3 Onboard Option ROMs

Some systems feature integrated devices which contain a discrete ROM external to the system firmware. This tends to be a result of the integration of previously-external peripherals into the system's core architecture. One such example is the Broadcom BCM57765 Ethernet controller embedded in the system board of Apple MacBook laptops that feature an integrated Ethernet port. As these devices operate identically to their non-integrated predecessors, a malicious software package might achieve persistence within a system by re-locating or copying its payload from the external device into an onboard, writable Option ROM.

4 Mitigation

A number of suggestions have been offered by the community to the end of protecting systems against these sorts of attack. We discuss them herein, beginning with defenses aimed at attacks against previous, similar exposures, and concluding with novel defenses and those unique to the Thunderbolt architecture.

4.1 Disable Option ROM posting

An oft-suggested defense against malicious Option ROM code is to simply fail to execute code from Option ROMs. While effective, this suggestion substantially impedes proper system behavior. For example, on a system lacking a physical Ethernet adapter, the drivers required to operate such a device might not be present in the system firmware. However, boot-time functionality, such as PXE, might require the availability of said drivers. If such a driver is only available on the device Option ROM (which is likely, as system manufacturers are unlikely to acquiesce to the inclusion of all possible device drivers in system ROM), failure to load drivers from option ROM would present a significant functional regression. As such, we reject this option as unfeasible.

4.2 Disable DMA access protocol

Under previous generation systems featuring the FireWire interface, direct memory access was exposed via a sub-protocol known as the Serial Bus Protocol-2. It has been suggested and recommended by some manufacturers that security conscious system administrators might disable this protocol so as to prevent DMA attacks, such as those implemented in Inception.

This option is inapplicable to Thunderbolt, however, as the DMA interface is not exposed through a purpose-specific sub-protocol of the connection, but is a direct feature of the bus protocol itself.

4.3 Disable Bus-Mastering on PCI devices

Similar to the Option ROM attack mitigation technique discussed earlier, it is often suggested that devices simply not be enabled for Bus Mastering– and, therefore, prevented from initiating DMA transactions. While this suggestion appears plausible, we deem it ultimately unworkable as a result of two complications.

Firstly, during the initialization of many systems, devices that can be enabled for Bus Mastering, are. This behavior has been observed on Apple laptops, and is thought to be common among modern systems. As a result, it is insufficient for an operating system to simply fail to enable bus mastering, as devices may have already been configured to enable such memory access prior to operating system execution. What's more, even were an operating system to attempt to *disable* bus mastering, there is nothing preventing a malicious device from compromising the host system in the interim period between platform enabling of bus mastering, and the operating system disabling of same. As such, this suggestion would require firmware level modifications, in addition to alteration of the design of operating system PCI drivers.

More severe, however, is the performance implication of disallowing DMA from peripheral devices. Lacking DMA, the only mechanism available to drivers to transact I/O to and from devices is polling, as discussed previously, which introduces a significant performance degradation in I/O heavy workloads. As a result, we dismiss this mitigation tactic as impractical.

4.4 Epoxy

Perhaps the most straightforward mitigation suggestion presented herein, the traditional (and, oftentimes, facetious,) suggestion for a complete defense against attacks borne by external peripherals is to physically prevent their access to the system by filling the involved connectors with epoxy, which hardens to prevent device connection. While we note that this is the only fully-effective mitigation technique we can offer, we also reject it as having an unacceptably detrimental effect on system functionality.

4.5 Secure Boot

In theory, Secure Boot would be able to prevent an Option ROM attack by requiring valid and trusted signatures on all loaded drivers, including those loaded from Option ROMs. However, the PKI management burden of actually implementing such a solution (where competitive vendors can all be authenticated, but malicious code cannot) appears nigh-on intractable. What's more, we believe this technique likely to unacceptably impinge upon end-users' freedom, and therefore dismiss it as a "cure worse than the disease."

4.6 IOMMU

In the realm of Thunderbolt-specific defenses against DMA attacks, the most often suggested defense is the employ of an IOMMU device. The effectiveness of this defense is evaluated herein.

4.6.1 Practicality

Unfortunately, IOMMUs are not available in every system; they are marketed as a value-added feature specifically targeting virtualized server workloads. As a result, they tend to be more rare among laptop systems than among desktops; in the specific case of Apple laptops, they only became available as of the refresh that implemented Haswell-architecture chipsets in Apple notebooks (in 2013.) Hence, there are at least two years of products in circulation featuring a Thunderbolt connector, but no IOMMU. What's more, we note that many vendors release devices that, in fact do feature an IOMMU, but where it is disabled in firmware or otherwise broken. As an IOMMU is not a device that can be installed in a system as an aftermarket add-on, we note that this technique cannot provide protection to the vast majority of Thunderbolt-capable devices already in-use.

However, on systems which do feature a working IOMMU, this technique does appear to provide protection against simple DMA attacks effectively. When enabled, an IOMMU device provides address translation for all devices on the bus. It is therefor required that all device drivers that induce a device to perform DMA populate the IOMMU translation buffer with mappings for the memory that device will access. As a device performing a DMA attack will not have a driver already executing in the system, the IOMMU will not have a translation entry for the addresses being written or read, so such an attack results in a page trapped by the operating system, rather than in system compromise.

4.6.2 Vulnerabilities

However, the use of an IOMMU for protection against malicious devices can be subverted by a pair of mechanisms, described herein.

Peer-to-peer transactions Within an IOMMU, the mappings maintained are each specific to a particular bus-slot. As such, a transaction to a given bus address issued by one device may not result in access to the same physical address as another device might, given the same bus address. As such, a device without any installed mappings in the IOMMU might still transact with system memory by inducing another device on the bus– one with valid mappings installed– to issue transactions on its behalf. We also note that the PCI specification explicitly allows devices to communicate with each other on a peer-to-peer basis, and that such communications are not intermediated necessarily by the IOMMU. We leave the exploration of this vulnerability to future work.

DMAR tables Many legacy system devices, including some which continue to be integrated by system manufacturers, expect to be able to read and write arbitrarily to system memory. One such example is a VGA adapter, which expects to be able to read from the VGA framebuffer in the DOS high memory area at-will. Other devices, such as the LPC bus interface, have similar expectations.

When an IOMMU device is enabled, these expectations will, by default, not be met, as there is not necessarily a driver available to inform the IOMMU of these devices' memory needs. As a result, the ACPI specification was updated to provide an interface for platform designers to inform the operating system of the memory-mapping needs of the various system bus devices, known as the Reserved Memory Region Request (RMRR). The RMRR is a subtable of the DMA Remapping table, which describe the location of the devices in the system that need particular IOMMU mappings set-up should the IOMMU be enabled. The operating system is expected to parse this table and so-populate the IOMMU translation buffer prior to enabling the IOMMU for interposition on bus address translation.

As noted previously, the EFI provides an interface for the pre-boot modification of ACPI tables, and a mechanism for the execution of code stored on device Option ROMs. It follows, therefore, that a device option ROM could, if present during the DXE load phase of the EFI boot process, install a memory-mapping request into the DMAR for a one-to-one translation of all or some subset of potential bus addresses. Such a request would result in the operating system installing mappings for the device allowing complete memory access, thus obviating the protection offered by the IOMMU.

We describe the implementation of such an Option ROM in the subsequent section, and consider such a combination Option ROM and DMA attack to effectively compromise the attack-mitigation potential offered by the use of an IOMMU.

4.7 Inverse virtual devices

Prior to the implementation of PCI passthrough within modern hypervisors, access to devices was interposed upon by an emulator, which appeared to be a physical device to the guest, but actually sanitized and validated commands being sent to the physical hardware. Such a virtual device, originally, was driven by a emulation process executing in the hypervisor's control domain; however, under the Xen architecture, "dom0 disaggregation" implanted the functionality required to execute these emulated processes in their own "stub domains."

We suggest it possible that a system might be constructed that functions somewhat as the inverse of the previously described architecture. Rather than exposing bus-connected devices to the host system, they could each be passed-through to device-specific stub domains, each with their own carefully controlled "physical" address space. The functionality implemented by such a device could be exposed to the host system through an interface similar to that used to expose devices to guests. This would provide an effective check against DMA attacks on host memory, because the "host" from the perspective of the device is an isolated virtual machine.

There are a number of significant hurdles that must be addressed before such a system could be feasible. For example, functionality would need to be implemented in order to support zero-copy I/O to the host through a guest (where it would be sanitized.) What's more, it's unclear from existing specifications how a system should react to an RMRR request for a device being passed through to a guest. We leave a full exploration of the challenges of this approach, and a more detailed analysis of it's feasibility, to future work.

5 Implementation

An Apple Thunderbolt-to-Gigabit-Ethernet adapter was obtained and utilized to construct a proof-of-concept of some of the attacks and mitigation countermeasures described in this paper. Two distinct binary images were constructed to that end.

5.1 DMA attack

Within the Thunderbolt-to-Gigabit Ethernet adapter exists a Broadcom BCM57762 network interface, which is composed of a number of sub-modules. One such sub-module is the Rx CPU, a MIPS-architecture processor that exists to implement functionality such as system management interfaces (IPMI, ASF, etc), and Ethernet, IP, or TCP segmentation and checksum offloading.

A small program, consisting of about 100 lines of MIPS assembly, was written to attempt to corrupt memory in the lowest 64k of the physical address space (which tends to be unused in modern systems.) This code was compiled using the GCC toolchain, and flashed into the device using the Broadcom-released "b57udiag" application. When the system IOMMU was enabled this device was not able to corrupt system memory; we thus deem the use of an IOMMU sufficient to prevent a simple DMA attack.

5.2 Option ROM attack

An EFI DXE runtime driver was written, consisting of about 3000 lines of C code. It was compiled with GCC, and built against the open-source TianoCore/EDK2 toolkit. The resultant EFI image was packaged as an Option ROM using using commonly available tools, and flashed onto the test device using the Linux tool "ethtool."

For demonstration only, this code hijacks control flow from the system twice, once when boot services are exited, and once when the system is about to power off, and forces the display of a notification image on the primary console for five seconds, during which time the system is otherwise unresponsive. Subsequently, control is returned to the system.

The driver also modifies the ACPI tables to request a linear mapping for itself over the lowest 16 megabytes of memory. When the IOMMU is enabled, the operating system tested (Linux 3.12) did install such a mapping into the IOMMU.

We therefore conclude that a DMA attack in concert with an Option ROM attack can successfully foil the protections offered by an IOMMU, provided the devices is connected to the system during boot. We also note that, given the close physical proximity of the power connector and the Thunderbolt connector in many systems, it is unlikely that an individual would be able to connect the device physically, but unable to induce a system reboot.

6 Conclusion

In this study, we explored a number of vulnerabilities exposed by the presence of a Thunderbolt interface within a system, including well-known attacks such as Option ROM and DMA attacks, and novel attacks, such as malicious peer-to-peer bus traffic. We further analyzed the various mitigation techniques that have been proposed, and explored their strengths and drawbacks. We then described the construction of a device implementing a representative subset of that which would be needed to construct a device-borne rootkit, and tested whether the mitigation tactics that have been suggested are sufficient to fully protect a system against malicious Thunderbolt devices. Unfortunately, complete protection does not appear to be feasible against a malicious user with unfettered physical access to the machine for a period of time long enough to induce a reboot, so we conclude by offering the following admonishment to users unwilling to epoxy their system's external ports: *practice good Thunderbolt hygeine.*

References

[1] Carsten Maartmann-Moe. Inception. `http://www.breaknenter.org/projects/inception`, 2011–2013.

[2] Carsten Maartmann-Moe. Adventures with Daisy in Thunderbolt-DMA-land: Hacking Macs through the Thunderbolt interface. `http://www.breaknenter.org/2012/02/adventures-with-daisy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface/`, 2012.

[3] Fernand Lone Sang, Vincent Nicomette and Yves Deswarte. I/O Attacks in Intel-PC Architectures and Countermeasures. `http://www.syssec-project.eu/m/page-media/23/syssec2011-s1.4-sang.pdf`, 2011.

[4] Heasman, J. Implementing and Detecting an ACPI BIOS rootkit. In *Black Hat Europe* (2006).

[5] K, L. DE MYSTERIIS DOM JOBSIVS: Mac EFI Rootkits. In *Black Hat US* (2012).

[6] Sevinsky, R. Funderbolt: Adventures in Thunderbolt DMA Attacks. In *Black Hat USA* (2013).

[7] Shawn Embleton, S. S., and Zou, C. SMM Rootkits: A New Breed of OS Independent Malware. In *SecureComm 2008* (2008).

[8] Wojtczuk, R., and Tereshkin, A. Attacking Intel BIOS. In *Black Hat USA* (2009).