Chapter 2

LEVERAGING THE USB POWER DELIVERY IMPLEMENTATION FOR DIGITAL FORENSIC ACQUISITION

Gunnar Alendal, Stefan Axelsson and Geir Olav Dyrkolbotn

Abstract Modern consumer devices present major challenges in digital forensic investigations due to security mechanisms that protect user data. The entire physical attack surface of a seized device such as a mobile phone must be considered in an effort to acquire data of forensic value.

Several USB protocols have been introduced in recent years, including Power Delivery, which enables negotiations of power delivery to or from attached devices. A key feature is that the protocol is handled by dedicated hardware that is beyond control of the device operating systems. This self-contained design is a security liability with its own attack surface and undocumented trust relationships with other peripherals and the main system-on-chips.

This chapter presents a methodology for vulnerability discovery in a black box USB Power Delivery implementation for Apple devices. The protocol and Apple-specific communications are reverse engineered, along with the firmware of the dedicated USB Power Delivery hardware. The investigation of the attack surface and potential security vulnerabilities can facilitate data acquisition in digital forensic investigations.

Keywords: Digital forensic acquisition, mobile device security, USB Power Delivery

1. Introduction

Law enforcement has special opportunities to leverage security vulnerabilities in digital forensic acquisition. Since law enforcement can physically seize devices, any exposed physical interface on the devices is a potential attack vector for data acquisition.

Exposed interfaces on mobile devices that can be leveraged without physically opening the devices include SIM card slots, SD card slots,

audio jacks and USB connectors. Interfaces that are exposed by physically opening devices are UART, JTAG and essentially any on-board peripherals that can be manipulated or replaced. The internal interfaces are often less accessible because opening a device like a mobile phone can be cumbersome and risky. A mobile phone is often glued shut and attempting to open it could disrupt normal operations, especially if the device must be powered on to exploit a specific vulnerability. This situation can occur if a phone is seized after the user has unlocked the device at least once since the device was powered on (i.e., after-first-unlock state), where user keys tied to user credentials are unlocked and more user data is potentially available. Thus, security vulnerabilities exposed via externally-accessible interfaces are preferred over internal interfaces.

The USB connector is one of the most common external interfaces in modern personal computers and embedded devices. As a result, the security of USB protocols is important from a digital forensic perspective. Wang et al. [31] have discussed USB attack strategies on the functional and physical layers, and several vulnerable scenarios for USB connected devices. However, they did not explore the security of the USB Power Delivery protocol.

Tian et al. [26] have investigated USB security. They have examined the security features provided by the USB Type-C connector, specifically, authentication that is included in recent USB Power Delivery revisions. They formally verified the authentication and identified USB attack vectors, but do not discuss implementation details. Examining actual implementations for verification of USB Power Delivery as an attack vector is, therefore, interesting and timely.

USB Power Delivery is a feature in newer devices that is available externally over standard USB physical interfaces such as a USB Type-C connector [28]. It is available on many modern personal computers and mobile phones in the after-first-unlock (AFU) and before-first-unlock (BFU) states. USB Power Delivery enables connected devices to negotiate the optimal power delivery (voltage and current), where one device acts as the source and the other as the consumer (sink).

Because devices can choose to swap the source and sink roles, the USB Power Delivery protocol supports the negotiation of the direction of power flow. Additionally, the protocol supports the negotiations of multiple devices connected to a single power source as well as re-negotiations at any time if more power is required. The protocol specification allows direct current levels up to 5 A, corresponding to a maximum of 100 W at 20 V. Thus, even the cables need to communicate using the USB Power Delivery protocol to ensure that they support

higher current levels. These cables are named "electronically marked cables" (EMCA) [29].

USB Power Delivery employs messages for communications [29]. Revision 2.0 of the protocol has control and data messages. Revision 3.0 specifies additional extended messages that support features such as firmware updates, battery information, manufacturer information and security messages. USB Power Delivery also supports a side-band channel for standard and non-standard vendor-specific communications.

Thus, the source, sink and cable can transmit and receive control, data and extended messages, as well as additional vendor-specific messages. The original and additional features of the protocol raise the question whether the protocol can be considered to be secure from a vulnerability perspective. The code implementing such a feature-rich protocol is large and complex, increasing the likelihood of faults, which include security vulnerabilities. Estimating the ratio of security vulnerabilities per line of code is difficult and cumbersome. Hatton [15] suggests a defect (bug) density of less than ten per thousand lines of code. Ozment and Schechter [19], who evaluated OpenBSD, suggest a vulnerability density three orders of magnitude less. Although the figures are not directly comparable, they indicate that more code increases the likelihood of security vulnerabilities.

The complexity of the USB Power Delivery protocol and its code base increase the likelihood that software security vulnerabilities could have been introduced during design and implementation. The protocol is also implemented in dedicated hardware, which raises questions about the state and integrity of the chip as well as the trust relationships with the rest of the system and the system-on-chip (SoC). This could expose the implementation to "evil maid" attacks [25] that replace the firmware in a USB Power Delivery chip or simply replace the entire chip.

The basis for any vulnerability research is the design and implementation details. Before applying any vulnerability discovery techniques [5, 17], access to code in any form and testing tools are extremely beneficial. Static vulnerability analysis benefits greatly from access to design and code details whereas fuzzing [24] requires simulation testing methods and tools.

Since most USB Power Delivery implementations are proprietary, the availability of source code is limited. Therefore, evaluating and estimating the likelihood of faults and security vulnerabilities based on lines of code is difficult. Additionally, since the protocol is implemented in dedicated hardware with undocumented interfaces, the ability to analyze and evaluate security via testing is limited. Few tools are available to perform black box testing or fuzzing of the protocol [1]. Extracting

firmware from a USB Power Delivery chip and analyzing the firmware are also difficult tasks. But they are important because such proprietary, non-scrutinized code may have many unknown security vulnerabilities.

It appears that USB Power Delivery has potential vulnerabilities in the protocol [23, 33] and its implementations [5]. Other vulnerabilities may arise from hidden features [8], implicit trust relationships [6] and hardware exposures such as evil maid [25]. Clearly, the exploitation of USB Power Delivery should be investigated as a means to enable the forensic acquisition of data from devices that incorporate the hardware and implement the protocol.

This chapter presents a new methodology for evaluating the potential of USB Power Delivery as an attack vector. The focus on USB Power Delivery implementations can provide insights into vulnerabilities. Binary diffing [10] of firmware versions can reveal security patches that can be exploited. Indeed, this research is important to understanding and leveraging USB Power Delivery as an entirely new attack surface for forensic data acquisition.

2. USB Power Delivery Protocol

The USB Power Delivery protocol specifications were released in 2012 as Revision 1.0 (version 1.0). The most recent specifications are provided in Revision 2.0 (version 1.3) and Revision 3.0 (version 2.0) [29]. USB Power Delivery offers a uniform method for devices to negotiate power supply configurations across vendors. The protocol is often used by devices with USB Type-C connectors. A USB Type-C connector has dedicated lines, CC1 and CC2, that enable USB Power Delivery communications between devices. However, a USB Type-C connector is not required to support USB Power Delivery; for example, Apple's proprietary Lightning Connector [12] supports USB Power Delivery. In fact, a cable with Apple Lightning and USB Type-C connectors could be used between an Apple device and any other USB Power Delivery enabled device to facilitate communications. The Apple Lightning and USB Type-C connectors are reversible, so the orientations of the connectors are not important.

The message-based USB Power Delivery protocol employs three types of messages: control, data and extended messages. Control messages are short messages that typically require no data exchange. Data messages contain data objects that are exchanged between devices. Extended messages, introduced in Revision 3.0, are data messages with larger payloads.

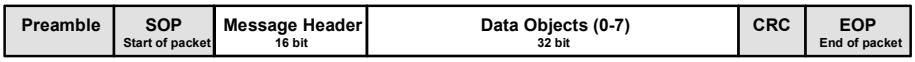

Figure 1. USB Power Delivery data message packet.

Figure 1 shows the format of a USB Power Delivery data message packet. The packet has a transport portion comprising the preamble, start of packet (SOP), cyclic redundancy check (CRC) and end of packet (EOP) fields, which encapsulates the message header and optional (up to seven) data objects. A data object has a fixed size of 32 bits, allowing for a maximum of 7×32 bits of data per message.

Table 1. Power Delivery protocol messages.

Control Messages Revs. 2.0 and 3.0	**Data Messages Revs. 2.0 and 3.0**	**Extended Messages Rev. 3.0 only**
GoodCRC	Source_Capabilities	Source_Capabilities_Extended
GotoMin	Request	Status
Accept	BIST	Get_Battery_Cap
Reject	Sink_Capabilities	Get_Battery_Status
Ping	Vendor_Defined	Battery_Capabilities
PS_RDY		Get_Manufacturer_Info
Get_Source_Cap	**Rev. 3.0 only**	Manufacturer_Info
Get_Sink_Cap	Enter_USB	Security_Request
DR_Swap	Battery_Status	Security_Response
PR_Swap	Alert	Firmware_Update_Request
VCONN_Swap	Get_Country_Info	Firmware_Update_Response
Wait		PPS_Status
Soft_Reset		Country_Info
		Sink_Capabilities_Extended
Rev. 3.0 only		Country_Codes
Data_Reset_Complete		
Not_Supported		
Get_Source_Cap_Extended		
Get_Status		
FR_Swap		
Get_PPS_Status		
Get_Country_Codes		
Get_Sink_Cap_Extended		
Data_Reset		

USB Power Delivery supports a wide range of standard messages to facilitate negotiations of power source configurations between devices. Table 1 lists the messages supported by Revision 2.0 (Version 1.3) and/or Revision 3.0 (Version 2.0). The backward compatibility means that pro-

Table 2. Structured Vendor_Defined message commands.

Structured VDM Commands
Discover Identity
Discover SVIDs
Discover Modes
Enter Mode
Exit Mode
Attention
SVID-Specific Commands

tocol complexity increases in new revisions as new messages are added but existing messages are not eliminated.

Some of the standard messages in Table 1 have sub-types. For example, a Vendor_Defined message (VDM) can be structured or unstructured. Structured VDMs have commands defined in the standard (Table 2). Unstructured VDM commands are defined by vendors and are undocumented. Vendors are free to implement proprietary communications using unstructured VDMs as demonstrated in [1]. Enabling vendors to add messages over and above the standard messages results in increased complexity and firmware code size.

To avoid conflicts when implementing proprietary vendor messages, VDMs require a standard vendor ID (SVID) defined in the specification or a vendor ID (VID) to be part of the VDM header. A VID is a unique 16-bit identifier assigned by the USB Implementers Forum [30]. A vendor with a valid VID is free to implement any VDMs needed to operate its USB Power Delivery enabled devices. Apple devices commonly use the VID `0x05ac` [13].

Connected devices negotiate power delivery via an explicit contract. Typically, this is initiated by the source device that sends a Source_Capabilities data message to which the sink replies with a GoodCRC message followed by a Request message (Figure 2). These responses inform the source that the sink is USB Power Delivery enabled and the highest protocol revision it supports. Specification revision information is included in the message header of the Request message. The highest specification revision supported by the sink corresponds to the highest specification revision supported by the source, which is indicated in the Source_Capabilities message. Thus, the connected devices know the revision and the message sets that are mutually supported.

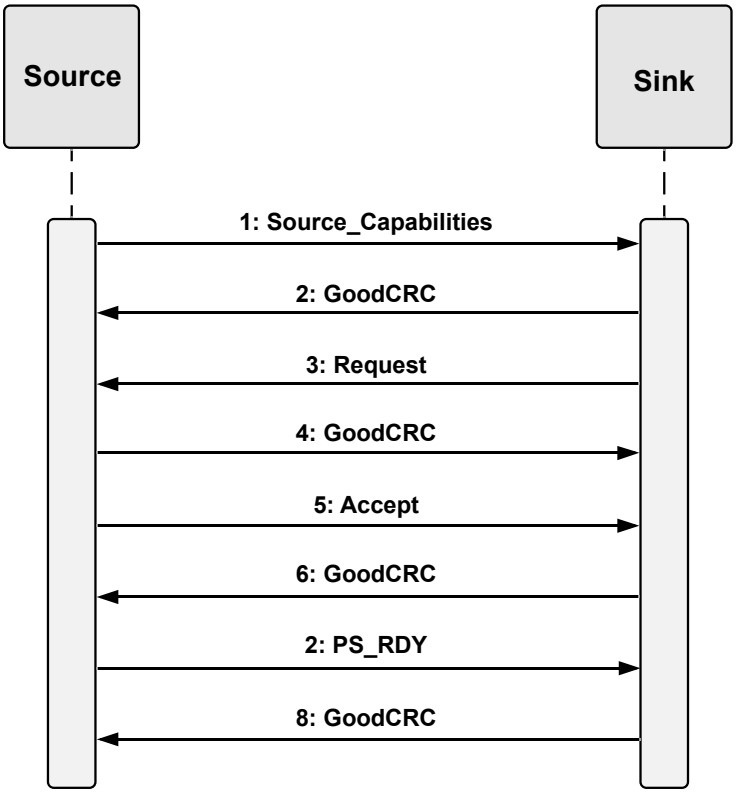

Figure 2. Generic, source-initiated explicit contract negotiation.

3. Research Methodology

The methodology for researching USB Power Delivery firmware involves information gathering, monitoring black box testing and simulation, and reverse engineering actual implementations (using binary code, documentation, source code, etc.). The individual methods often aid and overlap each other. For example, static reverse engineering of a binary is often assisted by monitoring and simulation. Even more powerful methods are instrumentation and debugging, which advance reverse engineering and vulnerability discovery.

Access to the source code of USB Power Delivery implementations (firmware) is difficult. This is because the source code is developed by the chip vendor and/or device vendor and are considered to be proprietary, business-confidential information. The documentation is also considered confidential and is kept in-house. As a result, the only option

for researchers intending to study USB Power Delivery implementations is to extract and reverse engineer firmware.

Reverse engineering firmware involves the extraction of machine code (binary) for a specific chip and applying static and dynamic methods to produce human-readable assembly code [14, 18], following which decompilation is performed to obtain pseudo high-level source code [7]. Static reverse engineering analyzes machine code without executing or interacting with the code [14]. Dynamic reverse engineering analyzes machine code by interacting with and debugging executing code (e.g., using black box testing) [14].

Static and dynamic methods and tools are available for analyzing well-known machine code structures such as PE [21] and ELF [16] binaries, but they are difficult to come by for hardware-specific and specialized firmware used by USB Power Delivery chips. Obtaining USB Power Delivery chip firmware is challenging. Vendors may include firmware in regular device updates as in the case of Apple iOS updates. However, USB Power Delivery chips may not receive any updates from vendors, requiring researchers to extract the firmware directly from the chips soldered on the targeted devices.

USB Power Delivery firmware can be retrieved from general iOS updates for Apple devices. In fact, the firmware for several USB Power Delivery chips can be obtained by unpacking and investigating the iOS updates that are often distributed in `.ipsw` archives.

Analyzing the firmware of USB Power Delivery chips is complex because the code is often based on specific, often unknown, hardware. In particular, little to nothing may be known about the architecture, memory layout and interfaces. Since the firmware does not directly interact with users, helpful, human-readable, informational/error messages are rarely embedded in the code. This renders static reverse engineering very difficult, making dynamic reverse engineering the only feasible option.

Dynamic reverse engineering involves the execution and observation of the behavior of firmware. A simulation tool can be used to evaluate code execution by communicating with the USB Power Delivery interface and, consequently, the firmware code. This can be accomplished using a proprietary USB Power Delivery simulation device as in [1]. USB Power Delivery messages are sent to the device to assist with static reverse engineering. Specifically, responses to the messages are matched in a trial and error manner to identity the corresponding firmware code. However, reversing the firmware and resulting assembly code are still very tedious. Also, the reverse engineering results may not fully match the full feature set of the original source code. Nevertheless, the results

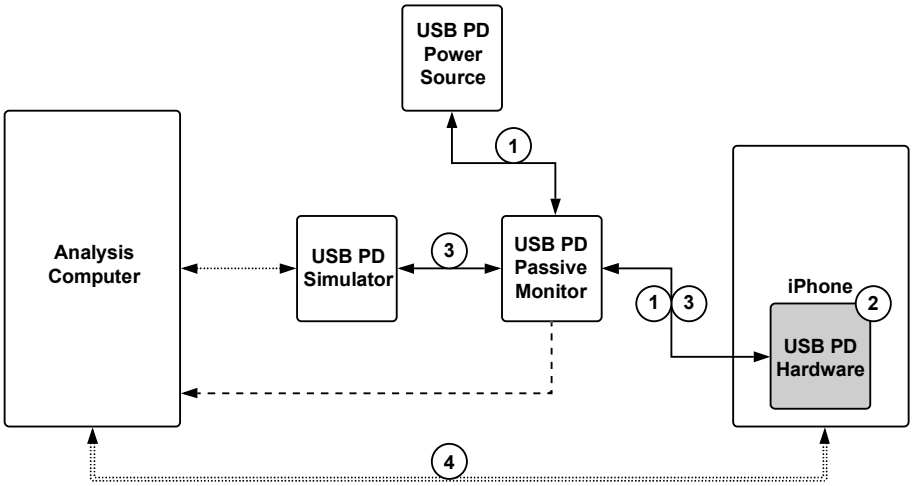

Figure 3. Experimental setup.

would help understand unknown parts of the protocol such as VDMs. The (re)produced pseudo code from the (re)produced assembly code could be used to estimate of the lines of code in the original source code and, thus, the complexity of the implementation.

The production of pseudo code from firmware binaries from different vendors is challenging and difficult to generalize. Therefore, this research opted to pursue a full reverse engineering effort for targeted devices such as Apple iPhones to help generalize the results to a wider selection of devices in the future.

Figure 2 shows the experimental setup. It incorporates an analysis computer, target iPhone, USB Power Delivery monitor, USB Power Delivery simulator and stock USB Power Delivery power sources.

The general workflow is to first conduct information gathering from open sources. A generic USB PD passive monitoring tool (1 in Figure 2) is employed to observe device functionality (especially beyond the specified behavior) when the device is connected to other Apple devices and devices from other vendors. This provides early indications of proprietary vendor code and supports subsequent reverse engineering.

Passive monitoring of USB Power Delivery communications only covers the use of a subset of the protocol and optional vendor-specific messages. Therefore, the USB Power Delivery chip firmware must be extracted from the selected Apple device (2 in Figure 2) and static reverse engineering techniques employed to disassemble the firmware and reproduce the pseudo code via decompilation.

Next, dynamic reverse engineering techniques and trial-and-error probing of the running device with actual messages are performed (3 in Figure 2). These augment the static reverse engineering efforts to provide a better understanding of the firmware. After important components of the firmware, especially undocumented vendor-specific functionality, are understood, attempts are made to implement and simulate the components to verify that the Apple device behaves and responds according to the reverse engineering results. A jailbreaking solution, `checkra1n` [20], is employed to facilitate on-device experiments. This provides root access to the test devices. Communications are performed over the normal USB interface (4 in Figure 2). Dynamic reverse engineering efforts and simulation of USB Power Delivery communications are also useful when conducting injection tests to identify vulnerabilities.

4. Results

This section presents the results of applying the research methodology discussed above to USB Power Delivery implementations in Apple iPhone models.

4.1 Information Gathering

The USB Power Delivery hardware in iPhone X, iPhone 8 and iPhone 8 Plus models appears to employ a Cypress CYPD2104 embedded microcontroller [32]. General datasheets and hardware design guides are available at the vendor site [9]. While the documentation provides useful insights, it was not possible to obtain complete documentation for the hardware, which would have provided useful information about the memory mapping of peripherals.

A Cypress CYPD2104 embedded microcontroller has a 48 MHz ARM Cortex-M0 CPU with 32 KB of flash storage for firmware, 4 KB SROM for booting and configuration, and 4 KB of SRAM. The I/O subsystem includes two serial communications blocks supporting I2C, SPI and UART, as well as several GPIOs. The interfaces are used to communicate with other peripherals such as the Apple system-on-chip. Another feature very relevant to reverse engineering and vulnerability discovery is Serial Wire Debug (SWD) access. The access could be over a JTAG interface that would enable on-device debugging capabilities, very useful for reverse engineering and vulnerability discovery. However, no attempts were made to access the interface in this research.

4.2 Passive Monitoring

USB Power Delivery is not enabled on all Apple devices. Tests on Apple iPhones suggest that it is supported by Apple iPhone 8 and later models. Using a commercial USB Power Delivery analyzer [27] on a USB Power Delivery enabled power supply connected to an iPhone reveals if the device supports USB Power Delivery. A supported device also reveals its specification revision (and thus the supported messages). An attempt by a source to negotiate an explicit contract with an unsupported device fails to elicit a response.

Table 3 shows a summary of the messages exchanged during an explicit contract between a source non-Apple power supply (Revision 3.0) and sink iPhone 10,6 (iOS 13.2.2; `iPhone10,3,iPhone10,6_13.2.2_17B102_Restore.ipsw` with SHA1 `9c50018b2ac7c2e3d667aa065aeda3a7ff80a4ef`. Table 4 shows a summary of the messages exchanged during an explicit contract between a source Apple power supply (Revision 2.0) and sink iPhone 10,6. Note that the GoodCRC messages are removed.

The non-Apple power supply supports Revision 3.0 (Index 115 in Table 3 and the test iPhone responds with Revision 2.0 (Index 127 in Table 3). This identifies the latest revision supported by the test device.

Connecting an Apple power supply reveals additional, vendor-specific communications (Table 4). Note that the communications are summarized and the GoodCRC messages are removed. The undocumented Apple device communications start with the first unstructured VDM with Index 299. This is accordance with the USB Power Delivery specifications, which state that proprietary communications should use unstructured VDMs. The communications are initiated when the Apple power supply asks the iPhone for its device ID. The iPhone device responds with an Apple VID `0x05ac`, following which the Apple power source initializes and starts the Apple-device-specific protocol. This protocol is dissected in Section 4.5.

4.3 Firmware Files

The USB Power Delivery firmware files were located in iOS updates [2]. These files are regularly released by Apple to update the iOS operating system and support on-board peripherals. The firmware files reside in an unpacked `.ipsw` file (directories: `048-90011-109/YukonB17B102.arm64CustomerRamDisk/usr/sta` or `048-90336-109/YukonB17B102.arm64CustomerRamDisk/usr/standalone/firmware/`) and are named `USB-C_HPM,x.bin`, where `x` varies based on the number of included firmware files.

The firmware files come in various versions for installation on device hardware (Section 4.1). Many of the files are equal in size and have

Table 3. Explicit contract between source non-Apple power supply (Rev. 3.0) and sink iPhone X (iOS 13.2.2).

Revision	Index	Time	Role	Message	Data
3.0	115	0:35.294.997	Source:DFP	[0]Source_Cap	A1 61 2C 91 01 0A 2C D1 02 00 F4 21 03 00 F4 C1 03 00 B1 B1 04 00 45 41 06 00 83 B5 F1 BC
	124	0:35.296.426	Sink:UFP	[0]GoodCRC	41 00 BB 6C BB A8
2.0	127	0:35.297.707	Sink:UFP	[0]Request	42 10 2C B1 04 13 3D 9D 18 5D
	131	0:35.298.419	Source:DFP	[0]GoodCRC	61 01 8F 78 38 4A
2.0	134	0:35.301.522	Source:DFP	[1]Accept	63 03 21 7B 00 96
	137	0:35.302.329	Sink:UFP	[1]GoodCRC	41 02 97 0D B5 46
2.0	140	0:35.412.687	Source:DFP	[2]PS_RDY	66 05 51 2A 14 02
	143	0:35.413.232	Sink:UFP	[2]GoodCRC	41 04 A2 A8 D6 AF

Table 4. Explicit contract between source Apple power supply (Rev. 2.0) and sink iPhone 10,6 (iOS 13.2.2).

Revision	**Index**	**Time**	**Role**	**Message**	**Data**
2.0	182	0:21.801.787	Source:DFP	[3]Source_Cap	61 17 F0 90 01 08 EA 21 1F CC
2.0	189	0:21.803.817	Sink:UFP	[0]Request	42 10 F0 C0 03 13 BC 0F E8 2B
2.0	197	0:21.805.451	Source:DFP	[4]Accept	63 09 3F 92 D5 76
2.0	203	0:21.834.345	Source:DFP	[5]PS_RDY	66 0B 56 07 AC E5
2.0	238	0:24.825.555	Source:DFP	[1]VDM:DiscIdentity	6F 13 01 80 00 FF 16 62 AB 1B
2.0	245	0:24.827.511	Sink:UFP	[2]VDM:DiscIdentity	4F 44 41 80 00 FF AC 05 00 54 00 00 00 00 00 21 7D 16 94 99 07 82
2.0	255	0:24.831.372	Source:DFP	[2]VDM:DiscSVID	6F 15 02 80 00 FF 58 38 5E 86
2.0	262	0:24.833.320	Sink:UFP	[3]VDM:DiscSVID	4F 26 42 80 00 FF 00 00 AC 05 F6 20 C2 26
2.0	270	0:24.837.322	Source:DFP	[3]VDM:DiscMode	6F 17 03 80 AC 05 BA E4 F8 1B
2.0	277	0:24.839.154	Sink:UFP	[4]VDM:DiscMode	4F 28 43 80 AC 05 02 00 00 00 72 AD 21 96
2.0	285	0:24.844.365	Source:DFP	[4]VDM:EnterMode	6F 19 04 81 AC 05 55 08 DD 38
2.0	292	0:24.846.477	Sink:UFP	[5]VDM:EnterMode	4F 1A 44 81 AC 05 8E 2F C5 E3
2.0	299	0:24.850.260	Source:DFP	[5]VDM:Unstructured	6F 1B 05 00 AC 05 E7 4D 56 1A
2.0	307	0:24.851.919	Sink:UFP	[6]VDM:Unstructured	4F 1C 15 00 AC 05 5E C3 C3 FF
2.0	315	0:24.853.357	Sink:UFP	[7]VDM:Attention	4F 3E 06 81 AC 05 02 01 AC 05 00 00 00 00 DC 69 C4 D9
2.0	324	0:24.856.927	Source:DFP	[6]VDM:Unstructured	6F 3D 02 01 AC 05 00 00 00 00 06 00 00 20 12 5E E4 81
2.0	333	0:24.858.774	Sink:UFP	[0]VDM:Unstructured	4F 10 12 00 AC 05 E6 16 E4 A7
2.0	340	0:24.860.209	Sink:UFP	[1]VDM:Attention	4F 32 06 81 AC 05 02 01 AC 05 04 00 00 00 70 47 1C CC
2.0	349	0:24.863.748	Source:DFP	[7]VDM:Unstructured	6F 3F 02 01 AC 05 04 00 00 00 00 02 08 00 D1 C4 AA B0
2.0	358	0:24.865.628	Sink:UFP	[2]VDM:Unstructured	4F 14 12 00 AC 05 26 B0 64 52

minor differences in their binaries. An important difference is that each has a different product ID (PID) that is reported by the corresponding firmware in response to a structured Discover Identity VDM (Table 2).

Table 5 lists the firmware files with their PIDs. The PID enables any connected device to identify the iPhone model via the USB Power Delivery protocol. An important observation is that the firmware files have approximately identical code across all the PIDs (and thus iPhone models) for a given iOS version. This means that any security issues discovered in firmware for a specific iPhone model would likely be present in multiple models, increasing the applicability of a forensic data acquisition method. Reverse engineering results for one device model could be reused across models, saving time and resources.

Research revealed that the different iOS 13.2.2 updates for iPhone 8, iPhone 8 Plus and iPhone X models included identical `USB-C_HPM,x.bin` files as shown in Table 5.

The common firmware codebase also supports binary diffing. Security patches discovered in two versions of a `USB-C_HPM,x.bin` file can be assumed to be present in the firmware of different iPhone models. This greatly reduces the resources needed to discover potential security patches across device models.

For a given `USB-C_HPM,x.bin` file corresponding to an iPhone 8 Plus model (`USB-C_HPM,4.bin`), different iOS updates can be downloaded and analyzed to detect changes to the USB Power Delivery firmware. Comparing the `sha1sum` values for differences is adequate to indicate a patch because there does not appear to be any iOS-specific rebuilding or versioning changes embedded in the firmware, leaving it untouched between updates unless the actual USB Power Delivery firmware is updated.

Table 6 shows several iOS updates for the iPhone 8 Plus model and the corresponding `sha1sum(USB-C_HPM,4.bin)` values. The trend is that the USB Power Delivery firmware is updated rarely. In fact, one patch was retained in iOS versions 13.3.1 to 13.4.

4.4 Firmware Reverse Engineering

Since most of the firmware files corresponding to different PIDs have few differences, reverse engineering can focus on just one of the `USB-C_HPM,x.bin` files in Table 5. The machine architecture is ARM little endian and the code is in the ARM Thumb mode [4], which is a subset of the ARM instruction set that uses variable-length instructions, often for improved code density. The code is also what is often referred to as "bare metal" code, meaning it can execute without any other abstraction layer (e.g., underlying operating system). The code

Table 5. Firmware files with their PIDs and test iPhone models.

File Name	sha1sum	PID	iPhone Model
USB-C_HPM,1.bin	83D9F3003DF9CC1915507BD090608AE0AA96CF5D	0x1654	
USB-C_HPM,2.bin	87BF3EEDA8C98081657F13B5B547924893EF0ED3	0x165d	
USB-C_HPM,3.bin	0314144681992F7A4FE8B0F69A7AB42CA159E76D	0x166c	iPhone 8
USB-C_HPM,4.bin	273A80375FE8FEC09D498221BE472958B818582F	0x167c	iPhone 8 Plus
USB-C_HPM,5.bin	ACC3DBE69E310E1FE3063725F5B436E807C83D94	0x167d	iPhone X
USB-C_HPM,6.bin	B7CE922CD8B3D0018E4861006A065C7C5FBD9D5B	0x1686	
USB-C_HPM,7.bin	916D096C8939F4E108A269A854198B95BD5A7BEE	0x1687	
USB-C_HPM,8.bin	4FC3EB5B1B0244C04F7D6BB6A917ACAAE9F7D56F	0x1688	

Table 6. USB-C_HPM,4.bin files in various iOS versions.

iOS Version	File Name	sha1sum(USB-C_HPM,4.bin)
13.4	iPhone_5.5_P3_13.4_17E255_Restore.ipsw	9767A86F62ABDC8C1046F4D807CC30DAB99A4693
13.3.1	iPhone_5.5_P3_13.3.1_17D50_Restore.ipsw	273A80375FE8FEC09D498221BE472958B818582F
13.3	iPhone_5.5_P3_13.3_17C54_Restore.ipsw	273A80375FE8FEC09D498221BE472958B818582F
13.2.3	iPhone_5.5_P3_13.2.3_17B111_Restore.ipsw	273A80375FE8FEC09D498221BE472958B818582F
13.2.2	iPhone_5.5_P3_13.2.2_17B102_Restore.ipsw	273A80375FE8FEC09D498221BE472958B818582F
13.1.3	iPhone_5.5_P3_13.1.3_17A878_Restore	273A80375FE8FEC09D498221BE472958B818582F
12.4.1	iPhone_5.5_P3_12.4.1_16G102_Restore	79CB8220D2C6F5917C1C11ED7B4BF733E3C9B1C8
12.3	iPhone_5.5_P3_12.3_16F156_Restore.ipsw	79CB8220D2C6F5917C1C11ED7B4BF733E3C9B1C8
12.2	iPhone_5.5_P3_12.2_16E227_Restore.ipsw	79CB8220D2C6F5917C1C11ED7B4BF733E3C9B1C8
12.0	iPhone_5.5_P3_12.0_16A366_Restore.ipsw	B374072044A97669A688A49E1723C55E9973A851
11.4.1	iPhone_5.5_P3_11.0_11.4.1_15G77_Restore.ipsw	1E20D8B4D54D6C092DA9B668A53AAAE81ABFA3EE
11.0	iPhone10,5_11.0_15A372_Restore.ipsw	1E20D8B4D54D6C092DA9B668A53AAAE81ABFA3EE

Table 7. `USB-C_HPM,4.bin` details for various iOS versions (see Table 6).

iOS Version	Revision	Functions	Code Bytes	OP Codes	Pseudo C Code Lines
13.4	2.0	248	18,310	8,419	6,247
13.2.2	2.0	249	18,598	8,552	6,206

directly interacts with the Apple system-on-chip and other peripherals through an I/O subsystem, mapped at specific memory addresses. Without documentation about the underlying USB Power Delivery hardware, the addresses are hardware-specific and often unknown. Therefore, from a reverse engineering perspective, it is necessary to make assumptions when code uses such unknown, hard-coded (non-position independent) addresses.

Table 7 shows the results of disassembling and decompiling the most recent versions of the firmware file `USB-C_HPM,4.bin` listed in Table 6. The total numbers of lines of pseudo C code for the two files are slightly more than 6,200. The USB Power Delivery Revision 2.0 was previously confirmed via passive monitoring. Therefore, the code supports all the messages listed for Revision 2.0 (Table 1). Code that implements additional unstructured VDMs is also included. It is expected that the number of lines of code would grow significantly to support a later revision (e.g., Revision 3.0). This is because Revision 3.0 supports a large number of additional messages (Table 1.

As described in Section 4.2, all the Apple-specific messages were identified and reverse engineered. Therefore, all the unstructured VDMs supported by the firmware could be identified in the disassembled code and pseudo C code. In fact, all the Apple-specific unstructured VDMs are handled by the same `handler` function. This function processes user input and is, therefore, an attractive target for vulnerability analysis. Erroneous handling of data in any USB Power Delivery message is a potential attack vector that could lead to a compromise of the USB Power Delivery functionality.

The number of lines of pseudo C code lines is relatively small compared with larger source code trees [19]. Since the firmware is "bare metal" code, the code is less generic and more difficult to compare with other sources. Therefore, the likelihood of security vulnerabilities in the firmware is difficult to compare with other estimates. Nevertheless, the code is in a state that is amenable to the application of established security vulnerability discovery techniques [5, 17, 24].

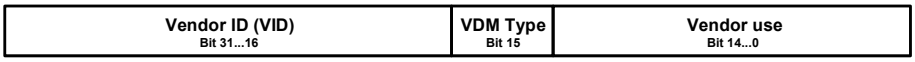

Figure 4. Unstructured VDM header.

4.5 Apple Vendor-Defined Protocol

The undocumented VDMs in Table 4 indicate that a special protocol is used by Apple devices to exchange device-specific information. Two connected Apple devices engage in an explicit contract negotiation as seen in the messages with Index 182 through 203 in Table 4. After this, the Apple-enabled power source requests the identity of the Apple iPhone X sink via a Discover Identity VDM with Index 238. Since the sink responds with a known Apple VID (`0x05ac`) and PID (`0x167d`), the two devices can engage in additional communications using messages with Index 255 through 292. The next message (Index 299) from the source to the sink is the first unstructured VDM and the first fully vendor-specific message. Upon dissecting the raw data in this message, bytes `[0:2]` were determined to correspond to the USB Power Delivery message header (Figure 1), bytes `[2:6]` to the VDM header and bytes `[6:10]` to the message CRC. Further dissection of the VDM header bytes `[6:10]` (little endian) revealed a VID of `0x05ac`, VDM Type of `0` (unstructured message) and Vendor Use of `0x5` (Figure 4). The unstructured VDM was determined to contain the expected Apple VID of `0x05ac` and an undocumented command `0x5`. For each undocumented command, a `handler` function can be identified in the associated firmware file `USB-C_HPM,4.bin` and disassembled.

Further dissection of the communications in Table 4 focused on the Attention VDM with Index 315 sent by the sink to the source (i.e., the iPhone asks the Apple power source for information). The response from the source has Index 324. This is interesting, because the iPhone only requests this type of information when it is connected to an Apple device. In fact, it turns out that the iPhone requests a range of data from the Apple power source (serial number, device name, manufacturer, etc.).

Root access to the iPhone was achieved using `checkra1n` [20]. This enabled the recovery of the data exchanged using the Apple-specific protocol. Next, the command `ioreg -f -i -l -w0 > /tmp/ioreg.txt` was used to obtain the content of the iPhone I/O Registry [3], which made it possible to interpret the exchanged data.

Table 8 shows example data exchanged between the Apple power supply source and iPhone sink. Note that the communications are summarized and the GoodCRC messages are removed. The ASCII data `C04650505D5GW85A8` at Index 401 was located in the Apple I/O Reg-

istry [3] as the Apple power device serial number. The data can be located using the command `ioreg -f -i -l -w0 | grep C04650`, which yields `"SerialNumber"="C04650505D5GW85A8" "SerialString"="C04650505D5GW85A8"`.

By leveraging the `handler` functions in the firmware, it is possible to identify all the implemented vendor protocol messages and, thus, all the supported unstructured VDMs and the messages required by the USB Power Delivery protocol. Control over all the supported messages coupled with the ability to communicate with the iPhone hardware facilitates the discovery and exploitation of security vulnerabilities. These include direct code execution on the iPhone hardware and poor input validation by peripherals/system-on-chip/kernel using the USB Power Delivery data (user input).

Table 8 shows an example of an the Apple power supply sending its serial number. Because all the messages supported by the firmware (including undocumented VDMs) can be replicated, all the data exchanged by the undocumented protocol can be modified at will.

4.6 Firmware Modification and Rollback

Analysis reveals that the `USB-C_HPM,x.bin` firmware files are unsigned and are, therefore, neither verified at installation time nor at runtime. This is verified by modifying the PID in a `USB-C_HPM,5.bin` file (see Table 5) and flashing it to the corresponding iPhone test device. With the aid of the `checkra1n` [20] jailbreaking solution, the Apple USB Power Delivery firmware flash executable `usbcfwflasher` included in the iOS firmware update file could be used to flash the modified `USB-C_HPM,5.bin` file. This can be performed on all `checkra1n`-supported Apple devices without requiring any user credentials.

The firmware modification is verified by monitoring a normal explicit contract with the additional Apple-specific VDM protocol between an Apple power supply and iPhone with the modified firmware. A successful firmware modification results in a different PID being returned from the iPhone in response to a structured Discover Identity VDM from the power supply.

Table 9 shows that the returned PID in the message with Index 83 is `0x1337` instead of the expected PID `0x167d` in the message with Index 245 in Table 4. Note that the communications are summarized and the GoodCRC messages are removed. The PIDs are the 16-bit little-endian values at bytes `[16:18]` in both messages.

The result is that it is possible to fully modify the USB Power Delivery firmware. This includes the ability to perform a firmware rollback and install an older, potentially-vulnerable, firmware version on a patched

Table 8. Data exchanged between source Apple power supply (Rev. 2.0) and sink iPhone 10,6 (iOS 13.2.2).

Revision	Index	Time	Role	Message	Data	ASCII
2.0	392	0:24.874.135	Sink:UFP	[5]VDM:Attention	4F 3A 06 81 AC 05 02 05 AC 05 30 00 00 00 F8 DF 19 1D	O:..... ..0..... ..
2.0	401	0:24.877.949	Source:DFP	[1]VDM:Unstructured	6F 73 02 05 AC 05 30 00 00 00 43 30 34 36 35 30 35 30 35 44 35 47 57 38 35 41 38 00 00 00 55 8A 48 BE	os....0. ..C04650 505D5GW8 5A8...U. H.

Table 9. Discover Identity VDMs between source Apple power supply (Rev. 2.0) and sink iPhone 10,6 (iOS 13.2.2).

Revision	Index	Time	Role	Message	Data
2.0	76	0:44.204.575	Source:DFP	[7]VDM:DiscIdentity	6F 1F 01 80 00 FF 17 8F 5B DE
2.0	83	0:44.206.298	Sink:UFP	[2]VDM:DiscIdentity	4F 44 41 80 00 FF AC 05 00 54 00 00 00 00 00 21 37 13 94 CA FB F8

device. Because this is a security vulnerability in itself, it is very useful for further vulnerability discover because researchers can implement any test code to expose, for example, further propagation in an iPhone or side-channel attack scenarios.

5. Conclusions

The methodology for analyzing USB Power Delivery implementations facilitates the discovery of security vulnerabilities for exploiting USB Power Delivery hardware to acquire data in digital forensic investigations. The ultimate goal is to further leverage privileges within the system, potentially through a new set of security vulnerabilities that are identified using the hardware as a springboard. Examples of the new vulnerabilities include implicit trust relationships, other components in the USB Power Delivery hardware and system processes that parse data provided as inputs by Apple VDM commands. The step-by-step methodology, which is demonstrated to expose the implementation details of a USB Power Delivery device, is applicable to a wide range of USB Power Delivery implementations by diverse vendors.

The results of using the methodology on Apple iPhones can be summarized as follows. Gathering information about the underlying USB Power Delivery hardware assists firmware reverse engineering, side-channel analysis and attack development. The ability to monitor USB Power Delivery messages facilitates the analysis of messages supported by a given device and helps discern if a proprietary vendor protocol is employed. Sending and receiving arbitrary messages using a simulation tool advances black box testing, reverse engineering and exploitation.

Additionally, the reverse engineering of firmware to yield disassembled code and pseudo C code is very useful for manual and automated vulnerability analyses. Diffing tests can help reveal patches that can be checked for potential security vulnerabilities. Rollbacks of vulnerable firmware can be accomplished on jailbroken devices without requiring user credentials because firmware signatures and rollback protection mechanisms are not implemented. The lack of signatures also facilitates arbitrary modifications of firmware that expose USB Power Delivery to evil maid attacks.

Future research will attempt to discover additional vulnerabilities. It will also attempt to simulate and instrument/debug the extracted firmware, with the goal of advancing fuzzing techniques for vulnerability discovery. Other avenues of future research include debugging test devices and chips via JTAG and conducting simulation via emulation and symbolic execution [11, 22, 24].

Acknowledgements

This research was supported by the IKTPLUSS Program of the Norwegian Research Council under R&D Project Ars Forensica Grant Agreement 248094/O70. Apple was notified about this research in advance of publication.

References

[1] G. Alendal, S. Axelsson and G. Dyrkolbotn, Exploiting vendor-defined messages in the USB Power Delivery protocol, in *Advances in Digital Forensics XV*, G. Peterson and S. Shenoi (Eds.), Springer, Cham, Switzerland, pp. 101–118, 2019.

[2] Apple, About iOS 13 Updates, Cupertino, California (`support.apple.com/en-us/HT210393`), 2021.

[3] Apple, The I/O Registry, Cupertino, California (`developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/TheRegistry/TheRegistry.html`), 2021.

[4] ARM, The Thumb Instruction Set, ARM7TDMI Technical Reference Manual, Revision r4p1, Cambridge, United Kingdom (`infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html`), 2004.

[5] A. Austin and L. Williams, One technique is not enough: A comparison of vulnerability discovery techniques, *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pp. 97–106, 2011.

[6] G. Beniamini, Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2), Project Zero Team, Google, Mountain View, California (`googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html`), 2017.

[7] G. Chen, Z. Qi, S. Huang, K. Ni, Y. Zheng, W. Binder and H. Guan, A refined decompiler to generate C code with high readability, *Software: Practice and Experience*, vol. 43(11), pp. 1337–1358, 2013.

[8] W. Chen and J. Bhadra, Striking a balance between SoC security and debug requirements, *Proceedings of the Twenty-Ninth IEEE International System-on-Chip Conference*, pp. 368–373, 2016.

[9] Cypress Semiconductor, CYPD2104-20FNXIT, San Jose, California (`www.cypress.com/part/cypd2104-20fnxit`), 2018.

[10] Y. Duan, X. Li, J. Wang and H. Yin, DeepBinDiff: Learning program-wide code representations for binary diffing, *Proceedings of the Twenty-Seventh Annual Network and Distributed System Security Symposium*, 2020.

[11] D. Engler and D. Dunbar, Under-constrained execution: Making automatic code destruction easy and scalable, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–4, 2007.

[12] A. Golko, E. Jol, M. Schmidt and J. Terlizzi, Dual Orientation Connector with External Contacts and Conductive Frame, U.S. Patent No. 0115821 A1, May 9, 2013.

[13] S. Gowdy, The USB ID Repository (`www.linux-usb.org/usb-ids.html`) 2021.

[14] A. Harper, D. Regolado, R. Linn, S. Sims, B. Spasojevik, L. Martinez, M. Baucom, C. Eagle and S. Harris, *Gray Hat Hacking: The Ethical Hacker's Handbook*, McGraw-Hill Education, New York, 2018.

[15] L. Hatton, Re-examining the fault density component size connection, *IEEE Software*, vol. 14(2), pp. 89–97, 1997.

[16] Y. Li and J. Yan, ELF-based computer virus prevention technologies, *Proceedings of the Second International Conference on Information Computing and Applications*, pp. 621–628, 2011.

[17] B. Liu, L. Shi, Z. Cai and M. Li, Software vulnerability discovery techniques: A survey, *Proceedings of the Fourth International Conference on Multimedia Information Networking and Security*, pp. 152–156, 2012.

[18] A. Maurushat, *Disclosure of Security Vulnerabilities: Legal and Ethical Issues*, Springer, London, United Kingdom, 2013.

[19] A. Ozment and S. Schechter, Milk or wine: Does software security improve with age? *Proceedings of the Fifteenth USENIX Security Symposium*, 2006.

[20] A. Panhuyzen, `checkra1n` (`checkra.in`), 2021.

[21] M. Pietrek, Peering Inside the PE: A Tour of the Win32 Portable Executable File Format (`bytepointer.com/resources/pietrek_peering_inside_pe.htm`), 1994.

[22] E. Schwartz, T. Avgerinos and D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 317–331, 2010.

[23] A. Sosnovich, O. Grumberg and G. Nakibly, Finding security vulnerabilities in a network protocol using parameterized systems, *Proceedings of the Twenty-Fifth International Conference on Computer Aided Verification*, pp. 724–739, 2013.

[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel and G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, *Proceedings of the Twenty-Third Annual Network and Distributed System Security Symposium*, 2016.

[25] A. Tereshkin, Evil maid goes after PGP whole disk encryption, keynote lecture presented at the *Third International Conference on Security of Information and Networks*, 2010.

[26] J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates and K. Butler, SoK: "Plug & pray" today – Understanding USB insecurity in versions 1 through C, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 1032–1047, 2018.

[27] Total Phase, USB Power Delivery Analyzer, Sunnyvale, California (`www.totalphase.com/products/usb-power-delivery-analyzer`), 2021.

[28] USB Implementers Forum, Universal Serial Bus Type-C Cable and Connector Specification, Release 2.0, Beaverton, Oregon (`www.usb.org/sites/default/files/USBType-CSpecR2.0-August2019.pdf`), 2019.

[29] USB Implementers Forum, USB Power Delivery, Beaverton, Oregon (`www.usb.org/document-library/usb-power-delivery`), 2019.

[30] USB Implementers Forum, Getting a Vendor ID, Beaverton, Oregon (`www.usb.org/getting-vendor-id`), 2020.

[31] Z. Wang and A. Stavrou, Exploiting smart-phone USB connectivity for fun and profit, *Proceedings of the Twenty-Sixth Annual Computer Security Applications Conference*, pp. 357–366, 2010.

[32] D. Yang, S. Wegner and J. Morrison, Apple iPhone X Teardown, TechInsights, Ottawa, Canada (`www.techinsights.com/blog/apple-iphone-x-teardown`), 2017.

[33] F. Yang and S. Manoharan, A security analysis of the OAuth protocol, *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 271–276, 2013.