MIN()imum Failure: EMFI Attacks against USB Stacks

Colin O'Flynn
Dalhousie University

Abstract

Electromagnetic Fault Injection (EMFI) allows generation of faults in a target device without needing to physically modify the target. This paper uses EMFI to recover secret data from two devices without opening the enclosure of the devices, making the attack possible without leaving any physical evidence. This is demonstrated on two devices: a Trezor bitcoin wallet and a Solo Key open-source FIDO2 authentication key.

The specific vulnerable code attacked with EMFI is part of the USB stack. The attack allows a host-provided value of `wLength` to be used in reading back up to 64 Kbyte of memory from the target device. Examples of this vulnerability are given for three popular general-purpose RTOSes.

To assist with evaluation of this attack, the open-source PhyWhisperer-USB hardware is also introduced. This tool provides hardware USB decoding and pattern matching to allow cycle-accurate fault injection timing.

1 Introduction

The USB stack is relatively complicated, and embedded devices with USB have had a number of vulnerabilities found due to incorrect parsing of USB messages. For this reason, USB fuzzing tools have been created to attempt to find useful bugs an attacker could target [7, 14]. This work was accelerated by Travis Goodspeed with the introduction of Facedancer, an open-source hardware tool designed for low-level USB fuzzing [5, 6].

Moving beyond incorrect USB stack implementation bugs, inserting glitches or faults into a device during USB parsing was first demonstrated by Micah Scott [12]. Scott used a voltage fault injection attack to dump firmware from a device, and her seminal work is built on here to demonstrate how other data can be recovered via fault attacks during USB operations, and without requiring modifications to the target.

Reading data from a target without physically modifying it is an important threat model for any USB portable security token or secure data store. Typically the portable devices need to resist an attack for the length of time it would take before they are noticed missing, as once the device is reported missing the owner can revoke any credentials stored in the device. An attack which is successful before the device is noticed missing, but destroys the device in the process, leaves an obvious clue which could lead back to the attacker.

Examples of devices that we consider a portable security token or secure data store can take many forms. This would include smart cards (encompassing access control, credit cards, and satellite tv cards) [8], encrypted USB storage, automotive keyfobs, FIDO authentication tokens, bitcoin wallets, and license dongles for software. All of these devices store an important secret inside the device, which an attacker has financial incentive to recover. Some of these devices have been tested or certified to a level of attack resistance – most smart cards for example have a common criteria rating [4], and FIDO has recently introduced 'Authenticator Certification Levels' to indicate how resistant a given authentication token is against advanced attacks (including chip-level with captured device assumptions).

The 'holy grail' of attacking any of those portable security tokens or secure data stores would be to attack it without being detected. This means to recover the sensitive data on the device without leaving a sign of physical tampering, and with the attack working in a 'reasonable' time-frame. What defines a reasonable time-frame depends on the specific device and threat model, but for most portable devices previously listed we can imagine a 0.5 – 2 hour window being reasonable.

The specific devices being investigated in this paper are USB-connected tokens or stores. The attack requires that the sensitive data is stored in the same memory space as the USB data buffers and information. An Electromagnetic Fault Injection (EMFI) attack will be used to cause the USB memory buffer reads to cross into the secure storage space, and return sensitive data from FLASH or SRAM.

The remainder of this paper will be organized as follows. First, an introduction to relevant sections of the USB protocol will be given in Section 2.1. Then a summary of previous work

Table 9-2. Format of Setup Data

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Figure 1: USB Setup Data Fields.

in Section 2.2 will demonstrate how existing fault attacks have been used against USB stacks. The specific targets under attack will be introduced in Section 3 (Trezor Bitcoin Wallet) and Section 4 (FIDO2 key), followed by details of the specific USB processing logic being faulted. The physical setup will be discussed in Section 5, which will also introduce the open-source PhyWhisperer-USB. Finally results of the attack will be presented in Section 6, which will include examples of wider applicability of this specific vulnerability, and will then be followed by countermeasures and recommendations to avoid the types of attacks discussed in this paper.

2 Background

This paper will discuss only fault injection on USB stacks, and does not cover general fuzzing of USB stacks for which the reader is referred to [5–7, 14].

2.1 USB Protocol

A short introduction to relevant sections of the USB 2.0 protocol will be included here. The reader is referred to the full specification [1], or a text such as USB Complete [2] for a more complete overview.

USB data transfer events are called transactions, and all USB transactions are initiated by the host computer. The host computer can read and write data from different 'endpoints', where each endpoint may have a certain function or type depending on the device functionality. Each endpoint also has a fixed maximum transfer size depending on the specific standard (typically 8 – 512 bytes). Transferring a block larger than this maximum requires the host to repeatedly request a number of blocks of the maximum size until all data is transferred. To indicate the transaction is finished, the device will return a packet smaller than the maximum endpoint size, which indicates to the host that no more data is available.

The most basic form of these packets is called a *setup* packet, which is sent to a default endpoint on the device. The setup packet is used to perform initial configuration of the device.

As part of this configuration, *device descriptors* describe all aspects of the connected USB device. These descriptors are one of the first items requested by the host computer, as it helps the host understand how to talk to the new USB device, and what the device is. As part of the setup request, the fields shown in Figure 1 will be transferred from the host to the device (this figure is Table 9-2 from The USB Specification [1]).

Of particular importance, note the inclusion of the `wLength` field. The `wLength` field is set by the host computer, and indicates the maximum amount of data the device can return. Since the host may not know the full descriptor size or type, it can use a small value of `wLength` to request only the beginning part of the descriptor which would include information on the full size. The host can then decided to request the full descriptor to retrieve all details about the attached device

2.2 Previous Work

The relatively simple method of transferring blocks of data at a time to the host has been previously exploited to read entire memory segments out over USB. This was first demonstrated by Micah Scott, who used a voltage fault injection attack [12]. Scott demonstrated that corrupting the function sending data on the device could cause it to send almost unlimited data out the USB port – since the host computer sees valid USB requests coming back, it will continue to read from the USB endpoint.

As part of this work, Scott also built the FaceWhisperer. This name is a combination of FaceDancer (used for USB fuzzing) and ChipWhisperer (used for fault injection). The triggering of the fault is generated by having control over a USB Phy/MAC chip which generates the read requests. In this case Scott can also control how data is read back, and allows exceeding limits that a host OS USB stack would place on sizes of buffers.

The idea of FaceWhisperer has been further extended in the GreatFET project, with the addition of Kate Tempkin's GlitchKit [13][1]. GreatFET is based on a microcontroller which has a USB host stack built in, and GlitchKit allows triggering of fault injection from the GreatFET firmware. This close integration of low-level USB host (including fuzzing capability) with fault injection timing insertion looks to be valuable for future work.

[1] See https://github.com/usb-tools for extensive details of this work including open-source training material.

Figure 2: Trezor wallet enclosure and PCB.

3 Hardware Wallet Target

The first target this attack will be demonstrated on is a hardware Bitcoin wallet. These wallets are designed to store the private key used for accessing and sending bitcoins. Since the wallet itself could become physically damaged, the user must have a method of backing up the private key outside of the wallet. To simplify this a mnemonic representation of the key is often used, which is specified in BIP39[2]. This mnemonic representation consists of a series of English words, which reduces the chance of errors compared to copying a hexadecimal representation of the private key. This mnemonic representation is called the "recovery seed".

This mnemonic representation of the private key can also be protected with a password. If this feature is not used, the raw private key is effectively sitting in memory unencrypted. In addition the PIN code is also sitting in memory without any protection present. This means that for this bitcoin wallet, recovering the memory is sufficient to break the system security in the majority of cases.

Recovering the secret key provides an attacker with the ability to clone the wallet. This means they could observe the contents of the wallet, and withdraw bitcoin at a future date. An attack which does not physically damage the wallet has considerable value for the ability to leave a now-compromised wallet in the possession of the original owner.

3.1 Trezor Wallet Design

The Trezor wallet is open-source, which simplifies understanding how the attack works in practice. The sources for Trezor are available at `https://github.com/trezor/trezor-firmware`. The exact version of the source code can be found under the "legacy/v1.7.3" tag on GitHub, as the flaws disclosed in this paper have been fixed in the latest firmware release. The Trezor is based on a STM32F205, with the device shown without enclosure in Figure 2. Note the STM32F205 is just below the surface of the enclosure, which will allow our EMFI attack to work through the enclosure of the device.

[2]See `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`

Table 1: Trezor memory layout.

Address Start	Address End	Contents	Size
0x08000000	0x08007FFF	Bootloader	32 KiB
0x08008000	0x0800FFFF	Metadata	32 KiB
0x08010000	0x080FFFFF	Application	1088 KiB

The actual sensitive recovery seed is stored in flash memory in a 'metadata' section. It's located just after the bootloader, as shown in Table 1. The bootloader can be entered by holding down the two buttons on the front of the Trezor, and allows a firmware update to be loaded over USB. Since a malicious firmware update could simply read out this flash location, the bootloader will verify various signatures are present on a firmware update to prevent such an attack.

Various attacks have been presented (and fixed) on this device. One of interest was presented by 'Sunny' (no real name known), then written up in [11]. This attack demonstrated that sensitive metadata would be present in SRAM during the update process, and it was possible to reprogram the device to read out the SRAM.

A more complete attack was presented in the Wallet.fail disclosure, which glitched the STM32 device to move from the RDP2 level (which completely disables JTAG) to the RDP1 level (which enables JTAG to read from SRAM, but not code) [10]. This attack allowed reading out of the sensitive data stored in SRAM once again, and required a different fix compared to the previous disclosure.

Both of these attacks targeted SRAM, as the read protection on SRAM is less thorough than the read protection implemented on the FLASH memory. The Trezor copies sensitive data to SRAM only during the update process when it has erased the internal FLASH memory. If our attack corrupted the SRAM (or needed a power cycle to recover from error states), performing that erase is very dangerous. Instead the authors of this work will attempt to directly read out the FLASH memory using the EMFI attack.

3.2 USB Descriptor Processing

The specific descriptor request being handled is the WinUSB descriptor. The firmware will typically process various descriptor requests, and triggering the WinUSB descriptor simply requires forcing the device into bootloader mode and sending the appropriate USB request.

Our objective is to trigger the processing of the WinUSB descriptor request. We can perform this using the pyusb package from Python, with the details of the request shown in Listing 1. This request has a `bmRequestType` set for a Device to Host request, of type Vendor, with the recipient being an Interface. The `bRequest` of ASCII `'!'` is used to specify the WinUSB descriptor request. The remaining settings are for

`wValue = 0` and `wIndex = 5`, which is simply part of the requirements of the WinUSB descriptor request in question.

The final value of interest is the `wLength` setting, which we specify as `FFFF`. This will be sent to the USB device as the maximum allowed response size.

Listing 1: Sending WinUSB descriptor request from Python.

```
r = dev.ctrl_transfer(
      int('11000001',2), ord('!'), 0, 5,
      0xFFFF, timeout=1)
r = list(r)
```

We can then look into the `WinUSB.c` file to see the processing itself. The important section of the descriptor handling is shown below, which is changing the return buffer pointer to the descriptor itself, and changing the length of the returned value to be a minimum of either the requested size (`wLength`), or the size of the descriptor in question. Since the USB stack on the host may initially request only the beginning the the descriptor, the device cannot always return the full descriptor.

Listing 2: WinUSB.c descriptor handling.

```
static int winusb_control_vendor_request(
  usbd_device *usbd_dev,
  struct usb_setup_data *req,
  uint8_t **buf, uint16_t *len,
  usbd_control_complete_callback* complete
){
 \\...a bunch of code until we get to:
 *buf = (uint8_t*)(&guid);
 *len = MIN(*len, guid.header.dwLength);
 \\...a bunch more code finalizing...
}
```

The result of this C code is shown in Listing 3. Note that the register `r1` is loaded first with the `wLength` value passed from the host. Only if the value of `wLength` is longer than the size of the descriptor (`0x92`), is the value in `r1` overwritten. Thus by skipping this single check, we would be allowing the host provided value of `wLength` to decide on the amount of data returned.

Listing 3: Assembly output of two lines from Listing 2.

```
ldrh   r1, [len]
ldr    r5, =guid
cmp    r1, #0x92
it cs
movcs  r1, #0x92
str    r5, [buf]
strh   r1, [len]
```

Note that some *host* USB stacks may limit the value of `wLength`, in which case the code from Listing 1 must be modified to request a smaller data chunk. The typical limit is 4096 bytes on Windows hosts and with some Linux hosts, so setting `wLength = 0x1000` is the most data that can be transferred on those hosts. Newer USB controllers can successfully request full-sized packets, but may also require an updated libusb version which does not have the 'old' 4096 byte size limitations.

Figure 3: Solo Key, a FIDO2 authentication token.

3.3 Verifying Code Vulnerability

For this attack to work, an attacker needs to skip the comparison check. Rather than jumping immediately to fault injection, we can also perform a 'simulated' glitch. This is possible if we recompile and reprogram our own Trezor device, since the production units do not have JTAG debug enabled. But such simulated glitches are useful for developers who do not wish to fully instrument their system. This can be done by commenting out the comparison, or use an attached debugger to set a breakpoint before the new value is copied over and manipulate the program counter to bypass the instruction.

This will use the code from Listing 1 to send the WinUSB control request which should return with the guid structure. It sends a length request of 0xFFFF for the request, which should be paired down to 146 bytes by the code. Performing the simulated fault injection verifies the returned data includes the sensitive metadata that allows us to break the wallet.

We will next look at a second vulnerable device to understand the general applicability of the previous attack.

4 FIDO2 Authentication Key

As a second example, a FIDO2 authentication token is attacked using EMFI on the `wLength` comparison. The specific device under attack is a Solo Key, which is an open-source FIDO2 authentication token shown in Figure 3. These keys can be used for two-factor authentication of web services (such as by Google, Microsoft, etc). Microsoft has recently begun promoting the use of 'Password-less protection', which promotes the use of FIDO2 keys for authentication [9].

Note that the device we are attacking is not certified according to a FIDO 'Authenticator Level' – these levels provide information about varying degrees of protection against hardware attacks, including claimed fault injection protection.

When in operation, the FIDO2 authentication token generates a unique ECC key pair for each service registered with the token. The service which will use the token as an authenticator for the user must store the public key, and the private key is now stored on the token, and can be used to validate

Table 2: Location of several variables in Solo Key memory.

Address	Variable Name	Description
0x200000bc	`USBD_HID_Desc`	HID Descriptor.
0x20001b0c	`_signing_key`	Pointer to key.
0x20001b10	`master_secret`	HMAC secret.
0x20001b50	`privkey.8369`	ECC Private Key.
0x20001b70	`sha256_ctx`	SHA256 context.

requests in the future. If an attacker could recover the ECC private key without damaging the token, they could authenticate as the victim without causing the actual victim to realize they have been compromised.

As previously mentioned, we require the sensitive data (ECC private key) to be stored in the same memory space as part of the USB descriptor. The open-source nature of the Solo Key allows us to validate the specific memory layout that will allow the attack to succeed. An example of the resulting memory layout for several interesting variables is given in Table 2. This table shows in particular the USB descriptor `USB_HID_Desc` is contained before many of the interesting cryptographic variables. In this case leaking `privkey.8369` is most valuable, as this represents an array used to store the loaded ECC private key.

Attacking the token requires it to load the ECC private key, which is done by requesting the device perform an authentication request with the service of interest. The device will load the ECC private key into the memory array, at which point we can perform the required attack. Unlike the Trezor we cannot directly leak the key from FLASH memory, as the layout of the memory in the Solo Key stores the keys in an upper section of flash that is more than 64 Kbytes away from any vulnerable descriptors.

Once the key is loaded into RAM, we send a descriptor request that triggers the code in Listing 4. As shown in Listing 2 the `USBD_HID_Desc` variable is located approximately 7000 bytes before interesting sensitive data is loaded into RAM. Thus by setting our `wLength` value in the request to 7000 or greater we can recover the sensitive data, assuming the `MIN()` comparison is faulted.

Listing 4: Handling of descriptor in usbd_hid.c showing access to `USBD_HID_Desc` variable.

```
else if( (req->wValue >> 8) ==
            HID_DESCRIPTOR_TYPE){
    pbuf = USBD_HID_Desc;
    len = MIN(USB_HID_DESC_SIZ, req->wLength);
}
```

4.1 Finding Private Key Location

The previous section assumed knowledge of the memory layout to find the private key. This assumption is not required for the attack to succeed in practice, provided we assume the memory layout of the device remains constant during use.

Instead, an attacker must first run a request to generate a new credential on a target FIDO2 authentication device. The FIDO2 device will provide the attacker with the public key that was generated, and is associated with a private key stored on the FIDO2 device itself.

The attacker can then run the `wLength` attack to dump a large section of memory. It is now simple to search for standard representations of the key in memory, by generating candidate public keys from candidate private keys. If a candidate public key matches the known public key, the attacker now knows the storage location of the private key in memory. This brute-force search of 64 Kbytes of memory can be performed in less than 60 seconds.

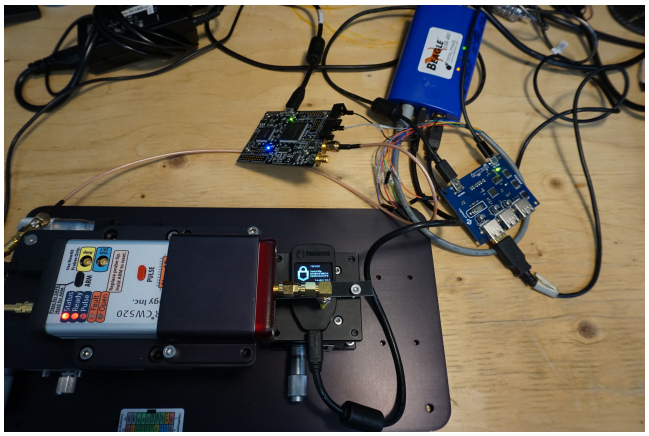

Figure 4: The Trezor EMFI setup (clockwise from top) includes a Beagle 480, YepKit USB Hub, Trezor target on XY table, ChipSHOUTER, and ChipWhisperer-Lite.

5 Fault Setup

Having demonstrated that the `wLength` attack will be successful by performing an instruction bypass with the debugger, we now need to use EMFI to cause the real instruction bypass in a practical setting. This section will concentrate on the specific example of the Trezor wallet attack. The overall setup and flow is similar between the two examples in this paper, the only difference is the specific descriptor requested, and the timing of the EMFI fault injection. Due to space limitations of this paper only the Trezor wallet attack is presented in detail.

Figure 4 shows the overall tooling. The Trezor is held down on an XY table with both of the front buttons held down. Holding the front buttons down forces the USB bootloader mode to be entered, as from the memory map shown in Table 1 only the bootloader-mode USB descriptors are located ahead of the sensitive metadata and can be used to leak those contents.

A ChipSHOUTER EMFI tool is used to generate the EMFI pulse. This is triggered from a TotalPhase Beagle 480 used to

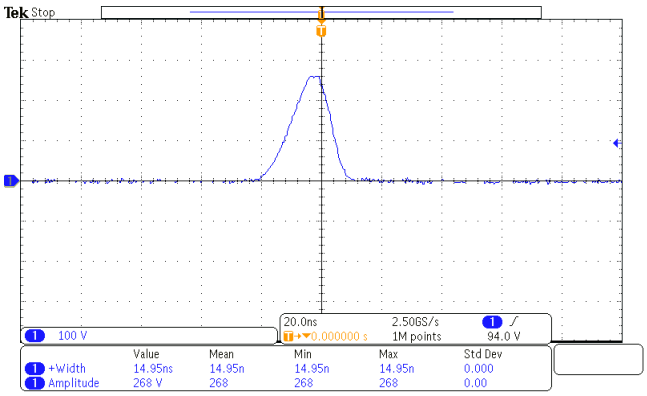

Figure 5: The inserted pulse measured at the ChipSHOUTER output, which reaches 268V in amplitude and is 15 nS width.

detect the exact timing of a specific USB request. This trigger is routed through a ChipWhisperer-Lite to allow modification of glitching offset & width. The Trezor is connected through a USB hub which allows us to power-cycle the target device. The power-cycling of the target is required since the EMFI glitch when inserted at the wrong location frequently crashes the target or triggers various error detection routines, including memory corruption detection, stack smashing detection, and bus faults.

5.1 EMFI Tooling

A ChipSHOUTER from NewAE Technology Inc. is used to generate the EMFI glitches. This is used with a ferrite-core coil, with a 1 mm ferrite core diameter. This injection tip is part of the standard ChipSHOUTER injection tip. The ChipSHOUTER is configured with a charge voltage of 400 V, and an external active-low trigger is used to trigger the EMFI injection.

The external active-low trigger comes from a 'glitch out' port of a ChipWhisperer-Lite. The ChipWhisperer-Lite has a MOSFET which can be used for VCC glitching, but in this case we use the MOSFET to drive the ChipSHOUTER trigger input. This allows us to manipulate the width and location of the glitch using Python-based tools that interface to both the ChipWhisperer-Lite and ChipSHOUTER.

The resulting waveform is shown in Figure 5. Note the 400 V setting of the charge voltage is not seen at the actual output due to the narrow pulse configuration.

5.2 Triggering and Scanning

Triggering the glitch is done by monitoring the USB traffic for the descriptor request that the code in Listing 2 sends. In this case the use of a TotalPhase Beagle 480 monitors for a DATA packet with contents `C0 21`. To determine the time difference between this USB request going 'over the wire' and the actual sensitive data, we instrument the open-source Trezor code. The code can be modified to include an I/O line trigger, which showed that the time delta varied from 4.2 to 5.7 μS. Knowing the timing based on this modified device, an unmodified device (without the I/O line trigger) can be rapidly attacked with an EMFI tool.

The timing has some variability due to the use of a queue-based processing of the USB messages. When the USB message comes into the Trezor, there is some time delay before it is processed. Rather than try to reduce this jitter, we simply reset the device and repeatably tried a fixed glitch with an offset at 4.4 μS. This value of 4.4 μS was selected as it empirically gave reasonably good results, and fit between the measured extremes of the jitter.

As we used a PC to send the descriptor request, we did not have an ability to carefully control the USB timing. Using an embedded USB host (such as GreatFET) would allow synchronizing the Trezor state machine processing the USB queue with the attack tooling state. It would be expected this could reduce the jitter to allow sending the USB request when the target device was in a known state (i.e., when it is about to check the queue for received USB messages).

5.2.1 Using Corrupt Packets for Glitch Timing

In this example, the open-source nature of Trezor made it possible to easily discover the correct glitch timing by instrumenting the firmware. The desired glitch location is during processing of the data (`wLength`) passed in the SETUP packet, and we set an I/O line trigger to indicate when this processing occurred. A pure black-box target would not allow this instrumentation, and here we demonstrate a simple method of finding this timing with a faster search than purely brute-force.

For this attack, we know we are trying to corrupt the `wLength` value processing. This value should be processed sometime between the SETUP transaction (which includes the `wLength`) and the IN transaction (which returns the descriptors). We can look for corrupt USB packets occurring immediately after the SETUP transaction to indicate when our glitch is most likely near the code of interest.

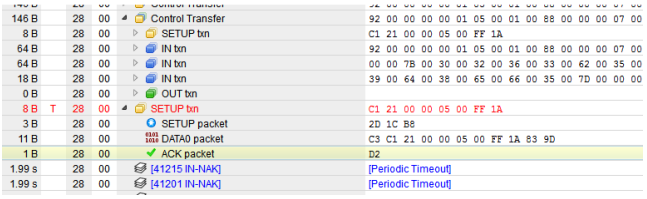

Figure 6: The SETUP phase in this control transfer is ACK'd, but the data phase does not follow as expected.

Figure 6 shows such an example. The upper part of that figure shows a correct 146-byte control transfers, which involves

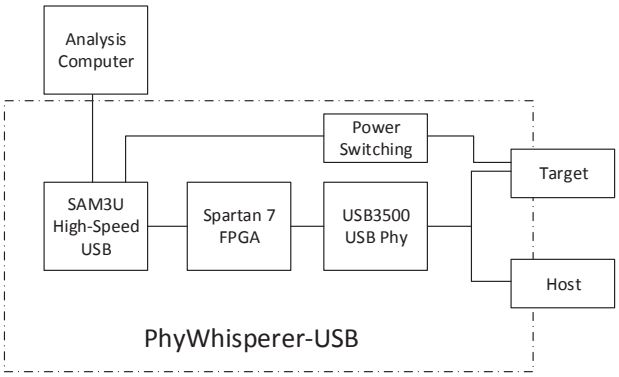

Figure 7: PhyWhisperer-USB combines a USB PHY with a FPGA.

one “SETUP txn" followed by three “IN txn".The lower part shows a corrupted transaction - the Trezor has ACK’d the “SETUP txn", but then never sends the follow-up data. The Trezor entered an infinite loop as it jumped to one of the various interrupt handlers for error detection. As the location of the fault is shifted along in time, various effects on the USB traffic are observed: moving the glitch earlier often prevents the ACK of the setup packet, moving the glitch later allows the first packet of follow-up data to be sent but not the second, and moving the glitch much later allows the complete USB transaction but then crashes the device. This knowledge helps us understand which part of the USB code the fault is being inserted into, even if that fault is still a sledgehammer causing a device reset instead of an intended single instruction skip.

Once approximate timing is known, it is easier to fine-tune this glitch location. Our actual exploitable target is only a single instruction which will require a small time-step size during the search. By instead looking for any corruption or crash during specific USB processing steps, we can use a much larger time-step size during the initial parameter search, without risk of missing a location of interest.

5.3 PhyWhisperer-USB

While GreatFET and GlitchKit provide fault injection capability, they have some limits on timing accuracy and capability due to use of a microcontroller host. The authors of this work initially used a TotalPhase Beagle 480 to provide more accurate fault injection timing. The objective of achieving more accurate timing and better tooling integration led to the creation of PhyWhisperer-USB.

PhyWhisperer-USB uses a FPGA to perform highly accurate glitch timing, with additional features not present in other tools such as rapid power cycling of the target device. A general block diagram of the device is shown in Figure 7, with the prototype shown in Figure 8. Full design details of the open-source hardware project are available online at https://github.com/newaetech/phywhispererusb.

Highly accurate fault insertion is possible as the USB3500 chip internally generates a 480 MHz clock to drive the USB sample logic. The data outputs from the USB3500 are synchronous to a clock that is derived from the 480 MHz sample clock. For glitch generation the FPGA uses this clock to internally generate a 240 MHz clock, giving up to 4.2 nS resolution on glitch offset timing. This allows faults to be inserted not only at the moment a packet is seen on the USB phy, but at arbitrary offsets with fine-grained resolution.

Compared to previous solutions, such as using the TotalPhase Beagle 480 to trigger external fault generation, the PhyWhisperer-USB has the advantage of remaining completely synchronous to the USB target device. That is triggers are always occurring relative to the exact clock edge of the USB bus, and offsets are measured in USB clock ‘ticks’. This is only possible by using the clock that has been phase-locked to the USB bus by the USB phy.

The current firmware supports triggering based on a packet or sequence being seen on the USB traffic. Thus another device is required to serve as the host - in this work it’s a Linux host computer, but it could be the GreatFET for example. This design allows PhyWhisperer-USB to be used with GreatFET to achieve better accuracy, without duplicating the work of GreatFET or GlitchKit.

In addition, PhyWhisperer-USB can be used to determine when invalid or corrupted packets are occurring. The location of these corrupted packets indicates where control flow of the DUT is becoming invalid, which can help in understanding the effect of inserted faults in the DUT. The close integration of the glitch generation logic and sniffing logic simplifies automatic searches for useful glitches by monitoring for corrupt responses, as the timing information is consistent between the sniffing and glitch generation logic.

The PhyWhisperer-USB hardware makes possible arbitrary packet insertion, which would allow the tool to be used as a low-level USB PHY interface for fuzzing and more advanced fault injection work. These fuzzing capabilities are not present in the current firmware and left as future work.

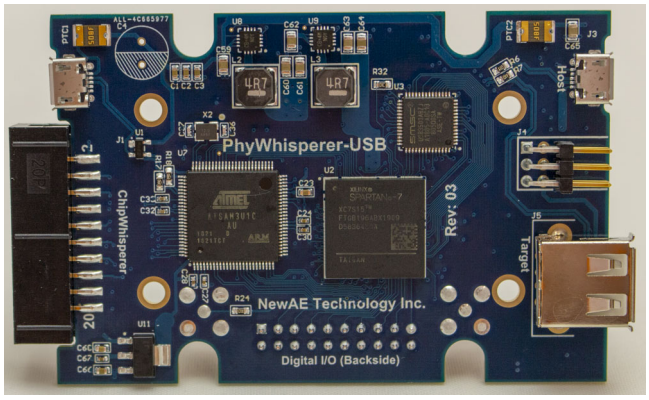

Figure 8: PhyWhisperer-USB prototype implementation.

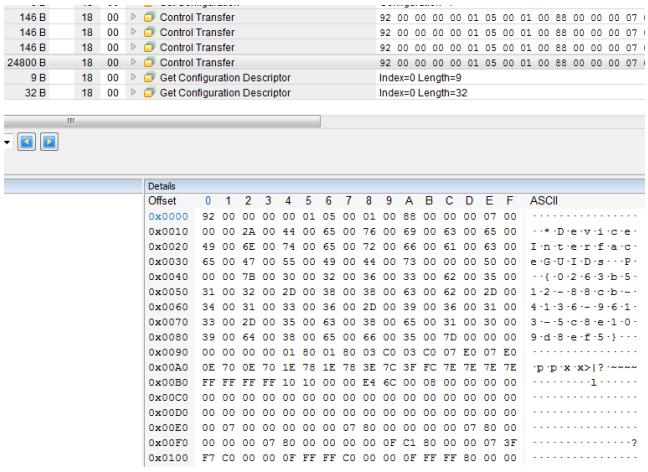

Figure 9: A successful glitch returns larger than the 146-byte descriptor.

6 Results and Countermeasures

The corruption of the `wLength` processing allows dumping of up to 64 Kbytes of memory from the target. Previously we described the general memory layout of two specific targets, but now the use of the memory dump for recovery of secret information will be described.

6.1 Results on Bitcoin Wallet

A correct timing of the glitch results in the expected over-sized packet. This is shown in Figure 9 which has a number of normal responses followed by one too-long response. The normal response of 146 bytes is followed by a response of 24800 bytes.

In the case of the Trezor bitcoin wallet, applying this attack is trivial due to the BIPS39 secret key encoding. The secret key is encoded as a series of English words, meaning an attacker simply looks for a series of strings in the returned over-sized USB packet. In this example the strings returned *exercise muscle tone skate lizard trigger hospital weapon volcano rigid veteran elite speak outer place logic old abandon aspect ski spare victory blast language*.

An attacker that has these sequence of words can now clone the wallet[3]. This means not only can the attacker steal bitcoin currently in the wallet, they can also monitor the wallet contents to determine the total value of this wallet and steal them at a future date. The legitimate user does not know this cloning has occurred, and the EMFI attack would not have physically damaged the wallet. In the case the secret was stored in FLASH memory, and we successfully read this secret out. We will now consider attacking the Solo key where a RAM stored secret is recovered.

[3]Users can optionally add a 'passphrase', which is not enabled by default. If the passphrase was enabled an attacker would need to know or guess this passphrase.

6.2 Results on Solo Key

A successful EMFI attack on the Solo Key results in the leak of a single ECC private key. This requires the attacker to have physically taken possession of the Solo Key. They would then select a service of interest (for example, https://microsoft.com) and send a request to the legitimate Solo Key under attack. This request causes the private ECC key to be loaded in memory.

Performing the `wLength` attack causes a large section of memory (RAM) to be dumped, including the ECC private key used for authentication with the requested service. The attacker must determine the memory layout to understand where the private key was stored; Section 4.1 describes how this is accomplished in practice.

The attacker can now use this ECC private key to respond to authentication requests when accessing user accounts. The original token can be returned to the user, who will not be able to detect tampering since no physical modifications were made to the token. Again the use of EMFI triggered from USB packets allows an attack which requires no modifications to the target device, complicating attempts to detect tampering.

6.3 General Applicability

The attack requires that sensitive data be stored in memory within 64 KBytes of the USB descriptors or buffers, something that can be verified by designers by inspecting the mapping files providing address information of variables. Fundamentally the attack is relevant only when sensitive data is stored on the device within this 64 KByte boundary of USB descriptors or buffers.

Assuming the sensitive data is stored within this boundary, the USB stack can also be trivially inspected to find if the `wLength` processing appears vulnerable. The `MIN()` call can be found in many USB stacks. If not using the exact same setup, the general passing of a maximum-size `wLength` is present in many general RTOS stacks.

Specific examples on three popular RTOS will be presented next – note an EMFI attack has not been performed on these stacks, as the attack would be simply be on an arbitrary implementation and not a real product. The objective of the following subsection is to demonstrate where such vulnerable code lies in other RTOS stacks.

6.3.1 RT-Thread

RT-Thread is an open-source IoT operating system with a 'wide scalability'. It contains a complete USB device stack.

We can observe the USB processing in `core.c`[4] as an example of similar vulnerable code which compares `wLength` to descriptor sizes. The specific vulnerable code example is shown in Listing 5.

Listing 5: Example of USB processing in the RT-Thread stack.

```
size = (setup->wLength>USB_DESC_LENGTH_DEVICE)
  ? USB_DESC_LENGTH_DEVICE : setup->wLength;
```

6.3.2 Amazon FreeRTOS

FreeRTOS is a popular RTOS first release in 2003, which supports a wide variety of host devices. Amazon offers 'IoT Extensions' to this under the Amazon FreeRTOS project at https://aws.amazon.com/freertos/. The USB stacks are provided by the host device vendors in this solution, so there is no single USB stack to analyze. For example as part of FreeRTOS we can analyze the NXP stack USB processing in the file `usb_device_ch9.c`[5]. The vulnerable comparison of `wLength` is shown in Listing 6.

Listing 6: FreeRTOS USB is provided per vendor, this example comes from an included NXP stack.

```
if (*length > setup->wLength)
        {
            *length = setup->wLength;
        }
```

The lack of a standard USB stack means that each vendor code would need to be inspected for this potential vulnerability.

6.3.3 ARMmbed

The final RTOS analyzed is the ARMmbed stack, which is supported by arm. The USB stack processing of interest is contained in the file `USBDevice.cpp`[6]. In this stack a variable called `_transfer.remaining` is always written with the size of the object in memory to transfer. The opportunity to perform fault injection is later in the code, when a check of the `wLength` size happens. This is shown in Listing 7, where fault injection would allow the user-supplied value of `wLength` to be used to define the transfer size.

Listing 7: Example of USB processing in the ARMmbed RTOS.

```
/* Transfer must be less than or equal to */
/* the size requested by the host */
if (_transfer.remaining >
    _transfer.setup.wLength) {
        _transfer.remaining =
         _transfer.setup.wLength;
}
```

[4] https://github.com/RT-Thread/rt-thread/blob/master/components/drivers/usb/usbdevice/core/core.c

[5] https://github.com/aws/amazon-freertos/blob/master/vendors/nxp/LPC54018/utilities/usb_device_ch9.c

[6] https://github.com/ARMmbed/mbed-os/blob/master/usb/device/USBDevice/USBDevice.cpp

6.4 Countermeasures

The ability of fault attacks to be used against embedded systems is well known [3], and various countermeasures can be generally applied. We will not duplicate this prior work, and instead we will be focusing on countermeasures more specific to this use case. These countermeasures have been applied by the device designers (Trezor Wallet and Solo Key) fixing the flaw disclosed in this paper.

6.4.1 Use of MPU Traps

A simple method of preventing read past end of buffer attacks is to use the Memory Protection Unit (MPU) that is present on even many small microcontrollers, including the STM32F205 in the Trezor wallet. Surrounding the sensitive metadata code area with guard bands would have completely prevented the ability of an attacker to read out the sensitive data. Such guard bands should always be used when possible, and especially if the sensitive data is in a known variables or location. Besides the specific attack shown here, such guard bands could prevent other read past end of buffer attacks.

The sensitive memory location can also be turned "on and off" by enabling or disabling memory access on the entire sensitive area, and only enabling memory access inside the valid functions. This is effectively emulating more advanced memory management features that are not available on typical small microcontrollers.

6.4.2 Limiting Response Sizes

By the USB specification, the 16-bit `wLength` could encode up to a 65 535 byte request size. But for an embedded device with a known functionality, it is unnecessary to support this full response size as no descriptor request would approach this size (typically being in range 16 – 240 bytes). Thus a simple mask could be used at in the USB stack, to prevent sending back of such large data blocks as part of a control transfer.

6.4.3 Random Timing Jitter

The USB specification defines the maximum response times for requests, normally between 50 – 500 mS depending on the request. Relative to the device clock speed this represents a considerable amount of random jitter that can be added to complicate the attackers search phase.

6.5 Other Fault Attack Targets

It should be clear that injecting a fault attack through the enclosure of this device remains possible, even when very accurate timing is required on the fault injection location. This work has concentrated on the specific attack on the USB stack, as this attack can be found in many other devices. Fault attacks may have other methods of breaking the device, such as bypassing the verification check used to prevent a malicious firmware image from simply reading out the secret data.

7 Conclusions

Performing an EMFI attack through the enclosure of a device demonstrates how a practical attack could be used with real consumer devices. This attack exploits a common logical flow present in almost all USB stacks to allow reading up to 64 Kbyte of data from the device.

Performing this attack requires very careful timing of the fault injection location relative to the USB transactions. To assist with this, an open-source hardware tool called PhyWhisperer-USB has been introduced. This tool allows triggering of a fault injection platform from a USB message with very high temporal accuracy.

Various countermeasures are possible, beyond standard fault-resistant design countermeasures. This includes using memory guards around sensitive data, and masking the size of returned USB data to reflect expected buffer sizes and not blindly returning the physical maximum allowed buffer sizes.

This paper has demonstrated the attack against two specific devices: a bitcoin hardware wallet, and a FIDO2 authentication token. The flaws were promptly fixed by the vendors using the suggested countermeasures. Because the flaws are generic across many other USB stacks, vendors of almost all devices which have sensitive data present should review their code for the possible vulnerability to the attack discussed here.

Acknowledgments

Thanks to Dmitry Nedospasov for extensive discussions on the Trezor metadata format and attacks on the Trezor device. Thanks to Pavol Rusnak for discussions and suggestions on countermeasures, and quickly deploying those countermeasures to the Trezor users. Thanks to Conor Patrick and Emanuele Cesena for discussions of FIDO2 protocol, and Solo Key applicability and fixes. Thanks to Jean-Pierre Thibault for assistance with development of the PhyWhisperer-USB. Finally this paper has been substantially improved thanks to the feedback of the anonymous WOOT reviewers, and with help from our WOOT shepherd.

References

[1] Universal Serial Bus Specification, April 2000.

[2] Jan Axelson. *USB Complete*. 5th edition, 2015.

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, February 2006. doi:10.1109/JPROC.2005.862424.

[4] Common Criteria. Common Criteria for Information Technology Security Evaluation, 2017. Available at https://www.commoncriteriaportal.org/cc/.

[5] Travis Goodspeed. Facedancer21. URL: http://goodfet.sourceforge.net/hardware/facedancer21/.

[6] Travis Goodspeed. Emulating USB Devices with Python, July 2012. URL: http://travisgoodspeed.blogspot.com/2012/07/emulating-usb-devices-with-python.html.

[7] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. pages 46–52, November 2010. doi:10.1109/EC2ND.2010.16.

[8] Keith Mayes and Konstantinos Markantonakis. *Smart Cards, Tokens, Security and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[9] Microsoft. Password-less protection, 2018. Available at https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE2KEup.

[10] Dmitry Nedospasov, Josh Datko, and Thomas Roth. Wallet.Fail, December 2018. URL: http://wallet.fail.

[11] Saleem Rashid. Extracting TREZOR Secrets from SRAM, August 2017. URL: https://saleemrashid.com/2017/08/17/extracting-trezor-secrets-sram/.

[12] Micah Scott. A USB Glitching Attack. *PoC||GTFO*, 2(0x13):30 – 37, October 2016.

[13] Kate Tempkin and Spill Dominic. Opening Closed Systems With GlitchKit, December 2017. URL: https://media.ccc.de/v/34c3-9207-opening_closed_systems_with_glitchkit.

[14] Rijnard van Tonder and Herman Engelbrecht. Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation. 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/van-tonder.