



circuit cellar

Inspiring the Evolution of Embedded Design

Embedded System Essentials

Attacking USB Gear with EMFI

Pitching a Glitch

Many products use USB, but have you ever considered there may be a critical security vulnerability lurking in your USB stack? In this article, Colin walks you through an example product that could be broken using electromagnetic fault injection (EMFI) to perform this attack without even removing the device enclosure.

By
Colin O'Flynn

COLUMNS

In past articles I've taken you through various theoretical attacks on embedded systems, demonstrated various attacks in standard systems and summarized

recent work from relevant conferences. This article is something new. I'm going to be presenting a new attack. While it's been disclosed to the vendor—and should have been fixed by the time you read this—you are getting as close to the bleeding edge of attack information as I can present in this article.

Our victim will be a Trezor bitcoin wallet. This little device can be used to store Bitcoins, which ultimately means a method of securely storing a private key used for cryptographic operations. We don't need to dig into details of the wallet operation, but a critical piece of information to understand is the idea of a "recovery seed". This recovery seed is a series of words which encodes a recovery key, and knowing that recovery seed is sufficient to recover the secret key.

This means someone who steals only that recovery seed—without further access to the wallet—could access funds stored on the wallet itself. It goes without saying that an attack finding that key would be rather detrimental to our experience using the wallet.



FIGURE 1
The Trezor wallet is shown here with the enclosure removed.

```
#define FLASH_BOOT_START (FLASH_ORIGIN)
#define FLASH_BOOT_LEN (0x8000)

#define FLASH_META_START (FLASH_BOOT_START + FLASH_BOOT_LEN)
#define FLASH_META_LEN (0x8000)

#define FLASH_APP_START (FLASH_META_START + FLASH_META_LEN)
```

It should be noted that there has been some other work that inspired this attack. The “wallet.fail” presentation at the Chaos Communication Congress (CCC) by Dmitry Nedospasov, Josh Datko and Thomas Roth demonstrated how one could break the STMicroelectronics (ST) STM32F2 security protection, allowing the dumping of its SRAM contents. Instead, I’m going to be showing you how to directly dump flash memory where the seed is stored. So, it’s a different attack but with similar end results.

I’m going to be using electromagnetic fault injection (EMFI), enabling us to actually

perform the attack *without even removing the enclosure*. This means someone can perform the attack without leaving a trace of modifying the wallet, no matter how carefully you inspect it. Before we get to the real attack, we need to cover some background.

POWERFUL EMFI

EMFI is a powerful method of performing fault injection attacks. Typically, we use some sort of pulse generator to drive an inductor and the inductor will generate a strong magnetic field. If you bring this magnetic field near a chip, this will induce voltages inside

LISTING 1

memory.h showing `FLASH_META_START` occurs after the bootloader and before the application

```
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) {
    (void)complete;
    (void)usbd_dev;

    if (req->bRequest != WINUSB_MS_VENDOR_CODE) {
        return USBD_REQ_NEXT_CALLBACK;
    }

    int status = USBD_REQ_NOTSUPP;
    if (((req->bmRequestType & USB_REQ_TYPE_RECIPIENT) == USB_REQ_TYPE_DEVICE) &&
        (req->wIndex == WINUSB_REQ_GET_COMPATIBLE_ID_FEATURE_DESCRIPTOR)) {
        *buf = (uint8_t*)&winusb_wcid;
        *len = MIN(*len, winusb_wcid.header.dwLength);
        status = USBD_REQ_HANDLED;
    } else if (((req->bmRequestType & USB_REQ_TYPE_RECIPIENT) == USB_REQ_TYPE_INTERFACE) &&
        (req->wIndex == WINUSB_REQ_GET_EXTENDED_PROPERTIES_OS_FEATURE_DESCRIPTOR) &&
        (usb_descriptor_index(req->wValue) == winusb_wcid.functions[0].bInterfaceNumber))
    {
        *buf = (uint8_t*)&guid;
        *len = MIN(*len, guid.header.dwLength);
        status = USBD_REQ_HANDLED;
    } else {
        status = USBD_REQ_NOTSUPP;
    }

    return status;
}
```

LISTING 2

The function `winusb_control_vendor_request` from `winusb.c` responds to requests for various information related to `WinUSB` over the control USB endpoint. Note the call “`MIN(*len, guid.header.dwLength)`” which decides on the length of the returned response.

metal on the chip. The result is an ability to manipulate internal voltage levels and insert ringing onto the power bus, causing the device to misbehave. These misbehaving activities are what we refer to as faults or glitches. Such faults or glitches could corrupt data (registers, SRAM) or corrupt program flow.

The Trezor wallet is open-source, which makes this attack a wonderful demonstration to teach you about EMFI and fault injection. You can freely modify the code, program old versions before they patched the bug, and generally perform other useful work to demonstrate this attack.

You can see the sources for Trezor on github. See the *Circuit Cellar* article materials webpage for the specific github link. If you want to follow this article, be sure to select the “v1.7.3” tag on GitHub. These flaws are fixed in a firmware release that will be available by the time you read this article, so you should look at the older (vulnerable) code to better understand the exact attack. The Trezor is based on ST’s STM32F205 and you can see with Trezor sans enclosure in **Figure 1**. Note that the STM32F205 is just below the surface of the enclosure—a feature we will use to improve our attack.

The actual sensitive recovery seed is stored in flash memory. It’s located just after the bootloader, as shown in **Listing 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 that various signatures are present on a firmware update to prevent such an attack. Loading unverified firmware would be one method of attack, but isn’t what we are going to use. The problem with all of these attacks is that the design of the Trezor erases the flash memory *before* loading and validating the new file, storing the sensitive metadata in SRAM during this process. The wallet.fail disclosure actually attacked this, since it’s possible to glitch the STM32 to go from code read protection level RDP2 (which completely disables JTAG) to level RDP1 (which enables JTAG to read from the SRAM, but not from the code).

If our attack corrupted the SRAM—or needed a power cycle to recover from error states—performing that erase is very dangerous. The wallet.fail attack was able to recover the SRAM, but the attack method we will use could corrupt the SRAM. That means any mistake would permanently destroy the recovery seed. Instead, we are going to try and directly read out the flash memory. This is much safer since we never perform an erase command, meaning the data is safely stored in memory waiting for us to extract it.

USB READ REQUEST

Because the bootloader contains USB, it also contains very standard USB processing code. Part of this is shown in **Listing 2**, which comes from the file `winusb.c`. I’ve chosen this particular request because there are actually two data structures present that are returned by this code—one is stored in FLASH and one is stored in SRAM. The USB request being processed first checks some information sent about the request. It looks for a matching `bRequest`, `bmRequestType` and `wIndex` which are all attributes of a USB request. Finally, the USB request itself contains a `wLength` field, which is how much data the computer is requesting be sent back. I can freely request up to `0xFFFF` bytes of data—and that is exactly what I will do. But, as you can see, the code does a `MIN()` operation to limit the length of the actual data sent back to be the minimum of either the requested length or the size of the descriptor I will send back.

So, what happens if that check was wrong? While it would let me send back the descriptor, along with all the 64K (`0xFFFF`) bytes of data that lies after the descriptor itself. This includes our precious metadata—the USB stack simply sends back the block of data as the computer requested. The entire security of the system depends on one simple length check!

If you’ve read a few of my articles, you might guess I’ve got a plan. We will be using

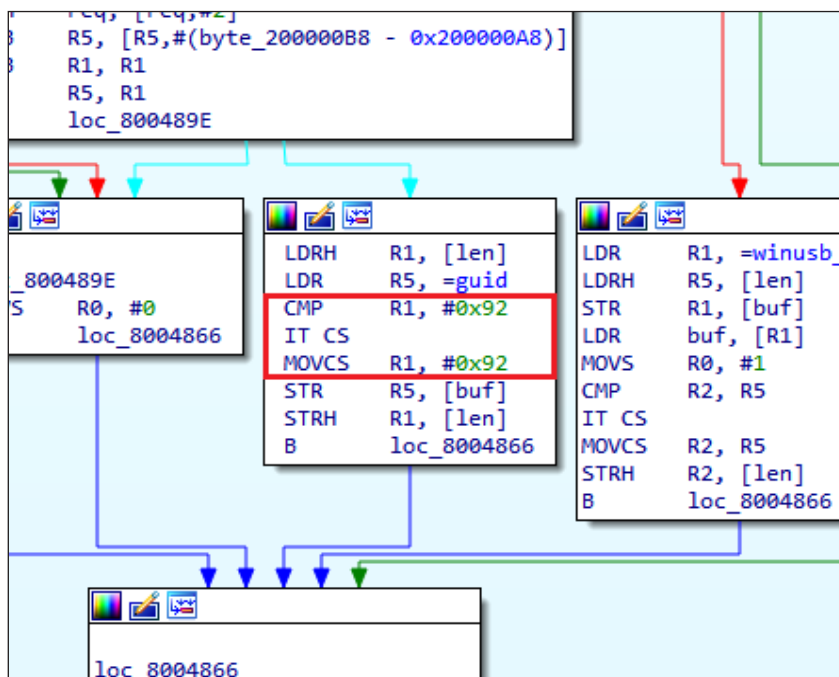


FIGURE 2

IDA disassembly of the function in question ultimately shows a single assembly instruction separates your sensitive data from being politely sent back on the USB port.

| Index | m:s.ms.us.ns | Len | Err | Dev | Ep | Record | Summary |
|-------|------------------|---------|-----|-----|----|-------------------------------|---------------------------------|
| 0 | 0:00.000.000.000 | | | | | ● Capture started (Aggregate) | [02/06/19 00:45:55] |
| 1 | 0:00.000.000.000 | | | | | 🚩 <Host connected> | |
| 2 | 0:00.000.633.500 | | | | | 🚩 <Full-speed> | |
| 3 | 0:23.658.183.950 | 146 B | | 22 | 00 | 📄 Control Transfer | 92 00 00 00 00 01 05 00 01 00 8 |
| 24 | 0:06.791.576.583 | 146 B | | 22 | 00 | 📄 Control Transfer | 92 00 00 00 00 01 05 00 01 00 8 |
| 45 | 0:03.879.450.166 | 146 B | | 22 | 00 | 📄 Control Transfer | 92 00 00 00 00 01 05 00 01 00 8 |
| 66 | 1:58.972.722.583 | 65535 B | | 22 | 00 | 📄 Control Transfer | 92 00 00 00 00 01 05 00 01 00 8 |
| 4171 | 0:11.333.695.616 | | | | | ● Capture stopped | [02/06/19 00:48:40] |

fault injection to bypass the check that depends on a single instruction. Before we dive into details of performing the actual fault, let's do a bit of "sanity check" on my claims. You can use these sanity checks in your own code to help understand the impact of similar vulnerabilities.

DISASSEMBLING CODE

The first sanity check is to confirm that a simple fault model can cause our intended operation. This can be trivially confirmed by inspecting a disassembly of the code, done with IDA in **Figure 2**. Note in particular that due to the resulting code flow, we need to skip only a single instruction to accomplish our goal of having the user-supplied length field be accepted.

The second sanity check will be to confirm there is not some higher-layer protection. For example, maybe the USB stack does not actually accept such a large response given that there's no actual need for this? This is a little harder to prove by simple inspection, but the open-source nature of the Trezor makes this possible. What we can do is modify the code to simply comment out the security check. If you didn't want to recompile the code, but did have debugger access, you could also use an attached debugger. Use the debugger to set a breakpoint before the new value is copied over and toggle the status of the flag, or manipulate the program counter to bypass the instruction.

Validating this sanity check will be done in the same way as the actual attack. This will use the code from Listing 2. This code sends 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. As you can see from **Figure 3**, when I do not modify the instruction, the USB request results in the expected-size response. Modifying the instruction (or using a debugger to manually clear the comparison flag) to bypass this check results in a full-size response. This demonstrates that there is no "hidden feature" that will fundamentally prevent the attack from working. With that knowledge, let's move onto getting this thing talking to us!

USB TRIGGERING AND TIMING

Before we can talk about how we insert the glitch, we need to know where to insert the glitch. We do know the exact code that triggers the glitch, and we do know the command we sent over USB. But we need to get better than that to introduce the exact instruction. In my case, since I have access to the software I'm going to "cheat" during my first test and measure the actual execution time. If I didn't have this capability, I would end up with a much slower sweep of possible locations.

The first thing I'll do is get a more solid trigger on the USB data itself. The entire area of using USB for glitch triggering was actually started by Micah Scott, who demonstrated voltage glitching to dump the firmware from a drawing tablet and developed a simple module to perform real-time glitching (which she called the FaceWhisperer). Instead I'm going to use a Total Phase Beagle 480, which can perform triggering based on physical data going over the USB line. The setup for that is shown in **Figure 4**. The Total Phase Beagle 480 also has a beautiful sniffer interface, so I can sniff the traffic and better understand what malformed packets are coming back. This capability is very useful since I can see, for example, the exact portion of the USB request being interrupted/corrupted. That might give me some hints about how far into the code the program has executed.

FIGURE 3

Using a debugger to step over the single check (or recompiling the code) shows that large chunks of memory will be sent back on request.

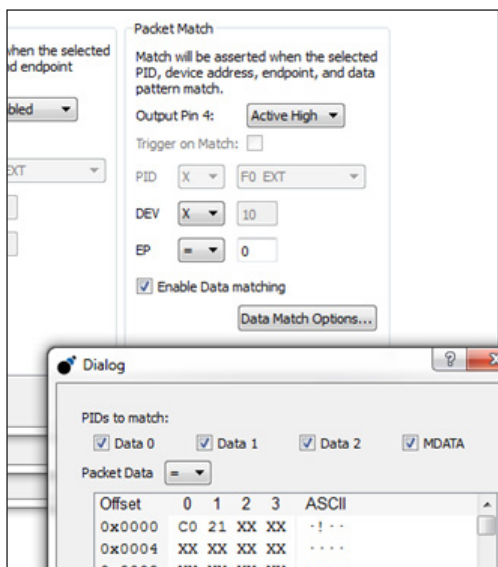


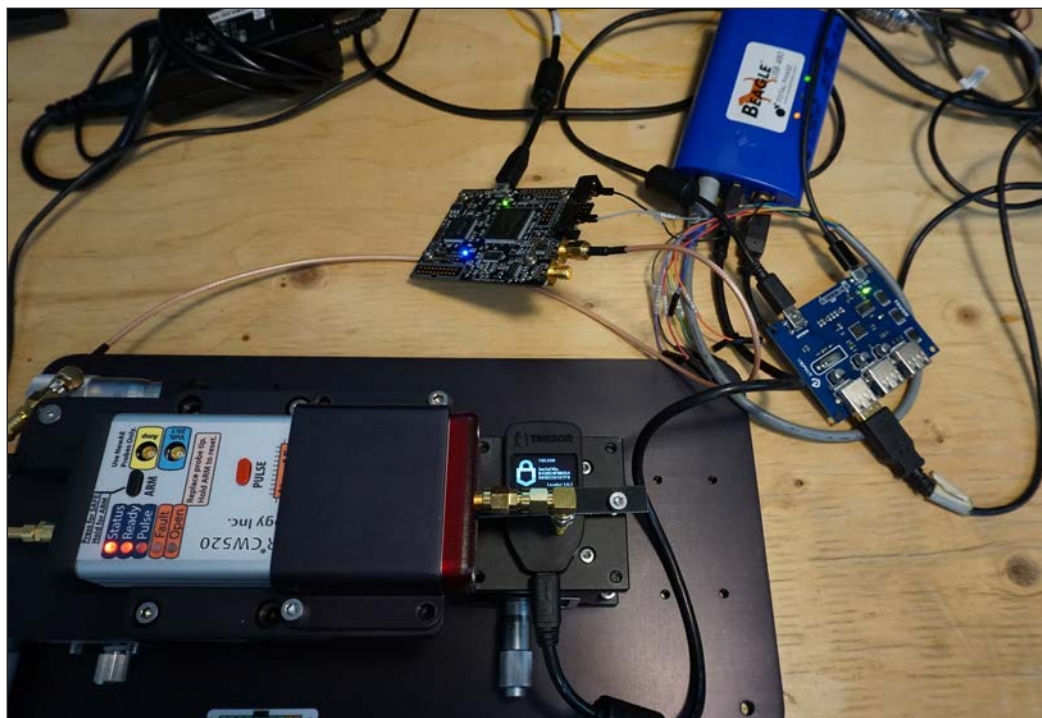
FIGURE 4

The USB protocol analyzer is setup to trigger on a specific packet related to our request.

COLUMNS

FIGURE 5

Complete setup of the EMFI attack including Beagle 480 for trigger generation, ChipWhisperer for timing modifications, ChipSHOUTER for EMFI insertion and a USB hub to power cycle the target.



Besides FaceWhisperer and the Beagle 480, there are other methods of triggering the glitch. Great Scott Gadgets offers its GreatFET device that has a module called GlitchKit. GlitchKit provides similar triggering capabilities, but generates the requests from the GreatFET itself. As of this writing the GlitchKit has more limited response capability, so I wasn't able to read the entire response back. Finally, you could look into a simple circuit using a USB PHY—such as Microchip Technology's USB3500—and an FPGA. Watch for the future open-source PhyWhisperer-USB from NewAE Technology which will give you that capability.

Once we have a trigger based on the USB request going "over the wire", we can insert a trigger by setting an I/O pin high when the sensitive code runs. We use this for characterizing the system, since we can use an oscilloscope to measure the time from the USB packet going over the wire to the sensitive code operating. In this case, the time ends up

being around 4.2 μ s to 5.5 μ s. It's not perfect timing, because there appears to be some jitter due to the USB packets being processed by a queue. We have just learned that, when performing the fault injection demo, we should expect that we do not achieve perfect reliability.

GLITCHING THROUGH THE CASE

For inserting the glitch, I'm using a setup as shown in **Figure 5**. This includes a ChipSHOUTER EMFI platform, a manual XY table for positioning the coil, the Trezor target, the Beagle 480 to generate a trigger, a ChipWhisperer to generate the timing offset and a Yepkit USB hub which provides a simple API to power cycle attached devices. The power cycle capability is useful as we will be very frequently crashing the target device.

A very simple script (shown in **Listing 3**) enables me to power-cycle the device and issue the WinUSB request. The physical "jig" that holds the Trezor actually holds the two power buttons down, ensuring it always enters bootloader mode on start-up. We want to use the bootloader since the bootloader is at a lower address than the metadata, so dumping any memory from within the bootloader is more useful when it comes time to recover the metadata.

The success rate is low—less than 0.1% of glitches are successful. We can however achieve a successful glitch within about 1-2 hours on average, making it a relatively useful attack in practice. A successful glitch is one where the USB request comes through with the full length of data, since I was able



ABOUT THE AUTHOR

Colin O'Flynn (colin@oflynn.com) has been building and breaking electronic devices for many years. He is an assistant professor at Dalhousie University, and also CTO of NewE Technology both based in Halifax, NS, Canada. Some of his work is posted on his website at www.colinoflynn.com.

```

import time
import time
import usb
import usb.core
import chipwhisperer as cw

def get_winusb(dev, scope):
    """WinUSB Request is most useful for glitch attack"""
    scope.io.glitch_lp = True #Enable glitch (actual trigger comes from Total Phase USB Analyzer)
    scope.arm()
    resp = dev.ctrl_transfer(int('11000001', 2), ord('!'), 0x0, 0x05, 0xFFFF, timeout=1)
    resp = list(resp)
    scope.io.glitch_lp = False #Disable glitch
    return resp

def reset_trezor():
    """Requires a YK USB Hub - has power control of each port"""
    subprocess.check_output([r'ykushcmd.exe', '-d', '1'])
    time.sleep(0.5)
    subprocess.check_output([r'ykushcmd.exe', '-u', '1'])
    time.sleep(1)

# ChipWhisperer used for trigger delay only
scope = cw.scope()
target = cw.target(scope)

# Values found from sweeping around
scope.clock.clkgen_freq = 147E6
scope.adc.basic_mode = "rising_edge"
scope.adc.samples = 500
scope.glitch.clk_src = "clkgen"
scope.glitch.output = "enable_only"
scope.glitch.trigger_src = "ext_single"
scope.glitch.repeat = 1
# Original extclock was 100MHz, so we scale offset
# relative to our actual clock to maintain 4.4uS
scope.glitch.ext_offset = 440
scope.glitch.ext_offset = (scope.glitch.ext_offset / 100.0E6) * scope.clock.clkgen_freq

dev = None

#Loop until we get too large a response
while True:
    if dev is None:
        dev = usb.core.find(idProduct=0x53c0)
        dev.set_configuration()

    try:
        #Perform USB request - glitch trigger happens via
        # TotalPhase Beagle 480
        res = get_winusb(dev, scope)
        if(len(res)) > 146:
            print("Data Over-Run Detected - DONE")
            break
    except usb.USBError:
        reset_trezor()
        res = None
        dev = None

f = open("outputresults.bin", "wb")
f.write(bytearray(res))
f.close()

```

LISTING 3

Shown here is a complete attack script in Python, which sends the USB requests while inserting faults.

| | | | | | | | |
|----|---------|------------------|--------|----|-------|------------------|--|
| FS | 6546642 | 0:00:056.668.166 | 146 B | 28 | 00 | Control Transfer | 92 00 00 00 00 01 05 00 01 00 88 00 00 00 07 00 00 00 2A 00 44 00 65 00... |
| FS | 6546643 | 0:00:000.000.000 | 8 B | 28 | 00 | SETUP txn | C1 21 00 00 05 00 FF 1A |
| FS | 6546647 | 0:00:000.025.333 | 64 B | 28 | 00 | IN txn | 92 00 00 00 00 01 05 00 01 00 88 00 00 00 07 00 00 00 2A 00 44 00 65 00... |
| FS | 6546651 | 0:00:000.070.750 | 64 B | 28 | 00 | IN txn | 00 00 7B 00 30 00 32 00 36 00 33 00 62 00 35 00 31 00 32 00 2D 00 38 00... |
| FS | 6546655 | 0:00:000.071.083 | 18 B | 28 | 00 | IN txn | 39 00 64 00 38 00 65 00 66 00 35 00 7D 00 00 00 00 00 |
| FS | 6546659 | 0:00:000.026.333 | 0 B | 28 | 00 | OUT txn | |
| FS | 6546663 | 0:00:025.202.500 | 8 B | T | 28 00 | SETUP txn | C1 21 00 00 05 00 FF 1A |
| FS | 6546664 | 0:00:000.000.000 | 3 B | 28 | 00 | SETUP packet | 2D 1C B8 |
| FS | 6546665 | 0:00:000.003.416 | 11 B | 28 | 00 | DATA0 packet | C3 C1 21 00 00 05 00 FF 1A 83 9D |
| FS | 6546666 | 0:00:000.008.666 | 1 B | 28 | 00 | ACK packet | D2 |
| FS | 6546667 | 0:00:000.013.166 | 1.99 s | 28 | 00 | [41215 IN-NAK] | [Periodic Timeout] |
| FS | 6546668 | 0:02:000.005.333 | 1.99 s | 28 | 00 | [41201 IN-NAK] | [Periodic Timeout] |

FIGURE 6

A physical USB analyzer (compared to attempting to use a software-only solution) is critical to see mangled packets on the bus, which lets us understand how far into requests the target got before freezing.

to bypass the length check. Finding the exact location takes some experimentation—you will get many system crashes due to memory errors, hard faults and resets. But if you are using a hardware USB analyzer such as the Beagle 480 you can see where these errors are happening, which helps you understand the glitch timing. If we didn't have the inside knowledge of the I/O pin we could toggle, this would be very valuable.

Figure 6 shows such an example. Note the USB transaction when performed correctly has a few steps. The upper part of that figure shows a number of correct 146-byte control transfers. The first part is the SETUP phase. The Trezor has ACK'd the SETUP packet, 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 me 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.

The final step of fine-tuning the fault to get a useful effect again is helped with our protocol analyzer. I physically moved the coil around over the surface, along with adjusting the glitch width and power level. It was possible—from the LCD screen—to visually see when the device entered an error handler


or seemed to continue unaffected. Finding a location that did not always enter an error is typically a useful starting point, and from there I searched through various parameters until a successful glitch occurred. Again, note that due to the deterministic nature of the glitch timing, you must be careful to search sufficiently long in possible candidate glitch settings.

PREVENTING THE ATTACK

While it's all good to cause the attack, how would you prevent against it? The first thing is to evaluate if your USB stack can be modified to prevent sending such large responses. If you never need to perform transfers of more than say 256 bytes, why not use an 8-bit number internally, or mask off the upper bits? Such a mask can be applied at multiple locations to complicate glitch attacks.

The second easy fix is to take advantage of memory protection, if your specific device supports it. This fault saw me slide from the USB descriptors in flash memory and read beyond them into sensitive metadata. But if we had bounded the sensitive metadata with invalid memory segments, our "slide" would have caused an exception due to the memory access error. When storing sensitive data in memory—either flash or SRAM—, bounding it with traps can be useful to catch any sort of attack that reads beyond an array. More generic countermeasures to fault attacks can also be applied, but I wanted to concentrate on specific countermeasures relevant to the memory ready attack shown here.

USE THE (MAGNETIC) FORCE

I hope you enjoyed this case study on electromagnetic fault injection. I've taken you through how EMFI could be used to attack a real product, with an exploit that has recently been disclosed to the Trezor team. Many other USB stacks use an almost identical code flow however, so I suspect you'll find this vulnerability could exist in your own system. Ultimately it depends on the use-case, but anything where sensitive data is stored in standard internal memory needs great care to keep that data inside your device. 

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

RESOURCES

Great Scott Gadgets | www.greatscottgadgets.com
 Microchip Technology | www.microchip.com
 NewAE Technology | www.newae.com
 STMicroelectronics | www.st.com
 Total Phase | www.totalphase.com
 Trezor | www.trezor.io
 Yepkit | www.yepkit.com