Breaking the Ledger Security Model

Mar 20, 2018

In this post, I'm going to discuss a vulnerability I discovered in Ledger hardware wallets. The vulnerability arose due to Ledger's use of a custom architecture to work around many of the limitations of their Secure Element.

An attacker can exploit this vulnerability to compromise the device before the user receives it, or to steal private keys from the device physically or, in some scenarios, remotely.

- Physical access before setup of the seed

  Also known as a "supply chain attack", this is the focus of this article. It does not require malware on the target computer, nor does it require the user to confirm any transactions. Despite claims otherwise, I have demonstrated this attack on a real Ledger Nano S. Furthermore, I sent the source code to Ledger a few months ago, so they could reproduce it.

  As you can tell from the video above, it is trivial to perform a supply chain attack that modifies the generated recovery seed. Since all private keys are derived from the recovery seed, the attacker could steal any funds loaded onto the device.

- Physical access after setup

  This is commonly known as an "Evil Maid attack". This attack would allow you to extract the PIN, recovery seed and any BIP-39 passphrases used, provided the device is used at least once after you attack it.

  As before, this does not require malware on the computer, nor does it require the user to confirm any transactions. It simply requires an attacker to install a custom MCU firmware that can exfiltrate the private keys without the user's knowledge, next time they use it.

- Malware (with a hint of social engineering)

  This attack would require the user to update the MCU firmware on an infected computer. This could be achieved by displaying an error message that asks the user to reconnect the device with the left button held down (to enter the MCU bootloader). Then the malware can update the MCU with malicious code, **allowing the malware to take control**

> **of the trusted display and confirmation buttons** on the device.
>
> This attack becomes incredibly lucrative if used when a legitimate firmware update is released, as was the case two weeks ago.

Proof of Concept

If you want to miss out on the fun of building an exploit yourself, you can find my proof-of-concept on GitHub.

If you follow the instructions there and install it on a Ledger Nano S running firmware 1.3.1 or below, you will be able to reenact the attack in the video above. However, because this is for educational purposes only, I have deliberately made the attack slightly less reliable.

A Note on Responsible Disclosure

Before I get to the details of the vulnerability, I would like to make it clear that **I have not been paid a bounty by Ledger** because their responsible disclosure agreement would have prevented me from publishing this technical report.

I chose to publish this report in lieu of receiving a bounty from Ledger, mainly because Eric Larchevêque, Ledger's CEO, made some comments on Reddit which were fraught with technical inaccuracy. As a result of this I became concerned that this vulnerability would not be properly explained to customers.

I discuss my interactions with Ledger at the end of the article.

Background on Hardware Wallets

Cryptocurrencies, such as Bitcoin, use public key cryptography to protect funds. You can only spend the funds if you have the private key.

This creates an issue for the user as to how they should secure their private key. Humans are notoriously terrible at securing secrets and devices; even security experts are not infallible.

To solve this problem, a class of devices called "hardware wallets" have been invented. As the name suggests, these are hardware devices that store users' private keys to protect against malware. Many of these devices connect to a PC via a USB port, but do not reveal the private keys to the PC, much like a hardware security module (HSM).

However, acquiring the private keys is not the only way an attacker can steal your beloved Bitcoin. An attacker who compromises such a device could simply change the recipient of the transaction and the amount being spent! If this is done surreptitiously, many users will be unaware of this attack until it's far too late to recover the funds.

Therefore, any useable hardware wallet really needs the following features, which differentiate it from a dumb HSM

- A trusted display for visual verification of the transaction information
- On-device buttons, in order to confirm or deny signing transactions

Hardware wallets need to protect against a wide variety of attacks, including:

- Remote attacks (when an attacker can steal your private keys through malware on your computer)
- Supply chain attacks (when an attacker can modify the device to do Bad Things™ before you receive it)
- Unauthorized physical access (when an attacker can compromise the device if they obtain physical access)

We can further divide the last attack vector into two types: theft and "Evil Maid attacks". If an attacker can steal the device, they have a longer duration of time to perform an attack, and possibly access to expensive lab equipment. However, they may be thwarted by you realizing your device is missing, and moving your funds to new private keys.

Security features, such as duress passphrases which aren't stored on the device, can prevent the attacker from stealing your funds in this scenario because the device simply does not contain the information necessary to recover the private keys.

On the other hand, with an "Evil Maid attack", the attacker might have a limited time to perform the attack, and won't have an expensive lab at their disposal. These attacks can be far more dangerous due to the wide variety of scenarios they can be employed in:

- As the name suggests, an "evil maid" could compromise your device while they clean your hotel room
- Your device could be taken from you for a short time as you pass through airport security
- You might entrust your device to a relative or lawyer

In this disclosure, we will focus primarily on the case of supply chain attacks. That is: **whether or not you can trust your hardware wallet when you purchase it from a reseller or third party.** But, as I explain briefly at the beginning of this article, the methods described here can be applied to the other two attack vectors.

Breaking The Architecture

In September 2014, Ledger released the HW.1. This wallet was based on the ST23YT66, a smartcard with USB support. Unfortunately, this design had severe limitations: there was no trusted display or buttons. This made the wallet dangerous to use.

Fast forward to July 2016: Ledger announced a new device called the Nano S. Based on the ST31H320 Secure Element, the new product included confirmation buttons and a trusted display, along with a USB connection.

In November 2017, I decided to take a close look at the security of the Nano S.

While I didn't get time to take a look at the newer Ledger Blue, it functions identically to the Nano S. **At the time of writing, no firmware update has been released to fix the vulnerability in the Ledger Blue**.

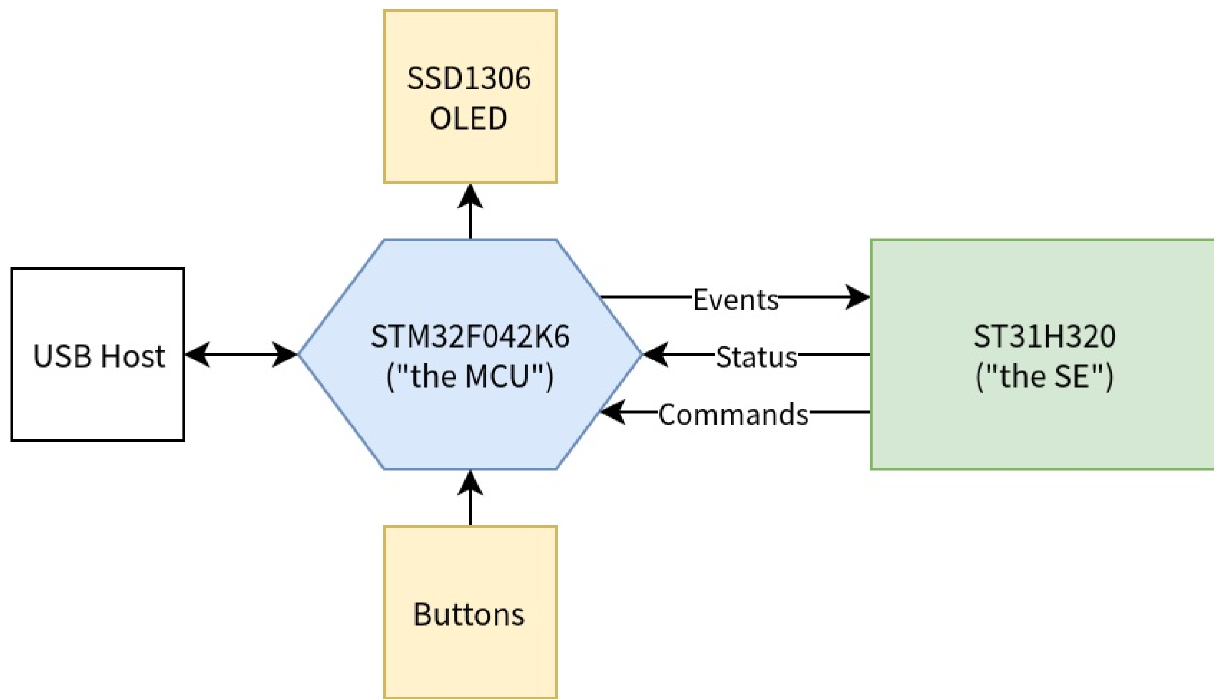
Dual-Chip Architecture

While there is no public datasheet available for the ST31H320, a quick look at the data brief shows that this Secure Element does not support displays! In fact, it doesn't even support USB. The only interface it supports is a low-throughput UART.

"What sort of witchcraft is this!?", I hear you cry.

As it happens, Ledger developed a new architecture to deal with this issue. The Nano S adds a second, non-secure microcontroller (STM32F042K6) which acts as a proxy for the Secure Element. This processor drives the display, buttons, and USB interface. It interfaces with the Secure Element, which stores the actual private keys.

From this point onwards, we'll refer to the ST31 Secure Element as the SE, and the STM32 microcontroller as the MCU. A diagram of the architecture looks like this:



TL;DR: The SE can only communicate directly with the MCU. However, the MCU can communicate with peripherals, on behalf of the SE.

An important feature of the Secure Element is that we can perform cryptographic attestation to determine that it is running genuine Ledger firmware. This is actually a selling point of the Ledger design: in fact, Ledger argues that this security feature is so powerful that Ledger wallets do *not* require tamper-resistant packaging (archive.is / archive.org), as described in the leaflet shipped with all devices.

Ledger's CTO even goes as far as to tell users that it is completely safe to purchase from eBay (archive.is / archive.org).

This brings us to the key problem. While the software on the SE can be attested to, the MCU is a non-secure chip and (as we show below) its firmware can be replaced by an attacker.

And herein lies the problem: to achieve Ledger's security guarantees, the chain of trust must be anchored in the SE. This means that the SE needs to verify the firmware on the MCU.

Hardware Tampering

While I will focus on software tampering in this article, it's important to note that, in the absence of a software vulnerability, you could still compromise the device by tampering with hardware.

It is incredibly important to note that, for these devices to be secure at all, you **must completely verify the physical hardware**.

Since neither the packaging nor the actual device are tamper-evident, it is trivial for an attacker to modify the device. I cannot repeat this enough: if you do not verify the physical hardware, **it is game over**.

Ledger provides instructions to do this, but I will note two issues with them.

1. The pictures are of varying quality. Ledger needs to provide high resolution images that display every component clearly.
2. The reverse of the device is not displayed at all!

> It is essential that you verify the back of the device, especially since this is where the JTAG header (a debugging interface) for the MCU resides.

Even if these two issues are resolved, I would question how expensive it is to have one of the MCUs with additional flash memory, but identical pinout, to be re-labelled as an STM32F042K6.

Nevertheless, while it is important to touch on this topic, hardware tampering is **not required** for the attack I will describe in this article.

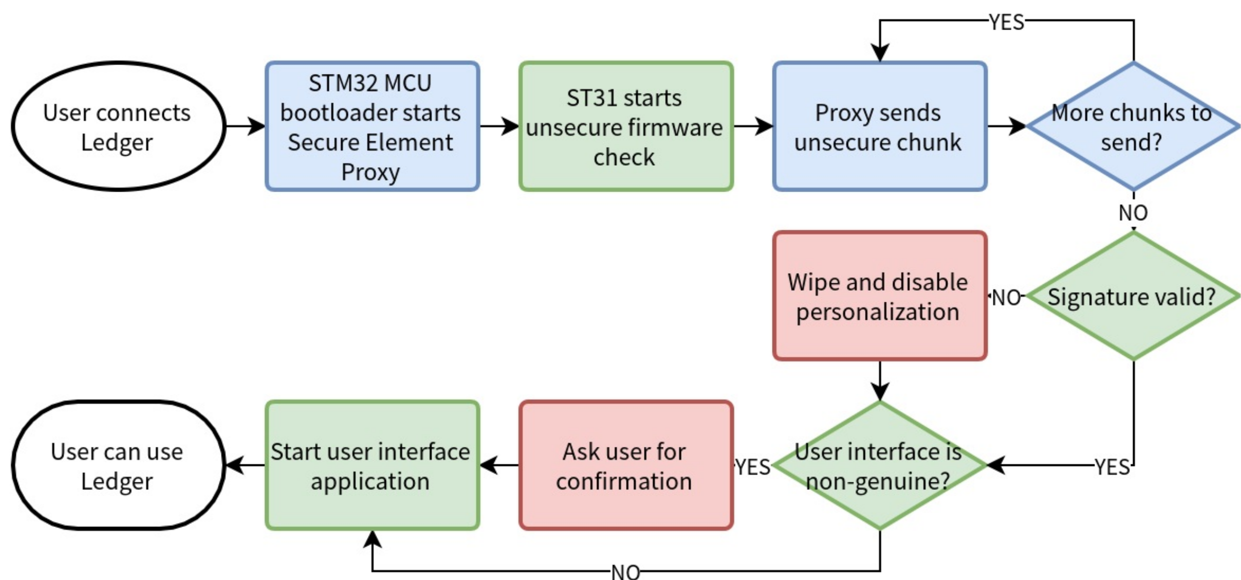
Verifying MCU Firmware

Let's assume that you have meticulously checked the hardware and it is definitely unmodified. What happens if the attacker simply changes the MCU's firmware?

Ledger considered this attack and, to prevent this, the MCU firmware is verified by the SE.

But it turns out that verifying the firmware on a non-secure processor is not so simple. The SE is nothing more than a glorified smart card, which means that the only method of communication with the MCU is via a low-throughput UART. With no direct access to the RAM or flash on the MCU, how can the SE verify its firmware?

Ledger's approach was for the SE to ask the MCU to pass over the entire contents of its flash memory, as detailed below.



At first glance this seems problematic. Basically, we are asking a (possibly compromised) MCU to prove that it's running the official Ledger firmware. But if the MCU is compromised, what stops it from sending over different code – code that it's not actually running? This is the challenge that Ledger attempted to tackle.

The theory adopted by Ledger is based on the fact that the MCU has a relatively limited amount of flash. To run malicious firmware, an attacker would *also* need to store the official Ledger firmware, so that it can satisfy the SE. Thus Ledger's approach was to make this difficult given the amount of flash available.

Specifically, by verifying the entire flash (and filling empty areas with random data), Ledger attempted to make it difficult to store malicious code on the MCU and also pass the MCU verification.

This is a remarkable idea, and perhaps it's possible to get it right. However, I was completely unconvinced by this solution.

Mode of Attack

While there are a few obvious methods to attack this design, such as supplying the malicious code via USB from an attached PC, it's much more fun to attempt a self-contained exploit (such as one that could be employed in a supply chain attack).

The method I chose was to "compress" the code. To use a compression algorithm such as DEFLATE or LZMA would be impossible due to the trade-offs between execution time, memory usage and code size. A user might notice if it took twenty seconds to start up their wallet!

Not to mention, while there were promising results compressing the entire flash, that was not the case for only the MCU firmware – and I did not want to replace the MCU bootloader, which is also present in flash. This is because there are two methods to install new firmware on the device:

1. Using the JTAG, a debugging interface used by embedded firmware developers to, amongst other things, upload new firmware.
2. Using the bootloader, which is the method used by Ledger users to install firmware updates. You can find the Python tool provided by Ledger to do this on GitHub.

   I was using this method because I don't enjoy soldering things. If I made a mistake while flashing the new bootloader, this method would stop working and the device would be bricked unless I used the JTAG interface.

Therefore, replacing the bootloader isn't an option and we have to rule out compression.

But there's another approach. When you compile a C program, the toolchain (the suite of software that compiles programs) will perform a number of magic tricks to make everything work. For example, many processors don't have instructions to divide very large numbers. The compiler works around this by inserting a software implementation of the division operation. Another example is when you declare initial values for variables defined in functions. When the function is called, the compiler will insert extra code at the beginning to copy this data onto the stack.

The extra functions the compiler inserts to perform these tasks are called "compiler intrinsics". Since the MCU has both a bootloader and firmware, and these are completely separate programs, these intrinsics will appear twice on the flash (once in each program).

The upshot of this is that we can insert our malicious routines in place of one *redundant* copy of the compiler intriniscs routines (specifically, the copy in the firmware). This leaves us with an intact copy of that code in the bootloader.

Because the intrinsic in the bootloader is identical to that in the firmware, when the SE asks the MCU for its flash contents, we can "piece together" a correct image by snipping out the malicious code and instead sending it the code from the bootloader. When the firmware needs to use the intrinsic, we can jump to the intrinsic in the bootloader instead.

If you're playing along at home, after building the bootloader and firmware from source code, you can use this command to find symbols to target

```
nm -S --size-sort -t d bin/token | grep -i " t "
```

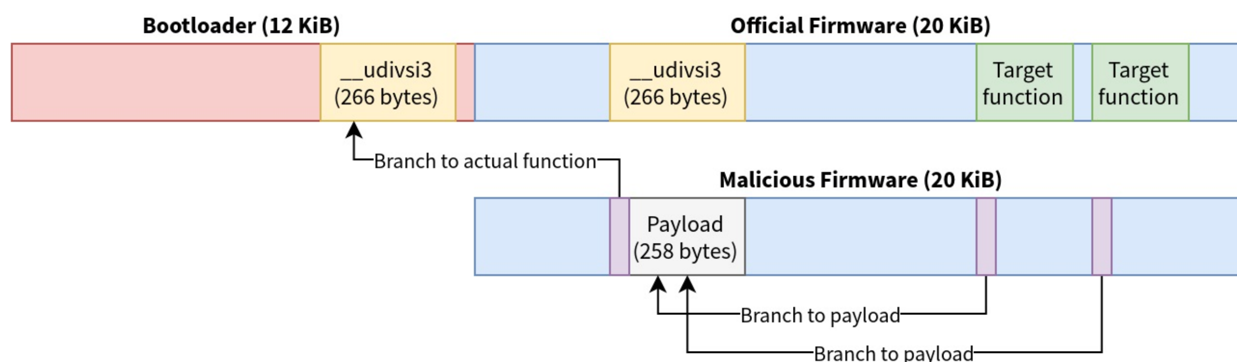
This command gave me a few interesting symbols that were identical in both the bootloader and firmware. No surprise, all three are compiler intrinsics.

```
134228637 00000124 T memcpy
134228761 00000140 T memset
134228357 00000266 T __udivsi3
```

To actually use the malicious code we have hidden, we will have to hook other functions. We do this by inserting branches to our payload in the functions we want to target. We need to hook the function that sends the flash contents to the SE, in order to send the bootloader function instead of our malicious code.

I also chose to hook the function that draws to the screen. This allows us to do a variety of fun and exciting tricks. Anything from changing displayed Bitcoin addresses and keylogging PINs to, as I will explain shortly, backdooring the private key generation is fair game!

With these two hooks and `__udivsi3` as our attack vector, our exploit looks a bit like this.



This approach frees up an incredible 258 bytes of payload! Well, we're definitely going to have to optimize for size, even if we throw `memcpy` and `memset` into the mix.

Making an Exploit

Our payload needs two components:

1. Code to modify the flash contents being sent to the SE, to trick the verification procedure
2. An attack such as a keylogger or key generation backdoor

I don't know about you, but backdoors seem like more fun to me.

Our exploit doesn't allow us to compromise the SE, so how can we add a backdoor?

Ledger's SE firmware has a user interface application which is in charge of the dashboard (and Settings). However, it is also used for the onboarding process (where the recovery seed is generated).

If we can modify the user interface, we can change the recovery seed that is generated during the onboarding process. This is quite easy since the user interface is open source and Ledger allows you (by design!) to install a modified UX application.

Under normal circumstances, the device would display a warning that the "User interface is not genuine", which would be a red flag for any attentive user.

But recall that I promised that I would explain how controlling the display can backdoor the key generation? The reason this attack works is that we can simply hide the non-genuine UX warning.

For this demonstration, we're not going to do anything sophisticated that a real attacker would do, such as generating a random-looking, yet entirely predictable, recovery seed.

We're going to do something much more obvious.

```
diff --git a/src/bolos_ux_onboarding_3_new.c b/src/bolos_ux_onboarding_3_new.c
index ce1849c..b950ae7 100644
--- a/src/bolos_ux_onboarding_3_new.c
+++ b/src/bolos_ux_onboarding_3_new.c
@@ -395,7 +395,7 @@ void screen_onboarding_3_new_init(void) {
```

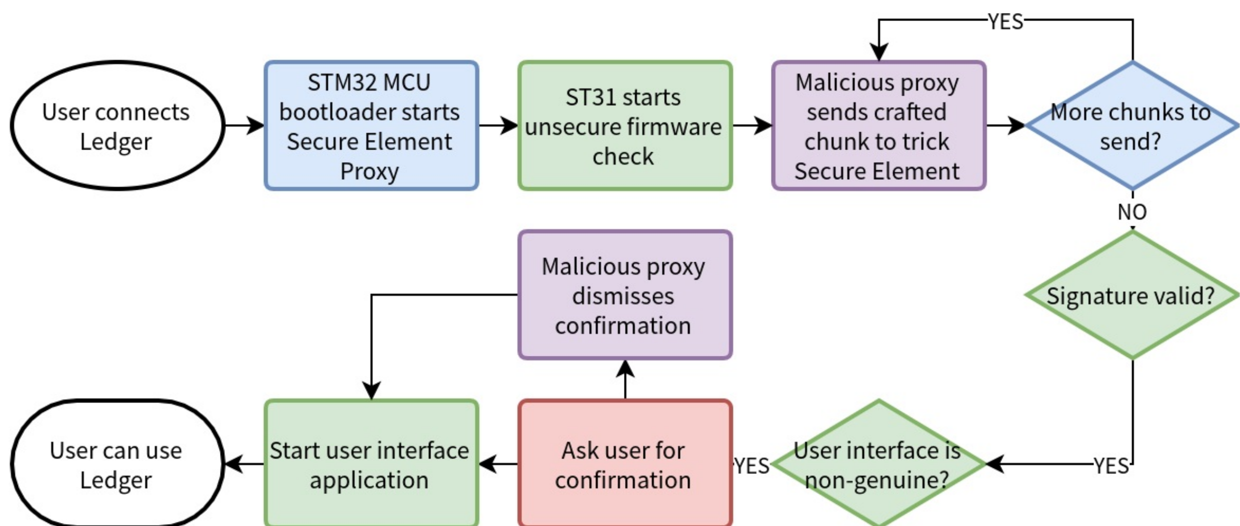
```
#else

    G_bolos_ux_context.onboarding_kind = BOLOS_UX_ONBOARDING_NEW_24;
-   cx_rng((unsigned char *)G_bolos_ux_context.string_buffer, 32);
+   os_memset(G_bolos_ux_context.string_buffer, 0, 32);
    G_bolos_ux_context.words_buffer_length = bolos_ux_mnemonic_from_data(
        (unsigned char *)G_bolos_ux_context.string_buffer, 32,
        (unsigned char *)G_bolos_ux_context.words_buffer,
```

If you're well-versed in C, you'll note that I'm replacing a syscall to the random number generator with a function call that sets all the entropy to zero. As you can see in the video at the start, it generates a recovery seed where the first 23 words are `abandon` (the last word is different because it is a checksum).

Since the private keys are derived from the recovery seed, if you control the recovery seed, you control all the Bitcoin addresses generated by the device.

If we put it all together, we get the following attack which I think is really neat.



Of course, since the SE believes the MCU is running genuine firmware, attestation still succeeds. And, as I mentioned earlier, no hardware tampering was required, which defeats Ledger's security integrity verification.

```
Terminal
File  Edit  View  Search  Terminal  Help
~ ⚡ python2 -m ledgerblue.checkGenuine --targetId 0x31100002
Product is genuine
~ ⚡
```

Since the attacker controls the trusted display and hardware buttons, it is astonishingly difficult to detect and remove a well-written exploit from the device.

Fixing the Attack

The problem with an architectural vulnerability like this is that it is challenging to fix without changing the architecture.

Ledger has employed multiple mitigations to try and prevent an attacker from exploiting this vulnerability.

First of all, the MCU firmware has been optimized and rearranged. Specifically, the firmware calls into functions in the bootloader instead of duplicating the functions. While this prevents this particular mode of attack, it's important to be aware that there are other, more "creative" methods of attack that I know of, and probably some that I don't know of.

Secondly, the SE now times the MCU when it asks it to send the flash contents. This is designed to prevent the use of compression algorithms. It is also supposed to prevent code being supplied by the computer over USB. I'm not sure how well it succeeds in doing the latter, due to the fact that the code can be kept in RAM.

However, it's of note that the SE runs at up to 28 MHz yet the "adversary" (the MCU) runs at up to 80 MHz! This throws into question whether a slower chip can accurately time a faster chip to prevent it from doing extra things, especially given the slow UART communication.

Ledger refused to send me a release candidate, so I haven't had an opportunity to verify how well these mitigations resolve the issue. But these raise an important question.

Is it truly possible to use a combination of timing and "difficult to compress" firmware to achieve security in this model?

Building secure systems using this model seems like an incredibly exciting research proposition and I think it's interesting to see companies like Ledger pushing the envelope on this.

Interaction with Ledger

Prior to the scheduled disclosure of this vulnerability, I had some interactions with the CEO of Ledger. You can find an archived copy of his main comment on archive.is and archive.org, in case it disappears for any reason.

In these comments, the CEO disputes that these attacks are critical. Some of Ledger's comments are subjective, and others are more factual. Below I discuss some of these comments.

The first claim I would like to address is that the vulnerability requires a set of incredibly unlikely conditions.

> *The vulnerability reported by Saleem requires physical access to the device BEFORE setup of the seed, installing a custom version of the MCU firmware, installing a malware on the target's computer and have him confirm a very specific transaction.*

I am puzzled as to where this claim could have originated from. From later contact with Ledger, I was informed that the CEO had not at all been briefed on the security vulnerability when they made these comments on Reddit.

As I stated at the beginning of the article, there are three methods to exploit this vulnerability, none of which require conditions as unlikely as those.

The malware attack vector I mentioned earlier leads nicely onto the next issue I have with Larchevêque's comment.

> *Saleem got visibly upset when we didn't communicate as "critical security update" and decided to share his opinion on the subject.*

When you fix a critical security issue, you can take one of two routes.

- Completely conceal the security fix

  This is an effective method to avoid drawing the attention of black hats (if your product is completely closed source, which is the case for Ledger).

  This has the downside that most users will avoid updating, especially if the process is very painful to do (as it was in this case).

- Alert users of a critical security issue and force an update

  This is commonly used for open source products or when the vendor suspects a security vulnerability is being used in the wild.

  However, this has the downside that it alerts black hats of the presence of a vulnerability. Therefore, it is essential that users update immediately to gain the "first mover" advantage over a potential attacker.

Ledger chose a flawed method, which takes the worst aspects of both of these approaches. By drawing attention to the security fixes in their firmware update, while not alerting users to update, you lose the "first mover" advantage.

This gives black hats sufficient time to determine how to exploit the vulnerability, putting all users at risk of the malware attack vector.

My concerns were proven correct, as I was contacted by multiple independent white hats who had determined the issue purely from Ledger's firmware update instructions.

Disclosure Timeline

- 11 Nov 2017: Officially reported vulnerability to Nicolas Bacca, Ledger CTO. Vulnerability determined to be implausible.
- 14 Nov 2017: Demonstrated practical supply chain attack with modified MCU firmware and user interface. Sent source code to Bacca.
- 30 Dec 2017: Bricked the Ledger Nano S by downgrading the firmware to an unsupported version. Press F to pay respects.
- 06 Mar 2018: Ledger released firmware update for Ledger Nano S.
- 20 Mar 2018: Write-up and proof-of-concept code released.
- **Firmware update for Ledger Blue unreleased at time of writing.**

Acknowledgements

I would like to thank Josh Harvey for providing me with a Ledger Nano S, so I could turn my theoretical attack into a practical exploit.

I would also like to thank Matthew Green, Kenn White and Josh Harvey for your invaluable help in editing this article.