KeepKey Stack Buffer Overflow Vulnerability (CVE-2021-31616)

Apr 30, 2021 • Christian Reitter
ID: CVE-2021-31616

Related articles: KeepKey Supervisor Vulnerabilities (CVE-2022-30330) • KeepKey Message State Handling Vulnerability (VULN-22003) • Glitching over KeepKey Firmware Protections (VULN-21020)

I have recently discovered a serious vulnerability in the KeepKey hardware wallet.
Through a stack buffer overflow, remote or local attackers can execute code on the device and perform actions such as stealing the wallet keys from within a malicious website. The vulnerability was introduced with firmware ***v7.0.3*** and patched with ***v7.1.0*** after my disclosure.

Contents

Consulting

I'm a freelance Security Consultant and currently available for new projects. If you are looking for assistance to secure your projects or organization, contact me.

Introduction

As with many of my previous blog articles, this is going to be a technical deep-dive into a complex security bug. Correspondingly, the article is written for technical readers with a background in the area of IT security. Go to the general summary of the issue if you are interested in the less technical version.

The ShapeShift KeepKey is a cryptocurrency hardware wallet with an open source firmware. I have done a lot of independent security research on over the last years, mainly through extensive automated testing with a custom fuzzing setup in combination with manual auditing. See my other blog articles for more context on this topic and discovered issues.

ShapeShift has recently integrated logic for the *THORChain* network into their exchange, which included new code changes and logic for their KeepKey wallet firmware. More specifically, the new logic is located in the Ethereum handling. As it turns out, it contains a major flaw:

See the following sections for details on this attack.

The Vulnerability

For the purposes of this bug, it is sufficient to know that the *THORChain* usage is related to a special subset of Ethereum transactions. The issue is in the new Ethereum signing code that was introduced to handle this new logic.

As with regular Ethereum transactions on the KeepKey, a `MessageType_EthereumSignTx` protobuf message from the host computer triggers some finite state machine behavior, which includes the `ethereum_signing_init()` function:

```
void ethereum_signing_init(EthereumSignTx *msg, const HDNode *node,
                           bool needs_confirm) {
```

ethereum.c

Initially, the code checks a few basic properties of the received request message, for example via `ethereum_signing_check()`. It rejects the most problematic messages that satisfy error conditions such as `if (msg->data_length > 16000000) {`.
However, these functions do not perform any in-depth validity checks on the message properties, and many of the message fields cannot be checked during the initial message processing due to how the protocol is designed. This is relevant for later.

Ethereum transmission requests that satisfy the `ethereum_isThorchainSwap(msg)` check will be processed with the newly introduced `ethereum_extractThorchainSwapData()` function early in the message handling:

```
  // Detect THORChain swap
  if (ethereum_isThorchainSwap(msg)) {
    if (token == NULL && data_total > 0 && data_needs_confirm) {
      char swap_data[256] = {'\0'};
      uint8_t swap_data_len = ethereum_extractThorchainSwapData(msg, swap_data);
```

ethereum.c

This is the problematic path that can be attacked.

As it can be seen in the code excerpt below, the first parameter for the `ethereum_extractThorchainSwapData(const EthereumSignTx *msg, char *buffer)` function call is the received message `msg` and the second parameter is the local stack buffer `swap_data[256]` from the `ethereum_signing_init()` function context.

Quick challenge for the more experienced security people among the readers: consider the following code snippet for a minute or two.
Can you spot the bug and what it does?

```
uint8_t ethereum_extractThorchainSwapData(const EthereumSignTx *msg,
                                          char *buffer) {
  // Swap data begins 164 chars into data buffer:
  // offset = deposit function hash + address + address + uint256
  uint16_t offset = 4 + (5 * 32);
  int16_t len = msg->data_length - offset;
  if (msg->has_data_length && len > 0) {
    memcpy(buffer, msg->data_initial_chunk.bytes + offset, len);
    // String length must be < 255 characters
    return len < 256 ? (uint8_t)len : 0;
  }
  return 0;
}
```

ethereum.c

To answer this, the essential weakness here is related to the length field, more specifically, the attacker-controlled `msg->data_length` field. It is incorrectly treated as a **trusted input** and allowed to influence low-level memory operations, similar to multiple previous security issues that I've discovered in other products through fuzzing.
Fuzzers have a good nose for reaching branches in the code while at the same time having basically no understanding of how the branch is *supposed* to be reached. They're ideal to trigger this type of edge case with an unusual length value. Sanitizers like AddressSanitizer can then detect the memory corruption that follows, revealing the issue out of millions of other unproblematic fuzzer runs.

In this specific case, the `msg->data_length` field is basically an arbitrary `uint32_t` integer that can be set by the attacker. It is only slightly constrained to a value of `< 16M` through the previously mentioned check in the parent function, which is of no practical relevance. More essentially, this length field is not limited to the maximum size of the `swap_data` (256 byte) or even `msg->data_initial_chunk` (1024 byte) buffers.
The `msg->data_length` represents the data size that is announced to the wallet as the *total length* of the transaction data that is supposed to follow in future messages. `msg->data_initial_chunk` is just the initial data section if the total expected length is bigger than the first chunk field can hold. For practical attacks, the `<= 1024` of this initial chunk is sufficient, as we'll see in later descriptions, so there is no need to actually send followup messages with additional data.

In the code snippet above, the `int16_t len` value is directly computed from the foreign `msg->data_length` value minus the fixed `offset` of `164`. There is a `len > 0` check to make sure the result is not used when it is zero or negative, which is meaningful, but fails to catch the even more important case for `len` values that are bigger than expected.
For some reason, the developers were anticipating that `len >= 256` values end up in this code branch since they included a special check to set the returned length to `0` in this case. However, there is no check to prevent the problematic memory corruption behavior that *also* happens with `len > 256`!
At those lengths, the `memcpy()` call results in an out of bounds write on the stack behind `swap_data`, which is located in the stack frame of the *caller* function.

In practical terms, this means that an attacker can manipulate the length parameter of the `memcpy()` arbitrarily through `msg->data_length` to values between 1 and 32767. The target buffer is always of fixed size 256 via `char swap_data[256]`. Since the destination buffer is nulled before usage, there is no information disclosure when performing a partial overwrite through `memcpy()` with for smaller lengths than usual. Therefore lengths `>256` are interesting for an attack.

To make matters worse, the main source of the copied data is the large byte field `msg->data_initial_chunk.bytes`, which is a fully attacker controllable byte array in the same message that triggers the attack. Up to `1024 - offset` = 860 bytes of data can be copied directly from there. For even bigger `memcpy()` operations, the source values are fetched as out of bounds reads from other fields of the same `msg` message, and so they are also controllable by the attacker to a large degree.

To summarize, one large Ethereum-related message is sufficient to trigger a large and very controllable out of bounds write on the stack.

Exploiting the Stack Buffer Overflow

The technical aspects will be presented in multiple additional blog articles, since the analysis is complex and involves a second coordinated disclosure.

- Part I of the analysis can be found here.
- Disclosure article on the faulty stack smashing protection.

The Fix

For obvious reasons, I recommended stronger limits for the length parameter of `memcpy()` as a first step to the vendor when reporting the issue.

This was done through a patch in firmware `v7.1.0`, which is fairly compact:

```
@@ -93,10 +93,10 @@ uint8_t ethereum_extractThorchainData(const EthereumSignTx *msg,
   // offset = deposit function hash + address + address + uint256
   uint16_t offset = 4 + (5 * 32);
   int16_t len = msg->data_length - offset;
-  if (msg->has_data_length && len > 0) {
+  if (msg->has_data_length && len > 0 && len < 256) {
     memcpy(buffer, msg->data_initial_chunk.bytes + offset, len);
     // String length must be < 255 characters
-    return len < 256 ? (uint8_t)len : 0;
+    return (uint8_t)len;
   }
```

```
    return 0;
}
```

ethereum.c

Going forward, I recommend to make the magic `256` limit value more explicit in that data handling, either by taking it as a length parameter in the function or by adding documentation to the function body. Additionally, the fixed `offset` could be an explicit constant and the integer overflow handling from the larger `msg->data_length` could be better, but neither of them is a security problem as of now.

Attack Scenario and Security Implications

The target device needs to be in an unlocked state to be vulnerable:

- on devices without a PIN, this is automatically the case
- otherwise the PIN has to be entered once (and is then cached)
- the correct passphrase is not necessary to trigger the attack even if enabled (but is also affected if cached)

Details of the unlock state of the KeepKey can be checked repeatedly by an attacker through a `GetFeatures` protobuf message. Once the reported conditions on the device are good, an attacker can then send the problematic USB packets to the device to trigger the exploit.

The attack has been confirmed to work

1. through malicious JavaScript and WebUSB in Chrome/Chromium
   - the user needs to accept the generic WebUSB browser dialog to connect the device
2. through programs with local user privileges
   - no user interaction required
   - depends on the necessary drivers and configuration

Observed Impact

I have shown that it is possible to get code execution on the device by smashing the stack and then manipulating the instruction flow.

Through Return Oriented Programming (ROP), an attacker can very deliberately trigger a number of code snippets from existing functions and combine them with unusual parameters to basically do anything they want. For example, one particularly problematic action is making the device send USB protobuf messages filled with *secret stack memory* from the `CONFIDENTIAL` memory area. Based on this attack approach, the decrypted BIP39 mnemonic and cached passphrase (if available) can be copied from the device to the attacker.

Fundamentally, the main security guarantees of a hardware wallet and U2F 2nd factor device are compromised once the secret keys are known. This attack breaks most aspects of the KeepKey security model.

Other impacts that I've been able to trigger include setting of new passphrases, PINs or BIP39 seed erasure. This could be problematic for users who do not have a paper wallet backup at hand. However, in most cases it would be quicker for an attacker to steal the funds instead, unless they have a reason to intentionally trigger device problems, e.g., as a coverup for the theft or some other specific scenario.

Interestingly, it is also possible to artificially trigger my previously discovered CVE-2019-18672 issue through this bug, but then again - that's less interesting to the attacker than simply getting the valid U2F device private key itself.

One of the technically interesting questions here is whether this attack can result in persistent firmware modifications. In other words, can you start trusting the device again after you've wiped it and installed a new firmware, or could it still contain malicious code or settings?
Although it is possible for attackers to change some regions of the device flash, my current understanding is that this does not allow persistent modifications to the firmware code without also alerting the user on the next device boot. The existing security configurations such as the internally dropped microcontroller permission levels, memory protection unit settings and limitations on flash writes through the supervisor abstraction *should* prevent an overwrite of the bootloader. If this is satisfied, the still-genuine bootloader will detect that the modified firmware is no longer correctly signed as part of the typical boot check, leading to a "Unofficial Firmware" security warning and mandatory user interaction. Which isn't perfect, but better than silently running with a modified bootloader/firmware combination (which could falsely report that it is correctly updated to a newer version to evade detection, for example). Note that the error also depends on the bootloader version, of which multiple are in use (at least `bl1.1.0` and `bl2.0.0` are relevant).

So, after upgrading to a new and signed firmware version and wiping the previous settings, the device and firmware *should* be trustable again. I have not looked at this scenario long enough to vouch for it, though.

Formal Scoring

Description	CVSS 3.1	Score
WebUSB-based attack with connect dialog interaction, full impact	AV:N/AC:L/PR:L/UI:R/S:U/C:H/I:H/A:H	8.0 (High)

Notes:

- Having both `PR:L` and `UI:R` is probably over-accounting for the WebUSB connection dialog.

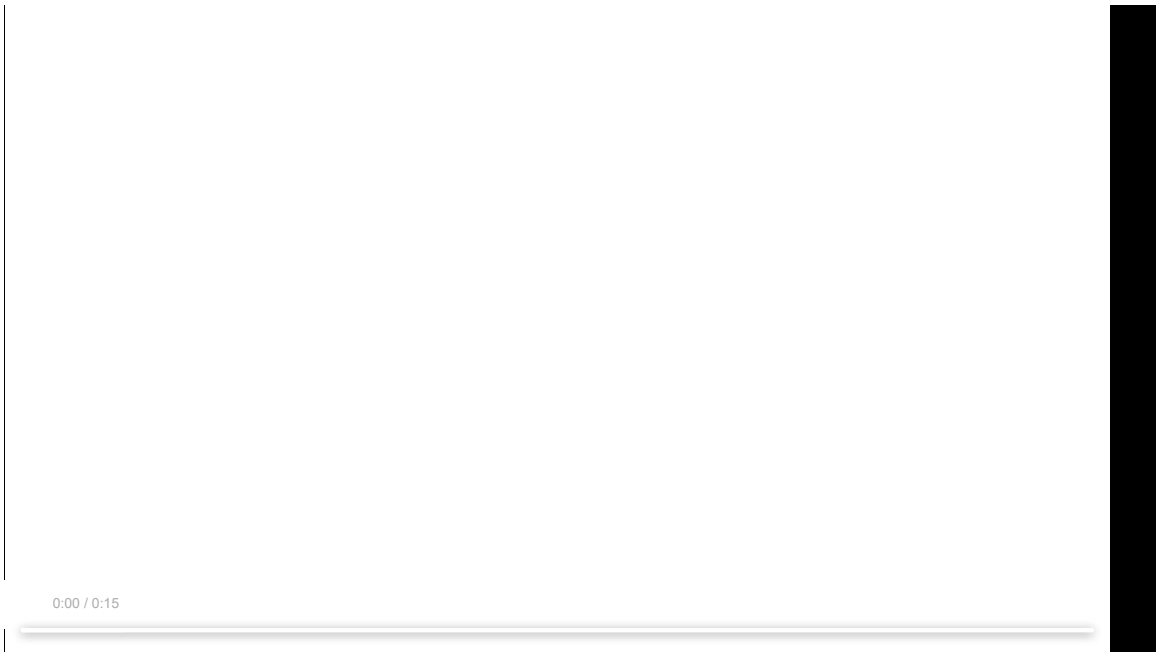
Proof of Concept (PoC)

I've sent four different PoCs to the vendor:

1. Denial of service via crash (the bug is reachable)
2. Custom display content (reliable code execution)
3. Copying BIP39 seed off the device (breaking confidentiality)
4. Copying cached passphrase off the device (breaking confidentiality)

They are not published at the moment. I'm planning to release more technical information on the attack soon.

Here is how the modified No.2 PoC works on the device:



Details: device without PIN for demonstration purposes, USB attack at 0:10 triggers an "impossible" layout_warning_static() call

Version Overview

Version	Status	Description
v6.x.x and earlier	not vulnerable ✓	Firmware versions do not contain the problematic code
v7.0.3	🔥 **firmware vulnerable** 🔥	Update **as soon as possible** and consider switching to a different BIP39 mnemonic.
v7.1.x	firmware not vulnerable	🔎 ⚠️ Remaining security concern if the device was used with firmware *v7.0.3* previously.

General Summary of the Issue

This publication is about a serious bug in the specific KeepKey firmware ***v7.0.3*** that was released in March 2021.
Devices with this version can be attacked. All other versions are OK.

Basically, an attacker can command the device to reveal or manipulate any wallet secrets that are known to the device at the time of the attack. They can change the PIN, erase the device or do other weird things. Depending on how the device is used and when the attack happens, the entered secret passphrase can be stolen as well. This means that all cryptocurrency funds for all coin variants "on" the wallet can be transferred out to the attacker right away (or months later!) since they can create an identical copy of the wallet.

The attack is possible as soon as the device is connected to a computer and unlocked. Unlocking is usually done by entering the PIN and special passphrase through a normal dialog on the computer, if either of those are used. Otherwise, the device is unlocked and vulnerable by default.

Attack scenarios:

1. If the attack is triggered by malicious software on the computer that the KeepKey is attached to, it is silent and without any user interaction.
2. If the attack happens through a malicious webpage, it requires a click on the normal browser confirmation dialog (for the device connection).

There is not a lot one can do to prevent the attack, and one cannot find out if any of the sneakier attack variants already happened.

As vulnerabilities go, this is pretty bad.

Some good news:

- I found the bug quickly after it got introduced.
- I went directly to the vendor with it while keeping it a secret (coordinated disclosure).
- The vendor realized that the issue was serious and quickly released a fix.

I obviously recommend upgrading from the problematic firmware to a patched version right away. Unfortunately, this might not be sufficient. If other people on the Internet have found the bug in the past weeks independently and realized its potential, they could have systematically attacked vulnerable KeepKey users. This is not that unlikely, since the v7.1.0 release changes showed where the bug was, for example. It is generally plausible that wallet keys have been copied by an attack "in the wild" somewhere, since it is proven that this is possible from the technical side.

If you 1. had the dangerous firmware version on a device with significant funds and 2. used it on any computer or website that you don't *really, really* trust, now would be a good time to check your funds and move them to another BIP39 seed and another passphrase.

Coordinated disclosure

I've now reported security issues for the KeepKey for four consecutive years, and this has been generally positive, but there were some ups and downs along the way. For example, there have been a few minor security reports to ShapeShift that dragged on through most of 2020 with months of delayed feedback and a lack of code changes. As someone with professional software developer experience, I understand that backlogs can be full and lower rated issues get passed over again and again as a consequence. However, from the researcher side, having to keep the ball rolling on unresolved topics is draining energy from other things. Running into a situation like that will naturally lower my interest in researching and reporting future security bugs for a given software, particularly in case of minor bugs.

Given this background, it was refreshing that the ShapeShift vulnerability handling processes have changed in 2021 and that the situation has significantly improved. It is not ideal yet, but it is definitely getting better.

I want to positively mention:

- their initial reaction time of **less than 24 hours** for human feedback on the issue
- the quick patch rollout of **less than 6 days** from disclosure to release
- from what I understand, ShapeShift has enabled urgent upgrade warnings for affected users on their platform
- the opportunity to do a call and discuss the issue in person (although late in the disclosure)

However, there are also several areas that can still be improved:

- publishing **timely security advisories** within a few days of public security patches
- informing the reporting researcher of relevant patches, releases and planned time frames of related topics
- reducing miscommunications within the team

Since I've noted missing security advisories during previous KeepKey disclosures, I recommend that they look into how to improve their advisory processes for future issues, so that customers can make better informed decisions when tasked with upgrades and warnings. Particularly with a complex and dangerous issue such as this one, having "Security fixes" in a changelog as the only public information is simply insufficient, even more so if some risk for users remains after the update, which is the case here in my opinion.

This has been a big and complex disclosure, but I'm definitely quite satisfied with the research results.
Stay tuned for more technical details on this topic.

Relevant product

Product	Source	Fixed Version	Patch	Publications	IDs
ShapeShift KeepKey	GitHub	fix: **v7.1.0** recommended: **v7.1.2**	patch	v7.1.0 changelog, ShapeShift advisory	CVE-2021-31616, VULN-21011

I'm not aware of other affected hardware wallets.

Detailed timeline

Date	Information
2021-03-24	Affected firmware version v7.0.3 is released
2021-04-08	I discover the vulnerability
2021-04-08	Confidential disclosure to ShapeShift
2021-04-08	ShapeShift acknowledges the issue and assigns VULN identifier
2021-04-12	ShapeShift releases patched firmware version 7.1.0 for download
2021-04-14	ShapeShift publishes the code for firmware version 7.1.0 on GitHub
2021-04-21	CVE request sent to MITRE
2021-04-23	MITRE assigns CVE-2021-31616
2021-04-26	Technical call with ShapeShift
2021-04-27	ShapeShift releases firmware version 7.1.2 on GitHub
2021-04-30	Publication of this blog article
2021-05-03	Corrections to the ShapeShift security advisory (see notes)

Notes:

- The ShapeShift security advisory originally referenced April 4th as the patch release date, which was a numerical error that has been corrected.
- My timeline uses the source code tags to determine the time of publication, but I'm told that the actual firmware availability was a bit earlier. This was added to the timeline, where known.

Credits

I want to give special thanks to Dr. Jochen Hoenicke, who has helped in later stages of the research.

A Note About Research Affiliation

I want to emphasize that this research was done on my own time and initiative. In particular, it was not sponsored by SatoshiLabs, for whom I do some paid freelance security research on the related Trezor project.

Bug bounty

ShapeShift has paid a bug bounty for this issue.

Christian Reitter

Information security and other interests.