



Kraken Identifies Critical Flaw in Trezor Hardware Wallets

BY KRAKENFX | JANUARY 31, 2020

Kraken Security Labs has devised a way to extract seeds from both cryptocurrency hardware wallets offered from industry leader Trezor, the Trezor One and Trezor Model T.

The attack requires just 15 minutes of physical access to the device. This is the first time that the detailed steps for a current attack against these devices has been disclosed.

Here's how we did it:

- This attack relies on voltage glitching to extract an encrypted seed. This initial research required some know-how and several hundred dollars of equipment, but we estimate that we (or criminals) could mass produce a consumer-friendly glitching device that could be sold for about \$75.
- We then crack the encrypted seed, which is protected by a 1-9 digit PIN, but is trivial to brute force.

Kraken Identifies Critical Flaw in Trezor Hardware Wa...



The attack takes advantage of inherent flaws within the microcontroller used in the Trezor wallets. This unfortunately means that it is difficult for the Trezor team to do anything about this vulnerability without a hardware redesign.

Until then, here is what you can do to protect yourself:

- **Do not allow anyone physical access to your Trezor wallet**
 - You could permanently lose your crypto
- **Enable Your BIP39 Passphrase [with the Trezor Client](#)**
 - This passphrase is a bit clunky to use in practice but is not stored on the device and therefore is a protection that prevents this attack.

This attack is very similar to [our previous research against the KeepKey wallet](#), which is expected because the KeepKey is a derivative and all devices rely on the same family of chips. [Trezor has known about these flaws](#) since designing the wallets.

Other teams, [like Ledger Donjon](#), have also performed variants of this attack, though the full details have not been made public until now.

These chips are not designed to store secrets and our research emphasizes that vendors like Trezor and KeepKey should not solely rely on them to secure your cryptocurrency.

We are fortunate to be working with the Trezor team to coordinate this disclosure and [you should also review their response](#). Pavol Rusnak, CTO of SatoshiLabs, adds “We are happy that Kraken

Security Labs are investing their resources in improving the security of the whole Bitcoin ecosystem. We cherish this kind of responsible disclosure and cooperation.”

At Kraken Security Labs, we try to discover attacks against the crypto community before the bad guys do. **We responsibly disclosed the full details of this attack to the Trezor team on October 30, 2019. We are going public with this vulnerability disclosure now so that the crypto community can protect themselves before a fix is released by the Trezor team.**

Technical Details

Extracting seeds from Trezor wallets is not new territory. Trezor has implemented significant mitigations against a variety of previous hardware attacks, including successful mitigations against the glitching attacks demonstrated during the [Wallet.Fail](#) talk at the [35th Chaos Communication Congress](#). This attack builds upon that research to bypass the mitigations.

Our attack begins by re-enabling the integrated bootloader of the processor using a fault-injection attack. This integrated bootloader has functionality to read-out the flash contents of the device, but verifies the protection-level of the chip while executing the command. By utilizing a second fault-injection attack it is possible to circumvent this check, and then the entire flash-contents of the device can be extracted 256 bytes at a time. By repeating the attack it is possible to extract all of the flash contents. Additionally, because the Trezor firmware utilizes an encrypted storage, we developed a script to crack the PIN of the dumped device, leading to a full compromise of the security of the Trezor wallets. The script was able to brute force any 4-digit pin in under 2 minutes. This attack demonstrates that the STM32-family of Cortex-M3/Cortex-M4 microcontrollers should not be used for storage of sensitive data such as cryptographic seeds even if these are stored in encrypted form.

The STM32F205 and STM32F427 are flash-based microcontrollers used in the Trezor One and the Trezor T, respectively. Many derivatives of the Trezor One, such as the Keepkey, also use the STM32F205. Both STM32F2 and STM32F4 are ARM Cortex-M3 microcontrollers of the STM32 family of ST Microelectronics. The STM32F2 and STM32F4 provides all the peripherals necessary for implementing the hardware wallet, including a PLL, as well as interfaces, such as USB. Most notably, the STM32F205 offers two common ARM programming interfaces: [JTAG](#) and ARM [SWD](#). In addition to these programming interfaces, the STM32F205 also offers an integrated bootloader that can be used to program the device using interfaces such as UART, USB and CAN.

Flash and SRAM Read Protection on the STM32

The STM32 family implements a security mechanism known as Read Protection or RDP. Because the only non-volatile storage on ARM Cortex-M devices is flash memory, the RDP value is stored in

a special page of flash memory that is otherwise inaccessible for writing from the application code. The RDP value is defined by the microcontroller configuration bits known as Option Bytes. There are three Option Byte values corresponding to the three RDP levels on STM32 devices.

RDP Level	Option Byte Value	Behavior
0	0xAA	Full access to SRAM and Flash
1	Any value except 0xAA and 0xCC	Read access to SRAM, no access to Flash
2	0xCC	No access to SRAM or Flash

Table 1: RDP Levels and the corresponding Option Byte Values on STM32-Family Devices.

Since the only non-volatile memory on STM32 microcontrollers is flash, this is also the only non-volatile storage for the cryptographic seeds and private keys. As a result, the flash must be protected from read out. Fortunately, the Trezor and all of its derivatives correctly utilize the RDP feature and are shipped with and/or set RDP to RDP Level 2 (see Table 1) on first boot. As a result, in practice, non-development firmware on user devices are always at RDP2 (RDP Level 2), which prevents an attacker from accessing SRAM or Flash. However, as demonstrated by the [Wallet.Fail](#) and [Chip.Fail](#) research, downgrading RDP2 to RDP1 can be reliably be performed at boot with voltage glitching. Once a device is at RDP1, its SRAM can be read out over the ARM SWD debugging protocol.

Because of the complex Power-On-Reset (POR) logic of the STM32, a normal assertion of reset, i.e. *soft reset* where the NRST line is held low for a short amount of time, does not result in a full power-on-reset and re-execution of the BootROM. This is also somewhat confirmed by the fact that a change in the security configuration, i.e. changing the Option Bytes to change the RDP Level, generally requires power cycling the chip. Conversely, this also means that once a chip is successfully glitched and a resulting downgrade of the security configuration has taken place, this security downgrade will remain in effect until the chip is power-cycled. This means that an attacker can repeatedly attempt to glitch the device, check whether the glitch was successful, while still executing Bootrom or very early into the application code without the application code loading. As a result there are no countermeasures that are effective against this class of attack because the attacker can assure the glitch succeeded before executing the application code. Once an attacker successfully glitches a device, the attacker simply performs a soft reset of the target, the system continues to run at RDP1, allowing the attacker to arbitrarily read the contents of SRAM memory at any given point in time. This is particularly problematic as many of the libraries implementing the cryptography necessary for signing cryptocurrency transactions rely on loading sensitive information into SRAM for computation. Additionally, the cryptographic seed may be loaded into SRAM during wallet derivation and the PIN of the user may be loaded for verification against user input. If the underlying firmware is verified or if a checksum is computed to check the integrity of the firmware, parts or all of this data may also be exposed to attacks [[O'Flynn Circuit Cellar](#)].

STM32 Boot process and Glitch Parameters

Much of the behavior of a microcontroller is defined by values it reads at power up. These include strapped pins that are read at boot (BOOT pins in the STM32 documentation) and the security configuration bits (Option Bytes in the STM32 documentation). Note, many of the following details have been determined through empirical reverse-engineering of the boot behavior of the STM32F2. Most Cortex-M microcontrollers contain ROMs that are executed at boot, commonly referred to as BootROMs. BootROMs are the first pieces of software executed by a chip and are responsible for loading important parameters, such as the security configuration of the chip. Subsequently, the user application or application code is executed. In the case of the hardware wallet, this is the actual firmware of the manufacturer. Note, because the glitching attack described in this work targets the BootROM code, it cannot be reliably mitigated by any countermeasures implemented in the vendor's firmware. A vulnerability in the firmware leads to an inherent hardware vulnerability that cannot be patched and requires the underlying hardware to be replaced completely with a new hardware revision.

Presumably because of the relative complexity of the STM32F2, the STM32 takes very long to boot, approximately 1.2ms – 1.8ms after power cycling the power supply to the chip. The boot time can be reliably measured in two ways: either measuring the power consumption of the device and measuring the amount of time that it takes for an initial rise in power consumption, for example with a [LSCM](#), or by observing the behavior of the reset (NRST/JTAG RST) line of the microcontroller. Within the first 100us – 200us, the BootROM of the chip is executed.

Re-enabling JTAG, SWD and the integrated BootROM Bootloader

During the STM32 startup, the integrated BootROM is executed before the application code (i.e., the wallet firmware) is executed. The BootROM implements several checks that in turn configure the device's security state. These include enabling/disabling the JTAG and SWD debugging interfaces, as well as the integrated serial BootROM bootloader. Because the STM32 family is flash-based, the only Non-Volatile Memory (NVM) available on these devices is flash [\[Obermaier\]](#). Since flash is the only NVM available on STM32s, the security configuration must be stored in flash as well.

There is a special dedicated region of flash for storing the security and device configuration, known as the Option Bytes (OB). Most important for the overall device security, the OB contains the Read-Protection Level (RDP Level) which is effectively the security configuration of the device. RDP levels range from Level 2 in which all debugging interfaces and bootloaders are disabled, Level 1 in which limited bootloader functionality and limited debug capabilities are enabled and

Level 0, which allows full access. By default, the Trezor and its derivatives are configured with RDP Level 2, the strongest level of security offered by the STM32.

During BootROM execution, the value of RDP is checked. If the value during the RDP check is not RDP2 then the BootROM checks the bootstrapping of two dedicated I/O pins against a predefined boot pattern, i.e. the logic levels of the BOOT0 and BOOT1 pins. If the boot pattern is not met, regular execution continues and the application code is executed from flash. However, if the pattern is met, i.e. in the case of the STM32F2, if BOOT0 is high, and BOOT1 is low, then the integrated BootROM serial bootloader and DFU bootloaders are enabled. Because the STM32 supports multiple serial protocols, one or more serial bootloaders are initialized and disabled when a valid synchronization condition is met on one of the serial interfaces. Once synchronized, the bootloader enters a mode where it receives and executes commands.

Patterns	Condition
Pattern1	Boot0(pin) = 1 and Boot1(pin) = 0

Table 2: The bootloader activation patterns as found in the datasheet of the STM32F205

STM32 BootROM Bootloader

The STM32 BootROM bootloader supports a variety of commands, including commands capable of reading and writing flash.

Command ⁽¹⁾	Command code	Command description
Get ⁽²⁾	0x00	Gets the version and the allowed commands supported by the current version of the bootloader.
Get Version & Read Protection Status ⁽²⁾	0x01	Gets the bootloader version and the Read Protection status of the Flash memory.
Get ID ⁽²⁾	0x02	Gets the chip ID.
Read Memory ⁽³⁾	0x11	Reads up to 256 bytes of memory starting from an address specified by the application.
Go ⁽³⁾	0x21	Jumps to user application code located in the internal Flash memory or in the SRAM.
Write Memory ⁽³⁾	0x31	Writes up to 256 bytes to the RAM or Flash memory starting from an address specified by the application.
Erase ⁽³⁾⁽⁴⁾	0x43	Erases from one to all the Flash memory pages.
Extended Erase ⁽³⁾⁽⁴⁾	0x44	Erases from one to all the Flash memory pages using two byte addressing mode (available only for v3.0 USART bootloader versions and above).
Write Protect	0x63	Enables the write protection for some sectors.
Write Unprotect	0x73	Disables the write protection for all Flash memory sectors.
Readout Protect	0x82	Enables the read protection.
Readout Unprotect ⁽²⁾	0x92	Disables the read protection.

Table 3: The list of supported bootloader commands, from the document “USART protocol used in the STM32 bootloader”

The “Read Memory” command can be used to read up to 256 bytes from the flash of the device. Analysis of the BootROM determined that this command performs a check for the RDP level every time the command is called, and only allows reading of flash-contents if the RDP level is set to RDP Level 0.

Bypassing the RDP Check for the Read Memory command

The STM32s used in wallets like the Trezor One are set to RDP Level 2 at manufacturing time. This deactivates all debugging features and disables the integrated BootROM bootloader. With voltage glitching it is possible to corrupt the RDP value being read from the Option Bytes, as shown in the Wallet.Fail research. This effectively allows an attacker to downgrade the security configuration of a target device from RDP Level 2 to RDP Level 1. A downgrade from RDP Level 1 to RDP Level 0 was determined to be infeasible in practice, due to the hamming distance between RDP Level 0 and RDP Level 2. By performing a voltage-glitch during BootROM execution it is possible to re-enable the JTAG and SWD debugging interfaces. It was determined that the integrated BootROM bootloader can be re-enabled in a similar fashion.

On the STM32F205, for example, by glitching at approximately 170us into BootROM execution it is possible to re-enable JTAG and SWD. The system will effectively enable the JTAG and SWD interfaces with the same behavior (i.e. the same limitations in terms of access) as RDP Level 1. It was determined that a voltage glitch at approximately 180us into BootROM execution will re-enable the integrated BootROM bootloader with the same behavior as the Bootloader at RDP Level 1. Once communication with the integrated BootROM bootloader is established, i.e the synchronization is complete, it is possible to issue commands available within the BootROM bootloader at RDP Level 1, for example GET ID. However, commands that are not available at RDP Level 1 will result in the STM32 returning a NACK and failing.

However, since it was determined that for certain commands, i.e. Read Memory, the BootROM bootloader command handler checks if the RDP Level of the device is RDP Level 0 for each command issued to it. A voltage glitch timed to coincide with the RDP Level check of the command handler while processing commands that are disabled for RDP Levels other than RDP Level 0 can result in a bypass of the RDP Level check and the command succeeding as a result. This means that it is possible to glitch commands that should fail based on the device's RDP configuration (i.e. bypassing the command handler that should have returned a NACK). As a result, it is possible to execute commands that are not available at RDP Level 1 or RDP Level 2. If applied to the Read Command, it is possible to arbitrarily read flash memory from the microcontroller. Since the cryptographic seeds of many STM32-based wallets are stored in the STM32 flash, the seed storage of these devices can be compromised.

Voltage-Glitching Hardware Setup to Dump Flash

A Digilent Arty A7 FPGA development board was used for glitch and pulse generation, as well as instrumenting the STM32 and accurately timing the glitch. An FTDI FT232H-based breakout board, the Adafruit FT232H, was used for UART serial communication with the BootROM bootloader command handler. A Maxim MAX4619 multiplexer was used to multiplex between a nominal operating voltage for the STM32 CPU core voltage and the glitch voltage, i.e. GND or 0v. A BreakingBitcoin board was used to simplify interacting, which is a pin-compatible Trezor breakout board. The same attack could be performed in-situ on a hardware wallet. However, removing the microcontroller and placing it in a socket was deemed to be easier than soldering all the connections (brimarily Boot0 and Boot1) for the in-situ attack.

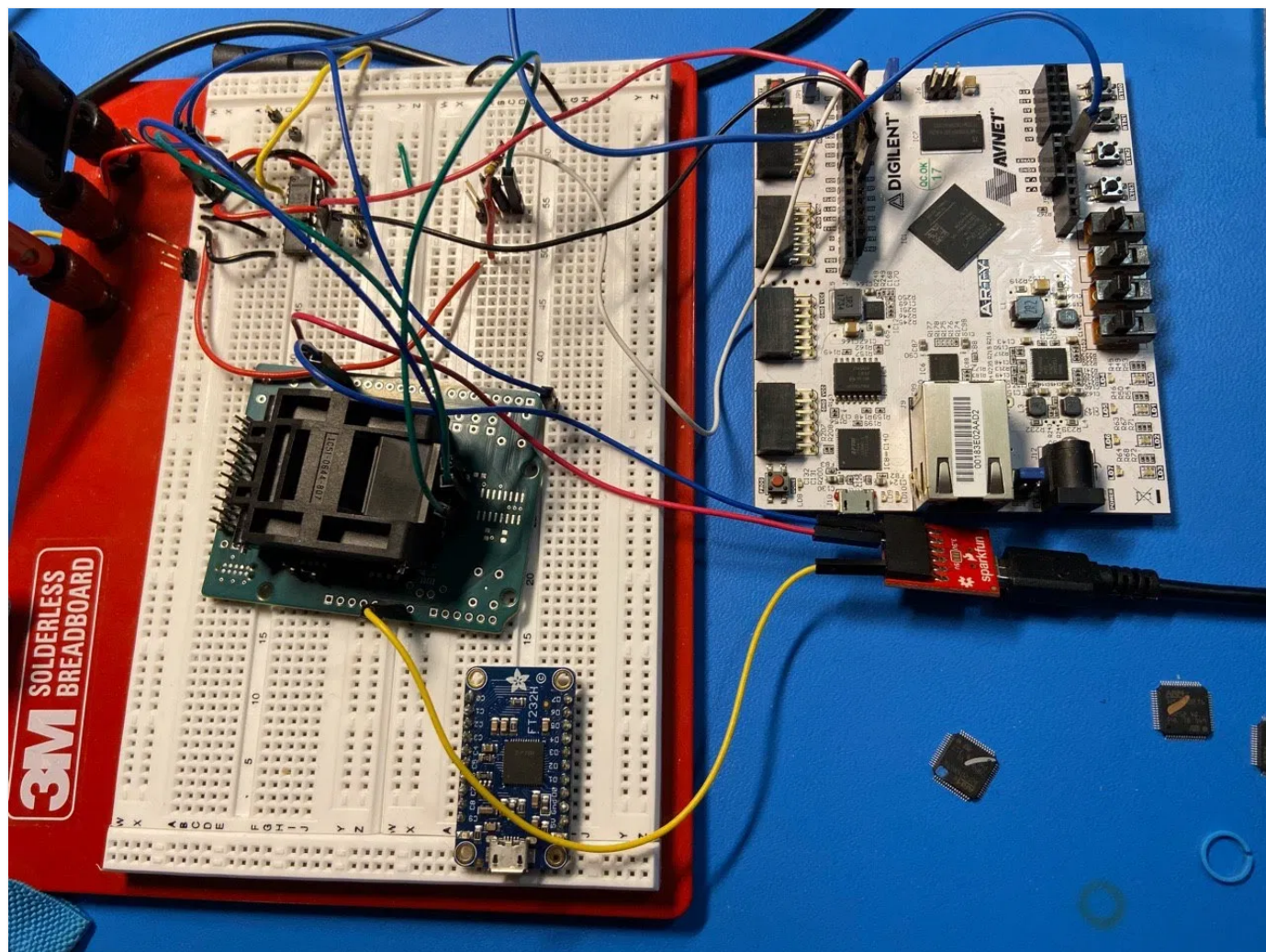


Figure 1: A glitching setup. On the top left, the MAX4619 Analog Multiplexer. A 64-pin LQFP64 socket for the STM32F205. On the right, a red FT232H USB-UART Adapter for interfacing to the BootROM UART Bootloader. Top right, a Digilent Arty A7 FPGA development board.

Brute-forcing the PIN of a Flash Dump

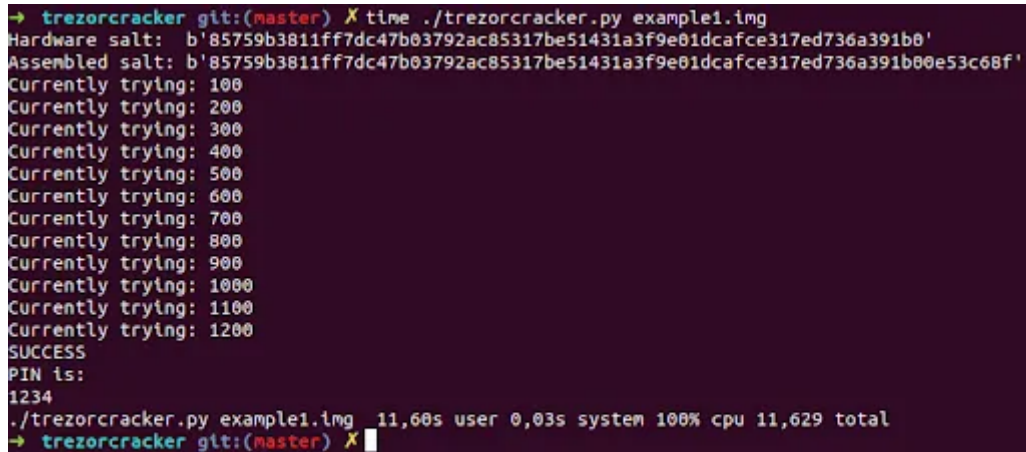
The Trezor encrypts its confidential storage with a key that is derived from the PIN and a salt. The salt consists of a hardware salt stored in the OTP bytes and a random salt from flash generated when the device is provisioned. The hardware salt is 44 bytes long and can be read with a single read of the option bytes.

The Trezor organizes its storage in “entries”, which can be identified by their app-value and their key-value. The entry containing the salt, Encrypted Data Encryption Key (EDEK), the Encrypted Storage Authentication Key (ESAK) and the PIN verification code (PVC) can be found under the app-value 0 and the key-value 2. It is possible to locate this entry, as it always starts with the hex bytes “02 00 3C 00”. The full entry is 64 bytes long, so combined with the hardware salt read it is enough to have two successful reads (as 256 bytes can be dumped per read).

Once the salt, EDEK, ESAK and PVC have been retrieved, an additional hardware salt needs to be retrieved from the OTP bytes. This can be done by re-enabling the SWD interface or by re-

enabling the Bootrom bootloader and issuing the corresponding Read Memory Commands.

On a single-core CPU, using the Python “pycryptodome” library, a performance of roughly 85 hashes per second was achieved. This can be optimized significantly by utilizing a GPU. Even at this relatively slow speed, a 4 digit pin can be brute-forced in under 2 minutes.



```
→ trezorcracker git:(master) X time ./trezorcracker.py example1.img
Hardware salt: b'85759b3811ff7dc47b03792ac85317be51431a3f9e01dcafce317ed736a391b0'
Assembled salt: b'85759b3811ff7dc47b03792ac85317be51431a3f9e01dcafce317ed736a391b00e53c68f'
Currently trying: 100
Currently trying: 200
Currently trying: 300
Currently trying: 400
Currently trying: 500
Currently trying: 600
Currently trying: 700
Currently trying: 800
Currently trying: 900
Currently trying: 1000
Currently trying: 1100
Currently trying: 1200
SUCCESS
PIN is:
1234
./trezorcracker.py example1.img 11.60s user 0.03s system 100% cpu 11.629 total
→ trezorcracker git:(master) X
```

Example Brute Force Script

```
#!/usr/bin/env python3
```

```
import hashlib
import sys
import argparse
import binascii
import struct
from Crypto.Cipher import ChaCha20_Poly1305

parser = argparse.ArgumentParser(description='Crack some wallets.')
parser.add_argument('flash_dump', help='The flash dump to parse.')
parser.add_argument('-otp', help='Randomness from OTP as hex.', default=44*"00")
parser.add_argument('-debug', type=bool)

args = parser.parse_args()
flash_file = open(args.flash_dump, "rb")

def find_header():
    while True:
        data = flash_file.read(4)
```

```
if data == b"\x02\x00\x3c\x00":
    return
elif data == None:
    print("Couldn't find header.")
    sys.exit(1)

def find_header():

    # Salt from flash
    salt = flash_file.read(4)

    # EDEK + ESAK
    edek = flash_file.read(48)
    pvc = flash_file.read(8)

    # Salt computation:
    # hardware_salt (from collect_hw_entropy)
    # random_salt (Random buffer generated, stored in flash)
    # ext_salt – unused
    hardware_salt = hashlib.sha256(binascii.unhexlify(args.otp)).digest()

    salt_assembled = hardware_salt + salt
    print(f"Hardware salt: {binascii.hexlify(hardware_salt)}")
    print(f"Assembled salt: {binascii.hexlify(salt_assembled)}")

def trezor_pbkdf(pin, salt):
    # PIN is always prefixed with 1
    pin_bytes = struct.pack("<I", int("1" + pin))
    if args.debug:
        print(f"PIN bytes: {binascii.hexlify(pin_bytes)}")
    dk = hashlib.pbkdf2_hmac('sha256', pin_bytes, salt, 10000, dklen=352/8)
    return dk

def chacha_enc(kek, keiv, data):
    cipher = ChaCha20_Poly1305.new(key=kek, nonce=keiv)
    # Return DEK & TAG
    return cipher.encrypt_and_digest(data)

def chacha_dec(kek, keiv, edek):
    cipher = ChaCha20_Poly1305.new(key=kek, nonce=keiv)
    return cipher.decrypt(edek)
```

```
for i in range(1, 9999):
    if i % 100 == 0:
        print(f"Currently trying: {i}")
        dk = trezor_pbkdf(str(i), salt_assembled)

# First 256 bits are Key Encryption Key (KEK)
KEK = dk[:int(256/8)]
# Remaining 96 bits are Key Encryption Initialization Vector (KEIV)
KEIV = dk[int(256/8):]

if args.debug:
    print(f"KEK: {binascii.hexlify(KEK)}")
    print(f"KEIV: {binascii.hexlify(KEIV)}")

DEK = chacha_dec(KEK, KEIV, edek)
# Encrypt to get the TAG, unfortunately seems to be the only
# way to retrieve it from Pycryptodome.
enc, TAG = chacha_enc(KEK, KEIV, DEK)
if args.debug:
    print(f"DEK: {binascii.hexlify(DEK)}")
    print(f"TAG : {binascii.hexlify(TAG)}")
    print(f"PVC : {binascii.hexlify(pvc)}")

if pvc in TAG:
    print("SUCCESS")
    print("PIN is:")
    print(i)
    sys.exit(0)
```

Key derivation

The salt, consisting of the hardware salt appended with the salt from the entry, is used in combination with the PIN (prefixed by a 1 and converted to a 32-byte little endian integer) to derive a key using PBKDF2-HMAC with SHA256 and a derived key length of 362 bits. The first 256 bits of the result are used as the Key Encryption Key (KEK), while the last 96 bits are used as the Key Encryption Initialization Vector (KEIV).

Key verification

The Data Encryption Key (DEK) is generated from the EDEK by decrypting the EDEK using the KEK and the KEIV. The encryption algorithm used is ChaCha20 with the Poly1305 message authentication code. If the first 64 bits of the PVC are identical with the first 64 bits of the TAG, then the PIN was correct and the DEK was successfully recovered.

If You See Something, Earn Something

- As you can tell, our security team is always on the lookout for vulnerabilities that may pose a threat to our clients.
- If you're interested in helping us identify potential bugs on our platform, check out our [Bug Bounty program](#).
- We payout a minimum of \$100 for each issue and potentially more for serious threats.

POSTED IN UNCATEGORIZED • TAGGED FEATURED

PREV

Daily Market Report for January 30 2020

NEXT

Daily Market Report for January 31 2020

TRANSLATE

Select Language

Powered by  Google Translate

PAGES

[About](#)

[Archive](#)

RSS FEEDS

[All Topics](#)

[| Kraken News](#)[| Market Reports](#)

ABOUT

Founded in 2011, Kraken Digital Asset Exchange is one of the world's largest and oldest bitcoin exchanges with the widest selection of digital assets and national currencies. Based in San Francisco with offices around the world, Kraken's trading platform is consistently rated the best and most secure digital asset exchange by independent news media. Trusted by hundreds of thousands of traders, institutions, and authorities, including Germany's BaFin regulated Fidor Bank, Kraken is the first exchange to display its market data on the Bloomberg Terminal, pass a cryptographically verifiable proof-of-reserves audit, and one of the first to offer leveraged margin trading. Kraken investors include Blockchain Capital, Digital Currency Group, Hummingbird Ventures and Money Partners Group. For more information, please visit: [Kraken.com](https://kraken.com).
