Verifying Hardware Security Modules with Information-Preserving Refinement

Anish Athalye, M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL
HSMs: powerful tools for securing systems

Factor out core security operations

Provide security under host compromise

Many types of HSMs
- U2F token
- PKCS#11 HSM
- Hardware wallet
- iPhone Secure Enclave

Hundreds of millions of deployed HSMs
HSMs suffer from bugs

Hardware

Software

Timing side channels

2.2.2 Data Read when the CPU accesses successively SRAM address “A” and SRAM address “A + offset of 16 KBytes (0x4000)”

Description

If the CPU writes to an address A in the SRAM memory and immediately (the cycle after) reads an address B in the SRAM memory, while B = A+0x4000, the read operation will return the content at address A instead of the content of address B.

CVE-ID

CVE-2019-18671

Learn more at National Vulnerability Database (NVD)

• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Insufficient checks in the USB packet handling of the ShapeShift KeepKey hardware wallet before firmware 6.2.2 allow out-of-bounds writes in the .bss segment via crafted messages. The vulnerability could allow code execution or other forms of impact. It can be triggered by unauthenticated attackers and the interface is reachable via WebUSB.

CVE-ID

CVE-2021-31

Description

Insufficient length checks in the ShapeShift KeepKey hardware wallet firmware overflow via crafted messages. The overflow in ethereum_extractETH can circumvent stack protections and lead to code execution. The vulnerability over WebUSB.

Security Advisory 2015-04-14


Summary

Format String vulnerability in KeepKey version 4.0.0 allows attackers to trigger information display (of information that should not be accessible), related to text containing characters that the device’s font lacks.
Goal: HSMs without security vulnerabilities

Rule out hardware, software, and timing side-channel vulnerabilities

Threat model

- Powerful adversary that gains control of host machine
- Full control over I/O interface to HSM
- Physical attacks and other side channels: out of scope
Approach: formal verification

Covers security and timing channels

Includes hardware and software

Low proof overhead

Specification

Implementation
Software (.c)
Hardware (.v)

Proof

Knox

✓ OK /
✗ FAIL
Related work

*Knox* is the first to verify correctness and security of hardware and software

  Including timing side channels

Hardware/software co-verification: Bedrock2 [PLDI'21], CakeML [PLDI'19]

  Focused on correctness, not security

Application security verification: Ironclad Apps [OSDI'14]

  Doesn't cover hardware or side channels
Contributions

*Information-preserving refinement (IPR)*, a new security definition

*Knox framework* for verifying HSMs using IPR

Case studies: built and verified 3 simple HSMs
- PIN-protected backup HSM
- Password-hashing HSM
- TOTP token

Approach rules out hardware bugs, software bugs, and timing side channels
**Example: PIN-protected backup HSM**

```python
var bad_guesses = 0, secret = 0, pin = 0

def store(new_secret, new_pin):
    secret = new_secret
    pin = new_pin
    bad_guesses = 0

def retrieve(guess):
    if bad_guesses >= 10:
        return 'No more guesses'
    if guess != pin:
        bad_guesses = bad_guesses + 1
        return 'Incorrect PIN'
    bad_guesses = 0
    return secret
```

Functional specification

Describes input-output behavior

No notion of timing
Implementation

Implementation includes hardware/software

CPU
Code that runs on it
Peripherals
Persistent memory
...

Interface: wires

Read output wires
Write input wires
Wait for a cycle
How to relate implementation to spec?

Want to capture:

1. Functional correctness: implementation implements spec
2. **Non-leakage**: Wire-level interface leaks no more than spec
   Including timing, e.g., PIN comparison with `strcmp()`

Implementation is at the level of wires

Specification is at the level of functions (has no notion of wires)
Information-preserving refinement (IPR)

Defined as indistinguishability between a real and an ideal world

Inspired by formalization of zero knowledge in cryptography
Information-preserving refinement (IPR)

Defined as indistinguishability between a real and an ideal world.

Inspired by formalization of zero knowledge in cryptography.

Interface adapters in each direction.

Matching interfaces
**IPR: driver**

*Driver:* translates **spec-level operations** to **wire-level I/O**

Like a device driver in an OS

Trusted, part of the specification

Captures functional correctness

```scheme
(define (store secret pin)
  (send-byte #x02) ; command number
  (send-bytes pin)
  (send-bytes secret)
  (recv-byte)); wait for ack

(define (wait-until-clear-to-send)
  (while (get-output 'rts))
  (tick)); wait a cycle

(define (send-bit bit)
  (set-input 'rx bit)
  (for ([i (in-range BAUD-RATE)])
    (tick)))

(define (send-byte byte)
  (wait-until-clear-to-send)
  (send-bit #b0) ; send start bit
  ;; send data bits
  (for ([i (in-range 8)])
    (send-bit (extract-bit byte i)))
  (send-bit #b1)); send stop bit
```
**IPR: emulator**

*Emulator* mimics *wire-level behavior*

Without direct access to secrets  
With queries to *spec-level operations*

Proof artifact, constructed by developer  
(just needs to exist)

Captures non-leakage
IPR rules out timing channels

What if circuit leaked info through timing, e.g., strcmp()?

Emulator does not exist: can get return value using query to retrieve(), but can't reproduce timing behavior.
IPR: emulator construction

Copy circuit, but replace operations on secret state with queries to spec
IPR transfers security properties from spec to impl

Only reveals secret when correct PIN supplied

Enforces guess limits

Forgets old secret/pin when store() is called

Doesn't leak past PIN guesses

```python
var bad_guesses = 0, secret = 0, pin = 0

def store(new_secret, new_pin):
    secret = new_secret
    pin = new_pin
    bad_guesses = 0

def retrieve(guess):
    if bad_guesses >= 10:
        return 'No more guesses'
    if guess != pin:
        bad_guesses = bad_guesses + 1
        return 'Incorrect PIN'
    bad_guesses = 0
    return secret
```
Knox framework

~ 3000 LOC on top of Rosette [PLDI'14]

Symbolically execute entire circuit + code

Relies on human guidance through *hints*
Evaluation: case studies

3 simple HSMs, run on an FPGA

Hardware: minimal RISC-V CPU, cryptographic accelerator, UART, ...

Software: control logic, peripheral drivers, HOTP, HMAC, ...

Succinct specifications

Low proof overhead

<table>
<thead>
<tr>
<th>HSM</th>
<th>Spec core</th>
<th>Spec total</th>
<th>Driver</th>
<th>HW</th>
<th>SW</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIN-protected backup HSM</td>
<td>32</td>
<td>60</td>
<td>110</td>
<td>2670</td>
<td>190</td>
<td>470</td>
</tr>
<tr>
<td>Password-hashing HSM</td>
<td>5</td>
<td>150</td>
<td>90</td>
<td>3020</td>
<td>240</td>
<td>650</td>
</tr>
<tr>
<td>TOTP token</td>
<td>10</td>
<td>180</td>
<td>80</td>
<td>2950</td>
<td>360</td>
<td>830</td>
</tr>
</tbody>
</table>

Lines of code for case studies
Subtle bug involving persistence and timing

```c
void retrieve(uint8_t *guess) {
    // return error if PIN guess limit exceeded
    // ...

    // check PIN guess and update bad_guesses
    if (!constant_time_cmp(&entry->pin, guess)) {
        // entry points to persistent storage
        entry->bad_guesses++;
        uart_write(ERR_BAD_PIN);
        return;
    }

    entry->bad_guesses = 0;
    // output secret
    // ...
}
```

- guess PIN = 0000, PIN = 0001, ...
- bad_guesses = 0
- bad_guesses++
- force device reset
- Adversary can't tell which branch was taken
  (no outputs up to this point) but still, security bug!
  Resets guess count to 0.
Real implementations have similar code

SoloKey: pattern similar to our bug

Other HSMs like OpenSK have more robust code to avoid this issue

```c
int8_t ret = verify_pin_auth_ex(CM->pinAuth, (u
1569
1570 if (ret == CTAP2_ERR_PIN_AUTH_INVALID)
1571 {
1572     ctap_decrement_pin_attempts();
1573     if (ctap_device_boot_locked())
1574         { return CTAP2_ERR_PIN_AUTH_BLOCKED;
1575     }
1576     return CTAP2_ERR_PIN_AUTH_INVALID;
1577 }
1578 }
1579 else
1580 {
1581     ctap_reset_pin_attempts();
1582 }
```
Conclusion

Information-preserving refinement (IPR)

Implementation reveals no more information than specification

Knox framework

For verifying HSMs using IPR

Case studies

Built and verified 3 simple HSMs

anish.io/knox