HPE iLO 5 security

Go home cryptoprocessor, you're drunk!

Alexandre Gazet, Fabien Périgaud & Joffrey Czarny

August 4-5, 2021

BRIEFINGS

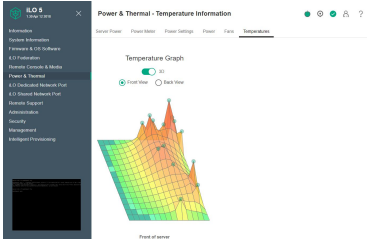
Alexandre Gazet

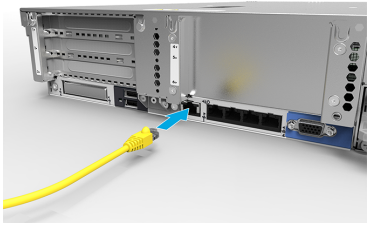
- Airbus security lab
- Co-author of "Practical Reverse Engineering"
- Work in progress

Fabien Perigaud

- Reverse Engineering team Technical Lead @ Synacktiv
- Vulnerability researcher
- CTF/challenge enthousiast in a previous life


HPE iLO
U160
iLO


iLO 5
Information
System Information
Firmware & OS Software
iLO Federation
Remote Console & Media
Power & Thermal
iLO Dedicated Network Port
iLO Shared Network Port
Remote Support
Administration
Security
Management
Intelligent Provisioning
Power & Thermal - Temperature Information
Server Power
Power Meter
Power Settings
Power
Fans
Temperatures
Temperature Graph
Front View
Back View
Front of server



A well-known technology:

- Deep-dive analysis of HPE iLO4 and iLO5[1][2]
- CVE-2017-12542: pre-auth RCE (iLO4)
- CVE-2018-7078: updates signature verification bypass
- CVE-2018-7113: secure boot bypass (iLO5)
- Abuse of the DMA feature to compromise host OS from iLO[3]
- etc.

[1] https://github.com/airbus-seclab/ilo4_toolbox
[2] Talks at RECON, SSTIC, ZeroNights
[3] https://github.com/Synacktiv/pcileech_hpilo4_service

- Early 2020, new iLO5 firmware versions: **2.x**
- High entropy data blob ⇒ **encrypted updates**
- Installation notes:

  "Upgrading to iLO 5 version 2.10 is supported on servers with iLO 5 1.4x or later installed."

- "Transitional" 1.4x versions are **unencrypted**

Encryption

Objective(s)

- Locate the implementation of the decryption
- Re-implement the decryption mechanism
- Update our firmware analysis toolbox

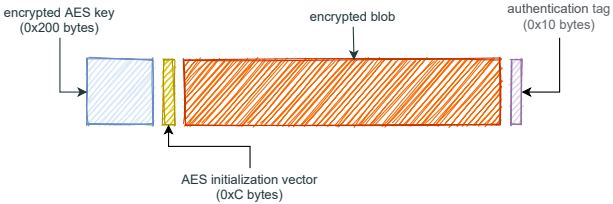
T - 0

Boot chain?

- Bootloaders and kernel stay the same between versions 1.3x and 1.4x

Firmware Update Manager (FUM)

- Task/module in userland
- Manages the cryptographic signature check and the updates writing in the flash memory
- Discovery of the new decryption feature



- OpenSSL primitives: envelope encryption[4]
- Authenticated symmetric ciphering of the data, AES Galois/Counter Mode (GCM)
- The symmetric key (AES) is **sealed** by a public key and asymmetric encryption (RSA)
- ⇒ The RSA private key is required to "*open the envelope*"

[4] https://wiki.openssl.org/index.php/EVP_Asymmetric_Encryption_and_Decryption_of_an_Envelope

RSA private key

```
.fum.elf.RW:0006186C 31 71 58 7A+              DCB "FSvA9nEAVQoVhRPcXA1rmHo32x5wopf2WpAd5awmUl5nxr0GmU42qf0CAwEAAQ=="
.fum.elf.RW:0006186C 52 4D 36 0A+              DCB 0xA
.fum.elf.RW:0006186C 71 6E 78 39+              DCB "-----END RSA PUBLIC KEY-----",0xA,0
.fum.elf.RW:00061B74             ; char RSA_PRIVATE_KEY[3435]
.fum.elf.RW:00061B74 2D 2D 2D 2D+RSA_PRIVATE_KEY DCB "-----BEGIN ENCRYPTED PRIVATE KEY-----",0xA
.fum.elf.RW:00061B74 2D 42 45 47+                                       ; DATA XREF: fum_decrypt_hpimage+2C↑o
.fum.elf.RW:00061B74 49 4E 20 45+                                       ; fum_decrypt_hpimage+34↑o
.fum.elf.RW:00061B74 4E 43 52 59+              DCB "MIIJrTBXBgkqhkiG9w0BBQ0wSjApBgkqhkiG9w0BBQwwHAQIjhdNQLNz8zoCAggA"
.fum.elf.RW:00061B74 50 54 45 44+              DCB 0xA
.fum.elf.RW:00061B74 20 50 52 49+              DCB "MAwGCCqGSIb3DQIKBQAwHQYJYIZIAWUDBAEqBBAEJcEYMX1ZMZFCj4/IdBSrBIIJ"
.fum.elf.RW:00061B74 56 41 54 45+              DCB 0xA
.fum.elf.RW:00061B74 20 4B 45 59+              DCB "UO2BH0h/huyP2JBd7bmEVyEhqK87PYg1FI6EuvxXekgCwG/567YPvzJZxYDvdssr"
.fum.elf.RW:00061B74 2D 2D 2D 2D+              DCB 0xA
```

- PEM base64, PKCS♯8
- The RSA private key is protected by a passphrase
- OpenSSL: `PEM_read_bio_RSAPrivateKey`
- ⇒ a callback function provides the passphrase

```
int __fastcall pem_password_cb(char *buf, unsigned int size, int rwflag, void *u)
{
  key_mask[0] = 0;
  key_mask[1] = 0xCE;
  key_mask[2] = 0;
  key_mask[3] = 0xD00000;
  key_mask[4] = 0x86C900;
  key_mask[5] = 0x9A0000;
  key_mask[6] = 0x700000;
  key_mask[7] = 0x190000;
  if ( size < 0x20 || rwflag == 1 )
    return 0;
  HW_SECRET[0] = MEMORY[0x1F200D8];
  HW_SECRET[1] = MEMORY[0x1F20B00];
  HW_SECRET[2] = MEMORY[0x1F20B08] & 0xFFFFFFF0;
  HW_SECRET[3] = MEMORY[0x1F20B0C];
  HW_SECRET[4] = MEMORY[0x1F21810];
  HW_SECRET[5] = MEMORY[0x1F21840];
  HW_SECRET[6] = MEMORY[0x1F21850];
  HW_SECRET[7] = MEMORY[0x1F21890];
  for ( i = 0; i < 0x20; ++i )
    buf[i] = *((_BYTE *)key_mask + i) ^ *((_BYTE *)HW_SECRET + i);
  return 0x20;
}
```

- Memory range `0x1F2xxxx` ≈ configuration registers from the iLO5 SOC (System-On-Chip) mapped into the userland task.
- ⇒ The `HW_SECRET` buffer is a “hardware” key

Hardware Key Extraction

Objective(s)

- Exploit a vulnerability on iLO 5
- Read SOC registers
- Re-implement decryption mechanism
- Update our firmware analysis toolbox

T + 2 days

- CVE-2018-7105 by Nicolas Iooss[5]
- Impacts iLO4 and iLO5
- "*Remote execution of arbitrary code, Local Disclosure of Sensitive Information*"
- Format-string vulnerability in the proprietary SSH restricted shell (post-authentication)
- Exploitation code available for iLO4
- ⇒ Port the first stage of the exploit to iLO5

[5] @fishilico: "*Add SSH exploit for CVE-2018-7105*"
https://github.com/airbus-seclab/ilo4_toolbox/commit/430bfb95

Difficulties

1. Vulnerability is in the task `ConAppCli` ⇒ SOC secrets are not mapped in the task's context
2. Memory R/W primitive with some constraints:
   - Addresses with null-bytes forbidden
   - As well as some special chars ("\n", "\r", etc.)

Memory aesthete

1. Patch/hook a function pointer to call a primitive exposed by the OS (≈ `mmap`)
2. Double mapping (virtual-physical translation) to remove null-bytes

```
int __cdecl timer_func()
{
  int result; // r0

  memmap(0x800000000LL);
  result = 1000 * (MEMORY[0x1F2000C] & 3) / 3 + 1000 * (unsigned __int8)(MEMORY[0x1F2000C] >> 2);
  dword_9A6D8 = result;
  return result;
}
```

memmap function

1. High level wrapper upon kernel primitives
2. But, depends upon structures defined in the task

Mapping descriptors

```
fum.elf.RW:00062AF0                 ; MR_RANGE MR_MAPPING
.fum.elf.RW:00062AF0 00 00 00 80+MR_MAPPING       MR_RANGE <0x80000000, 0x1F0000, 1, 0, 0>
.fum.elf.RW:00062AF0 00 00 F0 01+                                 ; DATA XREF: memmap_init:loc_2EA48↑o
.fum.elf.RW:00062AF0 01 00 00 00+                                 ; memmap_init:loc_2EA70↑r ...
.fum.elf.RW:00062B08 00 F0 0E 80+                 MR_RANGE <0x800EF000, 0x1F0100, 2, 0, 0>
.fum.elf.RW:00062B20 00 00 0F 80+                 MR_RANGE <0x800F0000, 0x1F0200, 4, 0, 0>
.fum.elf.RW:00062B38 00 00 1F 80+                 MR_RANGE <0x801F0000, 0x1F0300, 8, 0, 0>

                                  • • •

.fum.elf.RW:00062E20 00 00 00 83+                 MR_RANGE <0x83000000, 0x600000, 0x400000000, 0, 0>
.fum.elf.RW:00062E38 00 00 00 C0+                 MR_RANGE <0xC0000000, 0x1F20000, 0x800000000, 0, 0>
.fum.elf.RW:00062E50 00 10 00 C0+                 MR_RANGE <0xC0001000, 0x1F2100, 0x1000000000, 0, 0>
.fum.elf.RW:00062E68 00 20 00 C0+                 MR_RANGE <0xC0002000, 0x1F2200, 0x2000000000, 0, 0>
.fum.elf.RW:00062E80 00 80 00 C0+                 MR_RANGE <0xC0008000, 0x1F2300, 0x4000000000, 0, 0>
.fum.elf.RW:00062E98 00 90 00 C0+                 MR_RANGE <0xC0009000, 0x1F2400, 0x8000000000, 0, 0>
.fum.elf.RW:00062EB0 00 C0 00 C0+                 MR_RANGE <0xC000C000, 0x1F2500, 0x10000000000, 0, 0>
```

Key points

1. Describe devices memory ranges
2. For example, mapping the kernel memory is not possible
3. And yes, the mappings table can be **read and written** from the task's context :)

```
struct MEM_RANGE
{
    void *phys_addr;
    void *virt_addr;
    unsigned __int64 mask;
    int field_10;
    int field_14;
};
```

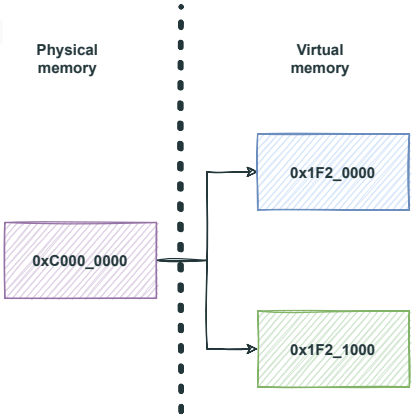
Single-byte memory patch

- From:

```
MR_RANGE <
  0xC0000000, 0x1F20000,
  0x800000000, 0, 0>
```

- To:

```
MR_RANGE <
  0xC0000000, 0x1F21000,
  0x800000000, 0, 0>
```



```
[+] dumping iLO HW keys:
[+] MMU: memory mapping magic:
  >> patch 0x2008@0xb8264
  >> patch 0x1008@0xb824c
  >> patch 0x18@0xb818c
  >> patch 0xc00000@0xb8181
[+] command hooks:
  >> hook 0x70158@0xb0bec

>> 0xbf7fffc3@0x1f200d8
>> 0x01851c0d@0x1f20b01
>> 0x32f26410@0x1f20b08
>> 0x08000621@0x1f20b0c
>> 0x8000009f@0x1f21810
>> 0x81001012@0x1f21840
>> 0x810010dc@0x1f21850
>> 0x81001121@0x1f21890
```

Hardware key extraction through a fmt-string over VPN

⇒ **Firmware 2.x decrypted with success!!!**

```
0)  Secure Micro Boot 2.02, type 0x03, size 0x00008000
1)  Secure Micro Boot 2.02, type 0x03, size 0x00005424
2)           neba9 0.10.13, type 0x01, size 0x00005644
3)              neb926 0.3, type 0x02, size 0x00000ad0
4)           neba9 0.10.13, type 0x01, size 0x00005644
5)              neb926 0.3, type 0x02, size 0x00000ad0
6)   iLO 5 Kernel 00.09.60, type 0x0b, size 0x000d6158
7)   iLO 5 Kernel 00.09.60, type 0x0b, size 0x000d6158
8)                 2.10.54,  type 0x20, size 0x001dd9dc
9)                 2.10.54,  type 0x23, size 0x00f2ad0b
a)                 2.10.54,  type 0x22, size 0x004e7f2
```

Wait!!!!

- 3 userland images
- Previously 2 images only: `main` (type 0x20) et `recovery` (type 0x22)
- **(double) facepalm: the new type 0x23 is encrypted...**
- The type 0x20 is now minimalist, however **not encrypted**


MARIO
058700
×20
WORLD
1-4
TIME
200
THANK YOU MARIO!
BUT OUR PRINCESS IS IN
ANOTHER CASTLE!

Second encryption layer

Objective(s)

- Understand the second layer of encryption
- Re-implement the second decryption layer
- Update our firmware analysis toolbox

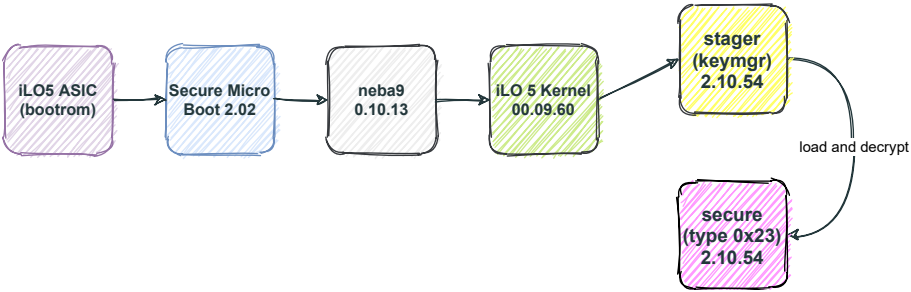
T + 1 week

Dissecting the new `main` image (type `0x20`)

```
---------------[ Shared modules ]---------------

> mod 0x00 -           libINTEGRITY.so size
> mod 0x01 -                   libc.so size
> mod 0x02 -             libopenssl.so size

-----------------[ Tasks List ]-----------------

> task 01 - path keymgr.elf - size 0x00013588
```

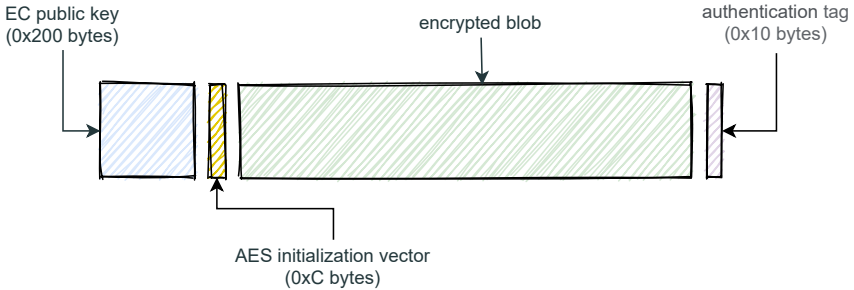
- Single-task image
- 3 libs including OpenSSL

An “augmented” bootchain

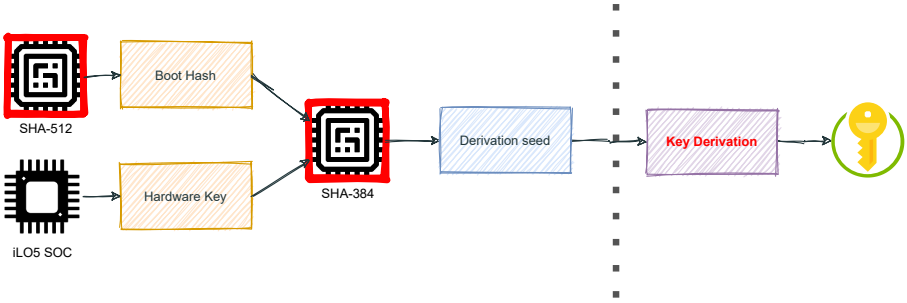


We are looking for the Keymaker

Layout of an encrypted image



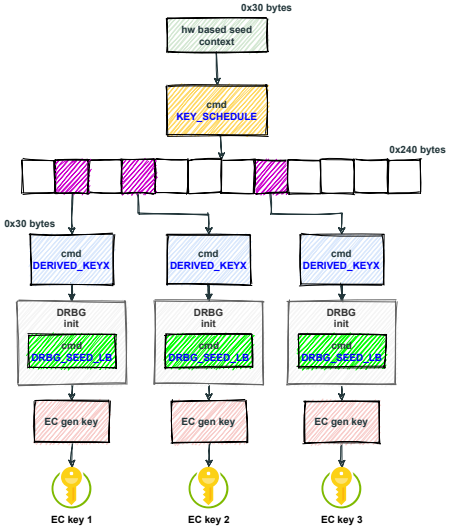
- **A private key is reconstructed during the boot of `keymgr`**
- OpenSSL primitives: `ECDH_compute_key`, Elliptic Curve Diffie-Hellman
- Used with the public key located in the image header to derive a shared secret
- Authenticated symmetric encryption of the data (AES-GCM)



keymgr: great reverse-engineering challenge

- **X factor: a cryptoprocessor**
  - Used operations: SHA384, AES-CTR, AES-GCM
- Two main steps:
  1. Derivation of a seed from hardware values
  2. A key scheduling function is fed with the seed

Key derivation is fun



"Commands", wrapper over the crypto-processor:

- Derivation step (SHA384) parametrized by a key:
  - "KEY_SCHEDULE"
  - "DERIVED_KEYX"
  - "DRBG_SEED_LB"

How to generate a deterministic key?

- OpenSSL's EC_KEY_generate_key
- Replacing default PRNG
- Deterministic seed (output from the DERIVED_KEYX command)

Objective

- Offline generation of this key? (without the iLO hardware)

Fail Hard. Fail Fast.

- Complete re-implementation in C, based on our static analysis
- Hardware value extraction through the 1day fmt-string
- $\Rightarrow$ **decryption failure**

Need to escalate

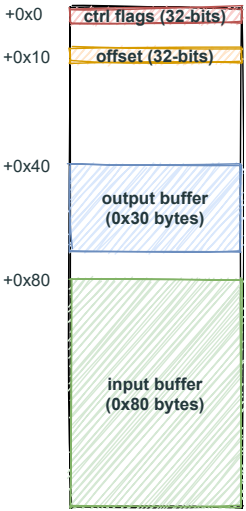
- Interfacing with the cryptoprocessor
- Step-by-step validation of the stages

Talking to the unknown

Objective(s)

- Validate our understanding of the exchanges with the cryptoprocessor
- Re-implement the second decryption layer
- Update our firmware analysis toolbox

T + 2 weeks



Control flags

- `SHA384_DIGEST_DATA_START`
- `SHA384_DIGEST_MORE_DATA`
- `SHA384_DIGEST_DATA_END`
- `SHA384_DIGEST_CLEAR_OUTPUT`

Offset

- Auto-incremented during writes into the input buffer
- Size of input data in bits

How?

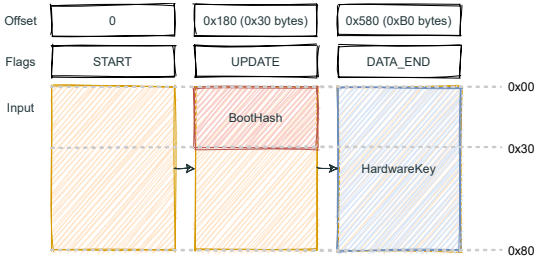
- Reading and writing the task's memory is enough

1-day to the rescue!

- Re-use of the SSH format-string vulnerability
- Mapping of the cryptoprocessor in the memory space
- Interaction through the exploit's R/W primitives

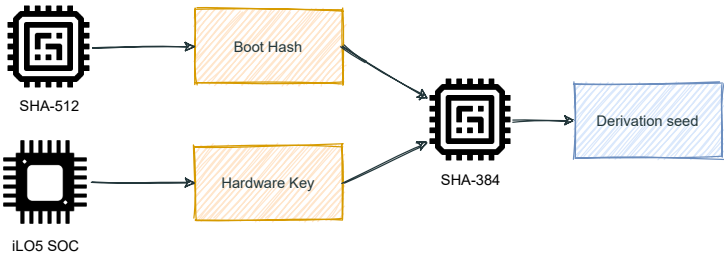
Bug #1: SHA384_DIGEST_MORE_DATA (UPDATE)

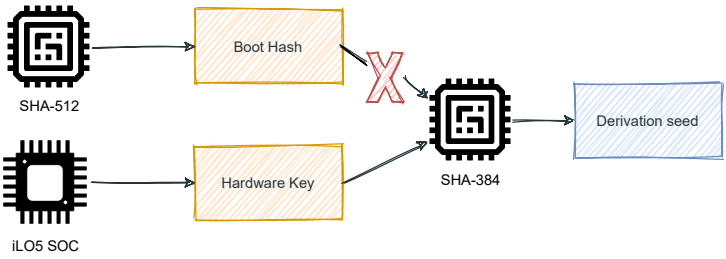
- Expected behavior: update of the internal state of the digest with the current content of the input buffer
- Observed behavior: if the input buffer has not been fully filled, data are ignored



Expected: `SHA384( BootHash || HardwareKey )`

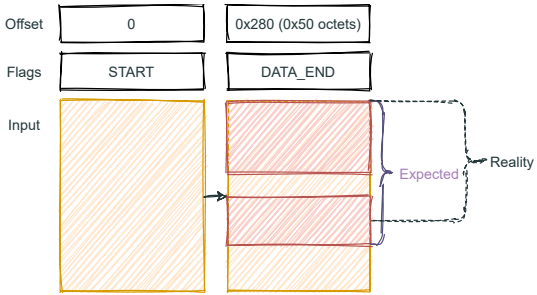
Observed: `SHA384( HardwareKey || HardwareKey[:len(BootHash)] )`


SHA-512
Boot Hash
iLO5 SOC
Hardware Key
SHA-384
Derivation seed


SHA-512
Boot Hash
iLO5 SOC
Hardware Key
SHA-384
Derivation seed

Bug #2: non-contiguous writes into the input buffer

- `Offset` register auto-incremented with the number of input bits, **without consideration for the position of the write in the input buffer**
- Internally the `Offset` value is used to read the data contiguously/linearly



Expected : `SHA384( INPUT[:0x60] )`

Observed : `SHA384( INPUT[:0x50] )`

Shall we perform a victory dance?

- Taking these 2 bugs into account (re-implementing them in our code)
- **New epic failure, the computed key is still invalid**

Debugging 101

Objective(s)

- Find debug information to understand our failures
- Grab this information
- Fix our implementation

T + 3 weeks

Debug messages

- Intermediary state

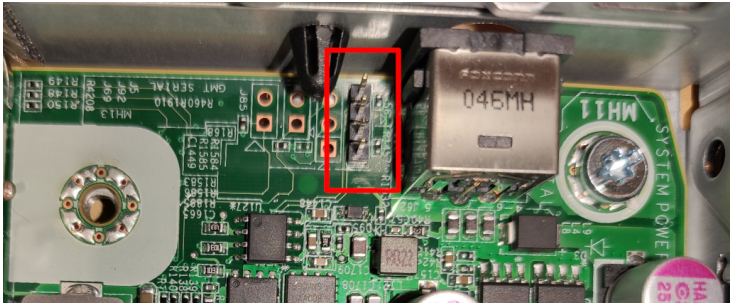
```
coproc_crypto_cmd(coproc_status, "KEY_SCHEDULE", 0, 0, 0, &DRBG_BUFFER_POOL, 0x240);
EVP_EncodeBlock(tmp_buffer, &DRBG_BUFFER_POOL, 0x21);// base64 encode
printf("Key Schedule Validation: %s\n", tmp_buffer);
```

- Final public key

```
ec_key = EC_KEY_POOL_.ec_key1;
bio = BIO_new_fp(FD_STDOUT, 0);             // dump key to stdout
PEM_write_bio_EC_PUBKEY(bio, EC_KEY_POOL_.ec_key1);
```

Goal

- Debug information is printed on the `UART` output
- Find `UART` on the `Microserver`



At boot time

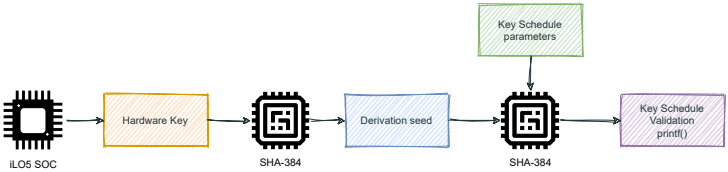
```
Key Schedule Validation: l03wN5OaDle9gyhfEJShR5jv0sKB0tfT25uk2U/vjxRA
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAE4StRIN6NFi6X000aNMLePDm0mYmXIpDf
03KrjkhWjZW8z3QNeyUXVNxHayZEKFL6Xk6vjkYYeJNdqg9yEzI0a2GK2emSgp4D
RNgUyUpix0jq5+lluKXWUyFQ6rBJ45Dr
-----END PUBLIC KEY-----
```

In our implementation

```
Key Schedule Validation: enRnbrYwZKTWBJF955yiy5VIbe4ROBE4g1EO5dOrYcVl
```

- Early failure in the derivation process
- Noose is tightening

Here be dragons



Theory

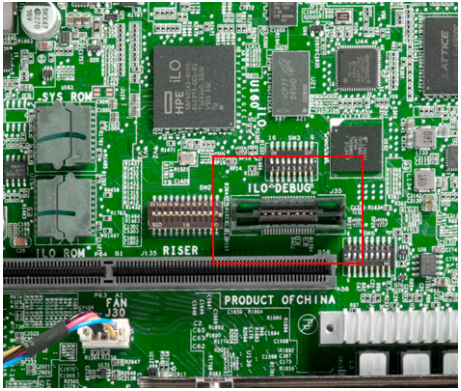
- Constants in hardware registers have changed between versions 1.x and 2.x
- We need better debugging capabilities

Hardware debugging

Objective(s)

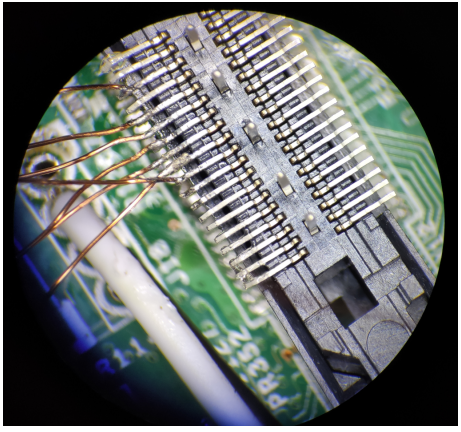
- Find debug ports
- Use this access to read the hardware registers
- Fix our implementation

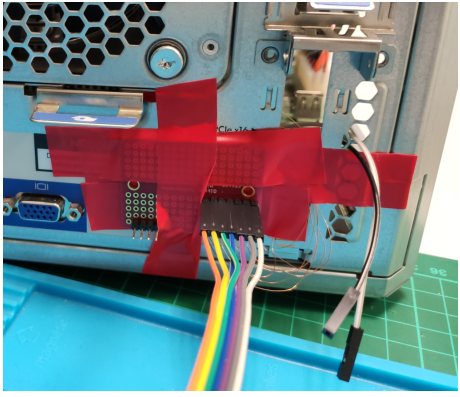
T + 3 weeks



On Microserver Gen10 motherboard

- Spotted on HPE website
- `MICTOR` port
- Should allow `JTAG` debugging





In the end, we bought a correct adapter :)

JTAG enumeration

```
Device ID #1: 0101 1011101000000000 01000111011 1 (0x5BA00477)
Device ID #2: 0000 0111100100100110 11110000111 1 (0x07926F0F)
```

Bad results

- Problems with TDI
- No solution found ⇒ we gave up
- We are software guys, let's find a software solution!

0day hunting

Objective(s)

- Find a new vulnerability in firmware versions 2.x
- Read the SOC registers
- Fix our implementation

T + 4 weeks

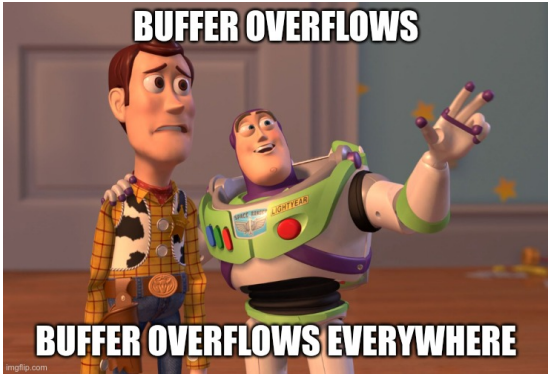
Target: 2.x firmwares

- CHIF interface
- Several tasks are reachable from the host
- `recovery` image is unencrypted: the `blackbox` task is present

`blackbox` task

- Many command handlers
- Command 5: debug menu
  - Text mode commands
  - Output on `UART`
  - Example:

```
bb fdump (file)       — Hex dump 'file'. Absolute path necessary
```



Massive usage of dangerous functions

- `sprintf`
- `strcpy`

No mitigation

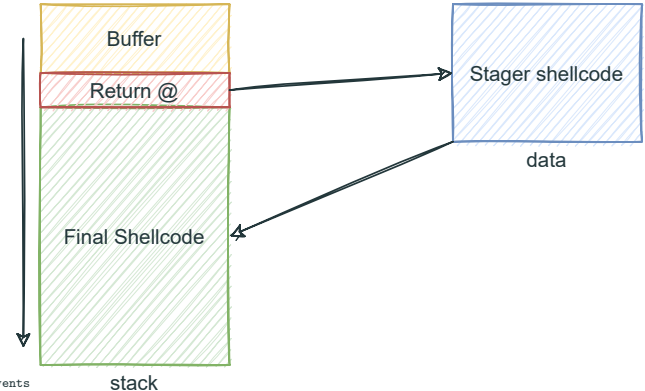
- NX
- ASLR
- Stack cookies

Target `fview` command

```
char stack_buffer[64]; // [sp+5C0h] [bp-C0h]

if ( argc > 1 && !strcmp(argv[1], "fview") )
 {
   if ( argv[2] )
   {
     sprintf(stack_buffer, "/mnt/blackbox/data/%s", (const char *)argv[2]);
     return sub_19E04(stack_buffer);
   }
   return error("filename please.\n");
 }
```

Shellcode execution

- Write a small shellcode in `data` section through command `chdir`
  - Simple `strcpy` of `argv[2]` at a fixed address in `data` section
  - Repeat command to handle null bytes
  - Small shellcode grabs `SP` value and jumps in the stack
- Put a kinda-arbitrary sized shellcode in the stack through the stack buffer overflow



```
root@debian:/home/synacktiv# python -i bb_exploit.py
[*] Run interactively with "python -i"
>>> bb_exploit_dump_users()
```

Dump i:/vol0/cfg/cfg_users_key.bin

```
0x00000000  3c bf b8 ae 1d 51 ea a8 98 2a f7 42 cb 21 21 78  <....Q...*.B.!!x
0x00000010  a6 fb 8f 98 49 a6 73 41 a1 56 db 1d 92 a4 f1 f8  ....I.sA.V......
```

IV

```
0x00000000  a7 e2 95 f6 28 a7 95 48 4c 0d 4e 76 07 04 78 0b  ....(..HL.Nv..x.
```

```
[01][03ff][Administrator] Administrator / QVW77R6R
[04][000b][UserName2] user2 / S3curePass
[03][0003][Username1] user1 / p@ssw0rd
[02][03ff][admin] admin / [redacted]
>>>
```

Reading “hardware” values

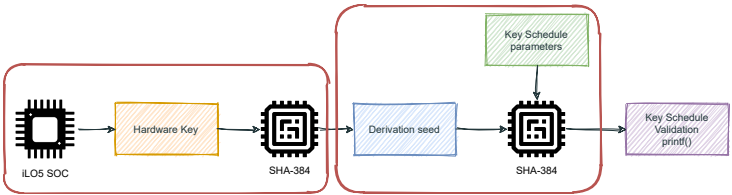
- $\Rightarrow$ Values are the same as in older firmwares

Reading “hardware” values

- ⇒ Values are the same as in older firmwares

Ripping `keymgr` code

- Execution of two blocks of code
- Comparison of the cryptoprocessor output buffer to our implementation

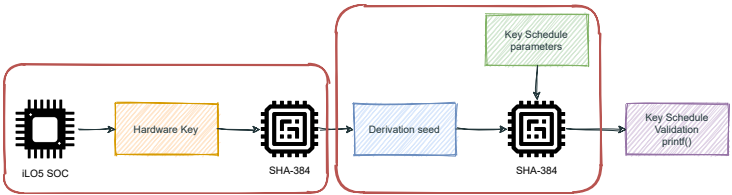


Reading “hardware” values

- $\Rightarrow$ Values are the same as in older firmwares

Ripping `keymgr` code

- Execution of two blocks of code
- Comparison of the cryptoprocessor output buffer to our implementation



Values are the same

New theory

- Our execution context is different from `keymgr`
- We have to instrument `keymgr`

Secure boot bypass

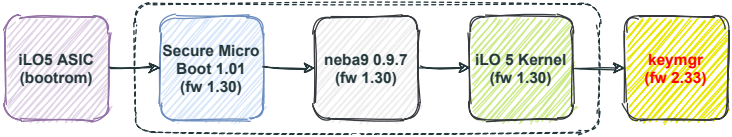
Objective(s)

- Create a modified firmware
- Add debug hooks
- Understand our mistakes
- Finally, fix our implementation

T + 6 semaines

Secure Boot bypass

- CVE-2018-7113: allows loading an unsigned userland image
- The plan: load a modified `keymgr` with an old vulnerable kernel

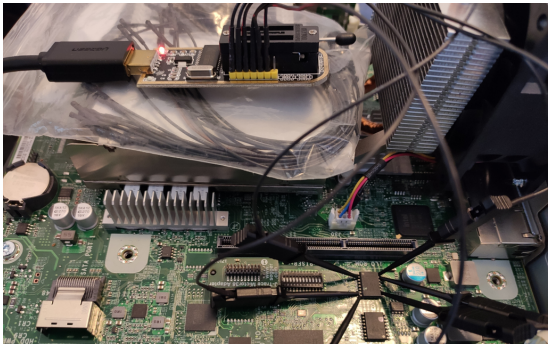


`keymgr` boots!

- Displays the same"*Key Schedule Validation*" as in a normal boot
- If we apply minor modifications, it still boots!

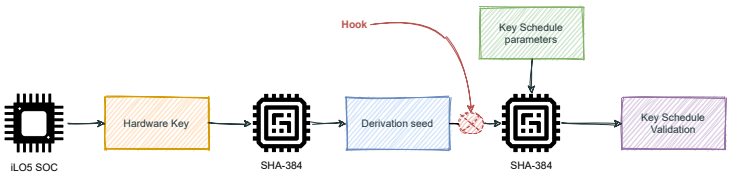
Arbitrary firmware flashing

- Either with a vulnerability: would need to reflash a valid firmware each time we want to flash a modified one
- Either through hardware: easier, faster



Hook setup

- Debugging through `printf()`
- Before the key schedule part
- Dump the value encoded as base64 on UART



- On UART:

```
Key Schedule coproc status:
vOlzOt0Taev0yrxBtI3b33av1zWCZkwtEpnCI+WFCa7cZyqm0KmQT8+sxyduqPya
```

Grabbed value

```
BC E9 73 3A DD 13 69 EB
F4 CA BC 41 B4 8D DB DF
76 AF D7 35 82 66 4C 2D
12 99 C2 23 E5 85 09 AE
DC 67 2A A6 D0 A9 90 4F
CF AC C7 27 6E A8 FC 9A
```

Our implementation

```
8F D6 EA 44 DD 13 69 EB
F4 CA BC 41 B4 8D DB DF
76 AF D7 35 82 66 4C 2D
12 99 C2 23 E5 85 09 AE
DC 67 2A A6 D0 A9 90 4F
CF AC C7 27 6E A8 FC 9A
```

Value is a SHA-384 hash. What is this magic?

Race condition

- No use of the cryptoprocessor synchonization mechanism
- The copy loop starts before the cryptoprocessor ends its computing, and starts copying an intermediary state

```
SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_DATA_END;// sha384.digest()
for ( m = 0; m < 0xC; ++m )
  *&derivation_seed[4 * m] = SHA384_DIGEST_OUTPUT[m];
*&SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;
v6 = coproc_crypto_key_schedule(derivation_seed);
```

Checking the theory

- SHA-384 implementation in Python
- Print the intermediary state before the final `.digest()`

```
>>> sha384(HardwareKey + ctx_HardwareKey[:0x30]).hexdigest()
FINAL STATE : bce9733a2cb047ecb74cd8d408507dbb7937cd5a4599ca655920362216d8a65a9
aa562b74b50e3bfdf5a26e88d15cece31574ffe3f5fa8217c9516988038505d
'8fd6ea44dd1369ebf4cabc41b48ddbdf76afd73582664c2d1299c223e58509aedc672aa6d0a990
4fcfacc7276ea8fc9a'
```

Our decryption tool works!

- Decryption of the first envelope
- Decryption of the userland image
  - 4 new tasks
  - Few differences
  - “All that for this :)”

Tool is available!

https://github.com/airbus-seclab/ilo4_toolbox

Conclusion

Objective(s)

- Enjoy the victory
- Bring balance to the Force

T + 2 months

- Recovered the firmware analysis capability
- Firmware encryption:
  - Unclear added-value for the end-user
  - Particularly complex implementation
  - No use of a secure element
  - Buggy usage of the cryptoprocessor?
- iLO5, the same intrinsic weaknesses:
  - OS primitives too permissive
  - Broken secure boot
  - Complete lack of modern mitigation techniques

- iLO5 are critical systems:
  - Patch
  - Isolate
  - Monitor
- Available fix:
  - **2.41**[6] (March 26th, 2021)
  - "*Critical - HPE requires users update to this version immediately.*"
- Security bulletin `HPESBHF04133`[7]

[6] https://support.hpe.com/hpesc/public/swd/detail?swItemId=MTX_9e149bcaae774cc190a26ea98e
[7] https://support.hpe.com/hpesc/public/docDisplay?docId=hpesbhf04133en_us

- Mark Menkhus and the Hewlett Packard Enterprise PSRT
- Our teams at **Synacktiv** and **Airbus** for their valuable feedbacks and advices
- Nicolas Iooss for his research
- Raphaël, Xavier for the open-heart surgery on the ML110 server :)
- Credits for the DOOM guy faces: `Reinchard2` @ `ZDoom` board[8]

[8] https://forum.zdoom.org/viewtopic.php?f=46&t=48921

Thank you for your attention

Questions?

To contact us:
fabien [dot] perigaud [at] synacktiv [dot] com - @0xf4b
alexandre [dot] gazet [at] airbus [dot] com
snorky [at] insomnihack [dot] net - @_Sn0rkY