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 enthusiast in a previous life
Previous work

A well-known technology:

- Deep-dive analysis of HPE iLO4 and iLO5\(^1\)
- 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](https://github.com/airbus-seclab/ilo4_toolbox)
\(^2\) Talks at RECON, SSTIC, ZeroNights
\(^3\) [https://github.com/Synacktiv/pcileech_hpiolo4_service](https://github.com/Synacktiv/pcileech_hpiolo4_service)
How it started

- 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

#BHUSA @BlackHatEvents
Where is the encryption implemented?

Boot chain?

- Bootloaders and kernel stay the same between versions 1.3x and 1.4x
Where is the encryption implemented?

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
- PEM base64, PKCS#8
- The RSA private key is protected by a passphrase
- OpenSSL: PEM_read_bio_RSAPrivateKey

⇒ a callback function provides the passphrase
When the passphrase meets the hardware

- 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

```c
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[0x1F200DB];
    HW_SECRET[1] = MEMORY[0x1F200B0];
    HW_SECRET[2] = MEMORY[0x1F200B0] & 0xFFFFFFFF;
    HW_SECRET[3] = MEMORY[0x1F200BC];
    HW_SECRET[4] = MEMORY[0x1F21B0];
    HW_SECRET[5] = MEMORY[0x1F2182];
    HW_SECRET[6] = MEMORY[0x1F21840];
    HW_SECRET[7] = MEMORY[0x1F21800];
    for ( i = 0; i < 0x20; ++i )
        buf[i] = *((BYTE *)key_mask + i) ^ *((BYTE *)HW_SECRET + i);
    return 0x20;
}
```
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
• 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

@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
Introducing on-request memory mapping

```c
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

```plaintext
.fum.elf:RW:00062AF0 ; MR_RANGE MR_MAPPING
.fum.elf:RW:00062AF0 00 00 00 80+MR_MAPPNG MR_RANGE <0x80000000, 0x1F000000, 1, 0, 0>
.fum.elf:RW:00062AF0 00 00 F0 01+ ; DATA XREF: memmap_init:loc_2EA48t;
.fum.elf:RW:00062AF0 01 00 00 00+ ; memmap_init:loc_2EA70t ...
.fum.elf:RW:00062B08 00 F0 0E 80+ MR_RANGE <0x800EF000, 0x1F010000, 2, 0, 0>
.fum.elf:RW:00062B20 00 00 0F 80+ MR_RANGE <0x800F0000, 0x1F020000, 4, 0, 0>
.fum.elf:RW:00062B38 00 00 1F 80+ MR_RANGE <0x801F0000, 0x1F030000, 8, 0, 0>
```

---

#BHUSA @BlackHatEvents
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 :)

```c
struct MEM_RANGE
{
    void *phys_addr;
    void *virt_addr;
    unsigned __int64 mask;
    int field_10;
    int field_14;
};
```
Double mapping for the win

Single-byte memory patch

- From:

  MR_RANGE <
  0xC0000000, 0x1F200000, 
  0x800000000, 0, 0>

- To:

  MR_RANGE <
  0xC0000000, 0x1F210000, 
  0x800000000, 0, 0>
I love it when a plan comes together

[+] 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!!!**

#BHUSA @BlackHatEvents
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) neba9 0.10.13, type 0x01, size 0x00005644
4) neb926 0.3, type 0x02, size 0x00000ad0
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
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

- iLO5 ASIC (bootrom)
- Secure Micro Boot 2.02
- neba9 0.10.13
- iLO 5 Kernel 00.09.60
- stager (keymgr) 2.10.54
- secure (type 0x23) 2.10.54

Load and decrypt
We are looking for the Keymaker
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)
Private key reconstruction

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
“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
  ⇒ 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
Cryptoprocessor at a glance

- **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
Communication with the cryptoprocessor

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)] )
Dropping a “secret”

SHA-512

iLO5 SOC

Boot Hash

Hardware Key

SHA-384

Derivation seed
Dropping a “secret”

- SHA-512
- iLO5 SOC

Boot Hash

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Flags</th>
<th>Input</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>START</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x280 (0x50 octets)</td>
<td>DATA_END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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
printf() is our friend

Debug messages

- Intermediary state

```c
  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

```c
  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);
```
Where’s Waldo?

**Goal**

- Debug information is printed on the UART output
- Find UART on the Microserver
Yet another failure

At boot time

Key Schedule Validation: 103wN50aDle9gyhfEJShR5jv0sKB0tfT25uk2U/vjxRA
-----BEGIN PUBLIC KEY-----
MHYwEAYHkoZIzj0CAQYFK4EEACIDYgAE4StRIN6NFi6X000aNMLePDm0mYmXIpDf
03KrjkhWjZW8z3QNeyUXVNxHayZEKFL6Xk6vjkYYeJNdqg9yEzI0a2GK2emSgp4D
RNgUyUpix0jq5+1luKXWUyFQ6rBJ45Dr
-----END PUBLIC KEY-----

In our implementation

Key Schedule Validation: enRnbrYwZKTBjF955yi45Vibe4ROBE4g1E05d0rYcVl

- 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
A strange port

On Microserver Gen10 motherboard

- Spotted on HPE website
- MICTOR port
- Should allow JTAG debugging
In the end, we bought a correct adapter :)

#BHUSA @BlackHatEvents
Results

JTAG enumeration

Device ID #1: 0101 1011101000000000 010001110111 (0x5BA00477)
Device ID #2: 0000 0111100100100110 111100001111 (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:

```plaintext
bb fdump (file)  — Hex dump 'file'. Absolute path necessary
```
90’s want their vulnerabilities back!

Massive usage of dangerous functions

- `sprintf`
- `strcpy`
Easy game

No mitigation

- NX
- ASLR
- Stack cookies

Target `fview` command

```c
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");
}  
```
Exploitation plan

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
A little disgression

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>3c bf b8 ae 1d 51 ea a8 98 2a f7 42 cb 21 21 78</td>
</tr>
<tr>
<td>0x00000010</td>
<td>a6 fb 8f 98 49 a6 73 41 a1 56 db 1d 92 a4 f1 f8</td>
</tr>
</tbody>
</table>

<....Q....*.B!!x |
| ....I.sA.V.......

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>a7 e2 95 f6 28 a7 95 48 4c 0d 4e 76 07 04 78 0b</td>
</tr>
</tbody>
</table>

....(..HL.Nv..x.

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

>>>
Reading “hardware” values

- ⇒ Values are the same as in older firmwares
Back to business

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

- 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
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!
Hardware hackers

**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:

v0lzo0Taev0yrxBtI3b33av1zWClwCkttEpnCI+WFCa7cZyq0mQT8+sxyduqPya
**Comparison**

<table>
<thead>
<tr>
<th>Grabbed value</th>
<th>Our implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC E9 73 3A DD 13 69 EB</td>
<td>8F D6 EA 44 DD 13 69 EB</td>
</tr>
<tr>
<td>F4 CA BC 41 B4 8D DB DF</td>
<td>F4 CA BC 41 B4 8D DB DF</td>
</tr>
<tr>
<td>76 AF D7 35 82 66 4C 2D</td>
<td>76 AF D7 35 82 66 4C 2D</td>
</tr>
<tr>
<td>12 99 C2 23 E5 85 09 AE</td>
<td>12 99 C2 23 E5 85 09 AE</td>
</tr>
<tr>
<td>DC 67 2A A6 D0 A9 90 4F</td>
<td>DC 67 2A A6 D0 A9 90 4F</td>
</tr>
<tr>
<td>CF AC C7 27 6E A8 FC 9A</td>
<td>CF AC C7 27 6E A8 FC 9A</td>
</tr>
</tbody>
</table>

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

Race condition

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

Checking the theory

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

```python
>>> sha384(HardwareKey + ctx_HardwareKey[:0x30]).hexdigest()
FINAL STATE : bce9733a2cb047ecb74cd8d408507dbb7937cd5a4599ca655920362216d8a65a9
aa562b74b50e3bfdf5a26e88d15cece31574ffe3f5fa8217c9516988038505d
'8fd6ea44dd1369ebf4cabc41b48ddbf76afd73582664c2d1299c223e58509aedc672aa6d0a990
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
Brace yourselves

- 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/hpsc/public/swd/detail?swItemId=MTX_9e149bcaae774cc190a26ea98e](https://support.hpe.com/hpsc/public/swd/detail?swItemId=MTX_9e149bcaae774cc190a26ea98e)

\(^7\) [https://support.hpe.com/hpsc/public/docDisplay?docId=hpeshf04133en_us](https://support.hpe.com/hpsc/public/docDisplay?docId=hpeshf04133en_us)
Greetings!!!!

- 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

---

8https://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