HPE iLO 5 security: Go home cryptoprocessor, you're drunk!

Alexandre Gazet[1], Fabien Périgaud[2], and Joffrey Czarny
alexandre.gazet@airbus.com
fabien.perigaud@synacktiv.com
snorky@insomnihack.net

[1] Airbus
[2] Synacktiv

Abstract. At the core of HPE Gen10 servers lies the Integrated Lights Out 5 (iLO 5) technology. This new revision (hardware and software) of the remote management technology introduced a key feature built into the hardware and described as a silicon root of trust.

Our previous studies [4, 6, 10] of the technology highlighted a critical flaw in the secure boot process, allowing us to load a rogue userland applicative image.

At the start of 2020, we observed that new HPE iLO5 firmware (versions 2.x) would come as encrypted binary blobs. In times where supply chain and platform security are more exposed than ever, we decided to review the security implications of the new firmware packaging.

In this paper we propose:
— A walkthrough on how to approach encrypted firmware updates
— A complete description and analysis of the new encryption mechanism, including hardware resources (cryptographic coprocessor)
— A new take on the Gen10 server hardware, resulting from our discovery of a debug port on the server boards
— A demonstration that Frankenstein firmware and supply chain attacks are still possible
— Lessons learnt about the implementation: strengths/weaknesses as well as mistakes that were made

All the results presented in this paper have been obtained with the following hardware platforms:
— HPE ProLiant ML110 Gen10 (platform id `0x020b`)
— MicroServer Gen10 Plus (platform id `0x0222`)

1 Transitioning to new firmware

With the introduction of iLO5 version 2.x firmware, we noticed that our previously developed tooling [5] had become ineffective and firmware extraction was not possible anymore.

With further scrutiny one could notice that the new firmware files were mostly high entropy blobs which suggested the introduction of an encryption overlay. Besides, according to the installation instructions of the 2.10 firmware, "*upgrading to iLO 5 version 2.10 is supported on servers with iLO 5 1.4x or later installed.*"". One could reasonably assume that a change to support the encrypted firmware was introduced in versions 1.4x.

Thus, we decided to start our analysis with these intermediary/transition versions (1.4x). The question was thus to discover how the firmware had evolved between firmware 1.3x, and 1.4x versions.

1.1 Understanding the changes

As a reminder, the boot chain of an iLO5 system is made of five components, as shown on figure 1:

1. a bootrom stored on the ASIC itself
2. a first-stage bootloader, `Secure Micro Boot`
3. a second-stage bootloader, `neba9`
4. a real-time operating system, `Integrity` OS
5. an `Integrity` userland image

Fig. 1. HPE iLO5 1.x boot chain

Diffing the structure of firmware 1.39 and 1.48, one could observe that the bootloaders were unchanged. Thus the modification lies either in the kernel, or in the userland applicative image.

1.2 Firmware Update Manager (`fum`)

The next logical target to check was the Firmware Update Manager (`fum`) task. This task handles most of the heavy lifting associated with firmware updates:

— check of the update target: `iLO` chip, `CPLD`, Innovation or Management engines, etc.

— check of the integrity of the update blob (cryptographic signature)
— writing into persistent storage media (SPI flash for example)
— etc.

Based on our previous knowledge of this component, we quickly noticed a modification in the code path that handles the `iLO` component update. One of the most noticeable artefacts is the presence of an encrypted `RSA` private key in the data of the task, as shown in listing 1.

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIJrTBXBgkqhkiG9w0BBQ0wSjApBgkqhkiG9w0BBQwwHAQIjhdNQLNz8zoCAggA
MAwGCCqGSIb3DQIKBQAwHQYJYIZIAWUDBAEqBBAEJcEYMX1ZMZFCj4/IdBSrBIIJ
[...]
xVKQ5YnhRsZnhe70T3pncVbMg+ZA3ai8urNxUrN7OhPeP2nicxlndWtqadrus/R5
meRspuW0AKzWAQN3EDse3SzlYTwGd6Jvb6ms/BqGrxvt
-----END ENCRYPTED PRIVATE KEY-----
```

Listing 1. New encrypted `RSA` private key

The newly introduced code heavily relies upon OpenSSL primitives. The decryption process can be summarized as exposed in listing 2.

The encryption of the firmware blob is based on a symmetric-key cipher, `AES256`, used with an authenticated mode, Galois/Counter Mode (`GCM`). The two functions `PEM_read_bio_RSAPrivateKey` and `EVP_OpenInit` are instrumental here. Their prototype is shown in listings 3 and 4; from its man page: "*EVP_OpenInit() initializes a cipher context ctx for decryption with cipher type. It decrypts the encrypted symmetric key of length ekl bytes passed in the ek parameter using the private key priv. The IV is supplied in the iv parameter.*".

Using this knowledge, one can infer the layout of the new encrypted firmware file format, as shown in figure 2:

— Bytes `0x0-0x200` (4096 bits) are the AES symmetric key, encrypted with the public part of the encrypted `RSA` private key found in the data of the task. This AES key is automatically decrypted by the high-level OpenSSL `EVP_OpenInit` function (envelope decryption).
— Bytes `0x200-0x20C` (96 bits) are used as an initialization vector for the AES256-GCM cipher.
— The last 0x10 bytes (128 bits) of the firmware blob are the AES GCM tag. The purpose of an authenticated mode is to enforce that the encrypted data has not been tampered with before the decrypted data is actually used.

```
bio_buffer = BIO_new_mem_buf(RSA_PRIVATE_KEY, -1);
rsa_key = PEM_read_bio_RSAPrivateKey(bio_buffer, 0,
    pem_password_cb, 0);
pkey = EVP_PKEY_new();
```

```
EVP_PKEY_assign_RSA(pkey, rsa_key);
evp_cipher_ctx = EVP_CIPHER_CTX_new();
cipher_type = EVP_aes_256_gcm();

EVP_OpenInit(evp_cipher_ctx, cipher_type, pbFirmware, 0x200,
    pbFirmware + 0x200, pkey);
EVP_CIPHER_CTX_ctrl(evp_cipher_ctx, EVP_CTRL_GCM_SET_TAG, 0x10,
  &pbFirmware[*pcbFirmwaresize_ - 0x10]);
EVP_DecryptUpdate(evp_cipher_ctx, pbFirmware, &outl,
    pbFirmware + 0x20C, *pcbFirmwaresize_ - 0x21C);
*pcbFirmwaresize_ = outl;

EVP_DecryptFinal(evp_cipher_ctx, &pbFirmware[outl], &outl);
*pcbFirmwaresize_ += outl;
```

Listing 2. Firmware decryption pseudo-code

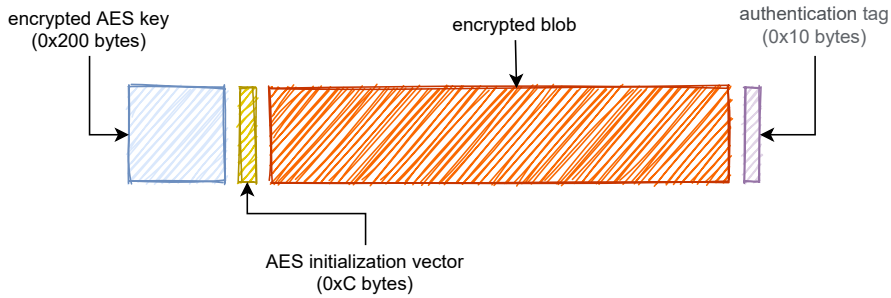

Fig. 2. HPE iLO5 encrypted firmware layout

```
int EVP_OpenInit(EVP_CIPHER_CTX *ctx, EVP_CIPHER *type,
    unsigned char *ek, int ekl, unsigned char *iv,
    EVP_PKEY *priv);
```

Listing 3. PEM_read_bio_RSAPrivateKey

```
RSA *PEM_read_bio_RSAPrivateKey(BIO *bp, RSA **x,
    pem_password_cb *cb, void *u);
```

Listing 4. PEM_read_bio_RSAPrivateKey

To "unlock" the use of the RSA private key, a callback function is passed to PEM_read_bio_RSAPrivateKey. This callback function fills out a buffer with the correct passphrase. The implementation in fum is shown in listing 5.

```
int __fastcall pem_password_cb(char *passphrase, int size, int
    rwflag, void *u)
{
  unsigned int i;
  _DWORD key_mask[8];
  _DWORD HW_SECRET[16];

  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 )
    passphrase[i] = key_mask[i] ^ HW_SECRET[i];
  return 0x20;
}
```

Listing 5. `PEM_read_bio_RSAPrivateKey` passphrase callback

`pem_password_cb` fills in a buffer of eight 32-bit words, using memory mapped values (from `0x1F20XXX` and `0x1F21XXX` addresses) and a static xor mask. Interestingly, these memory regions are not part of the sections of the `fum` task.

As shown in listing 6, some information regarding the physical to virtual memory mapping of the task are present within the binary itself (more on this later). According to this extract a 4KiB page of physical memory located at `0xC0000000` is mapped at virtual address `0x1F20000`; `0xC0001000` at virtual address `0x1F21000`.

According to our previous works, physical memory region `0xC0000000` and `0xC0001000` actually refers to internal memory of the system-on-chip (SOC) itself.

```
[...]
MR_RANGE <0x83000000,  0x600000, 0x0400000000, 0, 0>
MR_RANGE <0xC0000000, 0x1F20000, 0x0800000000, 0, 0>
MR_RANGE <0xC0001000, 0x1F21000, 0x1000000000, 0, 0>
MR_RANGE <0xC0002000, 0x1F22000, 0x2000000000, 0, 0>
MR_RANGE <0xC0008000, 0x1F23000, 0x4000000000, 0, 0>
```

```
[...]
```

Listing 6. `fum` physical-virtual memory mapping extract

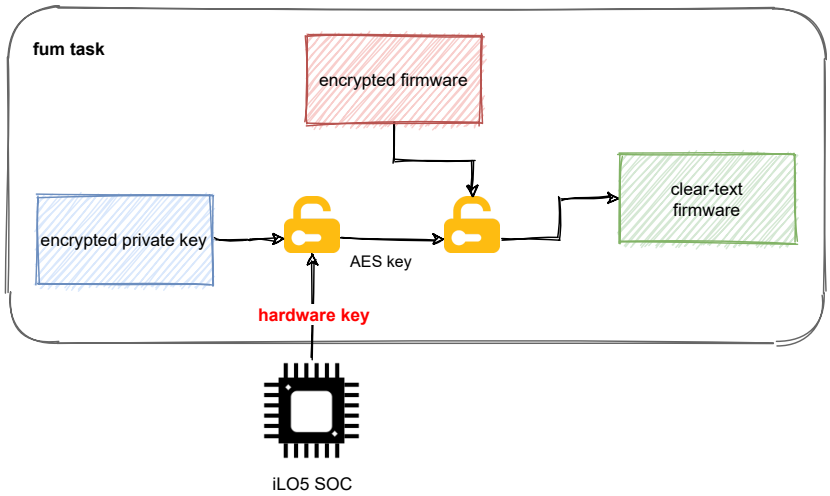

Fig. 3. HPE iLO5 firmware envelope encryption summary

To wrap it up, the `1.4x` firmware, and more specifically the `fum` task, derives a passphrase from an hardware key stored in the SOC, to unlock the cryptographic material needed to decrypt (AES256-GCM) `2.x` encrypted firmware update blobs. A summary is shown in figure 3. Now the question is, how to extract this hardware key?

1.3 Extracting the hardware key from the SOC

In 2018, our esteemed peer Nicolas Iooss reported `CVE-2018-7105` [11] to HPE, who subsequently released the security bulletin `HPESBHF03866` [2]. This format-string based vulnerability in the proprietary SSH restricted shell, post-authentication, offers a memory read and write primitive in the context of the SSH task (known as `ConAppCli`). HPE Integrated Lights-Out 3, 4 and 5 were impacted.

Nicolas released a powerful proof of concept exploitation code for vulnerable `iLO 4` systems. Thankfully, our faithful HPE ProLiant ML110 Gen10 server was still hanging around in our lab, running a vulnerable `iLO5` firmware version. Thus, we decided to port the first stage of that exploit to `iLO 5` systems and quickly obtained a functional memory read

and write primitive. Due to our major laziness and to the nature of the vulnerability, we had to cope with some limitations: null bytes and a couple of escape characters are forbidden in the target address of the primitive.

So far we know that a secret hardware key is read by the `fum` task. Still, all we have is a read/write primitive in the `ConAppCli` task (SSH).

1.4 The kernel is your friend

As briefly introduced previously, the tasks themselves embed in their data section some partial information regarding the physical to virtual address mapping. The userland tasks have the ability to request the kernel to map into their memory space some pre-defined memory resources.

The array of pre-defined memory resources is shown in listing 7. It is shared by all the tasks. Each entry is a `MEM_RANGE` (as shown in 9) structure describing a 4KiB page of physical memory, associated to its virtual address (in green) and to a 64-bit mask (in red).

To map a memory resource, a task calls the `memmap` function; its prototype is given in listing 8. The function takes a bit mask as single argument. Internally, it walks through the list of all the memory resources, if the mask argument matches the mask of an entry then this entry is mapped by the API with support from the kernel.

```
.ConAppCLI.elf.RW:000B7E20 ; MEM_RANGE MEM_RANGES[]
MEM_RANGE <0x80000000, 0x1F00000, 1, 0, 0>
MEM_RANGE <0x800EF000, 0x1F01000, 2, 0, 0>
MEM_RANGE <0x800F0000, 0x1F02000, 4, 0, 0>
[...]
MEM_RANGE <0xC0000000, 0x1F20000, 0x0800000000, 0, 0>
MEM_RANGE <0xC0001000, 0x1F21000, 0x1000000000, 0, 0>
MEM_RANGE <0xC0002000, 0x1F22000, 0x2000000000, 0, 0>
[...]
MEM_RANGE <0xC0011000, 0x1F2D000, 0x1000000000000, 0, 0>
MEM_RANGE <0>
.ConAppCLI.elf.RW:000B82D0
```

Listing 7. Memory resources description in `fum`

```
int __fastcall memmap(__int64 mask)
```

Listing 8. `memmap` function

```
struct MEM_RANGE
{
  void *phys_addr;
```

```
  void *virt_addr;
  unsigned __int64 mask;
  int field_10;
  int field_14;
};
```

Listing 9. `MEM_RANGE` structure definition

As shown in listing 10, we noticed that a function in `ConAppCli` makes use of the `memmap` function to map the lower addresses of the SOC (i.e. `0xC0000XXX`, bit mask `0x800000000`). However this function is only called when the task is exiting. It means that during most of the life-cycle of this task, the SOC region is not mapped.

```
int __cdecl timer_func()
{
    int res;

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

Listing 10. `timer_func` in `fum`

To call this function at our discretion, we reused a trick from Nicolas' exploit. The internal structures describing the commands of the restricted shell are located in a read/write section, as shown in listing 11. Without too much detail, it means one can modify the function pointer of a command's handler. Using the read/write primitive given by the format string vulnerability, we overwrite the handler of the `system1` object to call the timer function instead.

As a result, sending the command "`show /system1`" to the restricted shell will trigger the execution of the `timer_func` and ultimately the mapping of the SOC range in the `ConAppCli` task.

```
ConAppCLI.elf.RW:000B0BB0 VTABLE_SYSTEM_NAME
OBJECT_VTABLE <System_nameShow, System_nameShow,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>
.ConAppCLI.elf.RW:000B0BEC VTABLE_SYSTEM_NUMBER
OBJECT_VTABLE <System_numberShow, System_numberShow,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>
```

Listing 11. Restricted shell command definition structures in `ConAppCli`

A couple of extra tricks helped us to get the full potential of this exploit. Again we abuse the fact that the pre-defined memory resource structures

are located in a writeable section, and the fact the kernel honours our requests (with some limitations).

— **Problem**: Mapping the `0xC0000000` physical addresses range is easy, however due to the format string limitation, reading virtual addresses in the form `0x1F200xx` is impossible due to the null byte.
Solution: One can update the virtual address of the region entry to `0x1F21000`, thus using some sort of double mapping, to ease our read through the format string exploit.

— **Problem**: How to map the range `0xC0001000`?
Solution: The timer function calls `memmap` with a mask set to `0x800000000`. One can update the mask of this entry to `0x1800000000`, thus it will match and be mapped as well.

Putting all the pieces together, we use our Python implementation to remotely exploit the format string vulnerability, map the SOC into `ConAppCli`, and read the hardware key from the SOC (as shown in listing 12).

We then use this key to build a decryptor based on the same OpenSSL primitives.

```
[+] dumping iLO HW keys:
[+] MMU: memory mapping magic:
  >> patch_addr post 0x2008@0xb8264
  >> patch_addr post 0x1008@0xb824c
  >> patch_addr post 0x18@0xb818c
  >> patch_addr post 0xc00000@0xb8181
[+] command hooks:
  >> hook_addr post 0x70158@0xb0bec
  >> hook_addr post 0x70158@0xb0bb4

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

Listing 12. Reading SOC memory through a format string

1.5 We are looking for the Keymaker

Applying our tooling on the newly decrypted 2.x firmware, such as the 2.10 shown in listing 13, led to puzzling results.

```
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 0x004e7f28
```

Listing 13. HPE iLO5 2x new firmware structure

Overall the firmware structure is quite similar to 1.x firmware. However, there are now 3 different `Integrity` userland applicative images (index `0x8`, `0x9` and `0xa`).

— Type `0x20` used to be the main image, however here its size is surprisingly small.
— Type `0x22` is the recovery image.
— Type `0x23` is unknown at this point. Besides, once again looking at its content, it is a high entropy blob of seemingly encrypted data.

What is in the first image (type `0x20`)? The dissection of this `Integrity` image is shown in listing 14. It is made of a single task named `keymgr`.

```
-----------------[ Sections List ]-----------------

> name: .secinfo, 0x6e000, 0x260 bytes
> 0x0000 -                    .keymgr.elf.RO
> 0x0001 -                    .keymgr.elf.RW
> 0x0002 -                   .keymgr.elf.RW2
> 0x0003 -               .libINTEGRITY.so.RO
> 0x0004 -               .libINTEGRITY.so.RW
> 0x0005 -                       .libc.so.RW
> 0x0006 -                      .libc.so.RW2
> 0x0007 -                  .libopenssl.so.RW
> 0x0008 -                       .libc.so.RO
> 0x0009 -                 .libopenssl.so.RW2
> 0x000a -                  .km.Initial.stack
> 0x000b -                        .boottable
> 0x000c -                  .libopenssl.so.RO
> 0x000d -                           .km.heap
> 0x000e -                           .secinfo

-----------------[ 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
```

Listing 14. `keymgr` single task image

After the extraction of the first hardware key we were confident we had successfully removed the encryption. Still, it looks like the princess is in another castle again.

At a high level the image type `0x20`, or `keymgr` as it is mono-tasked, is a stager, responsible for loading and decrypting the secure, full, encrypted applicative image, as shown in figure 4.

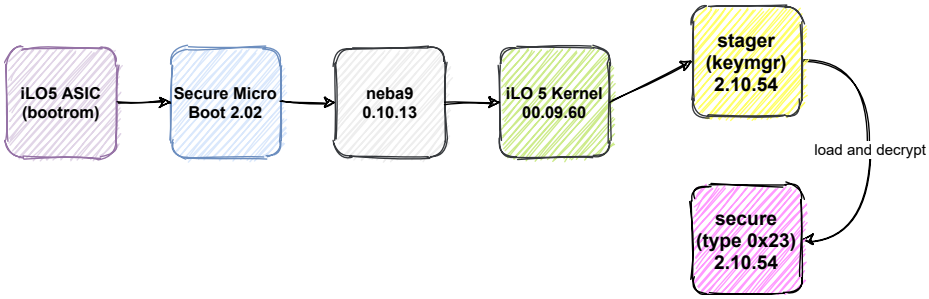

Fig. 4. HPE iLO5 2.x boot chain

We fell down the rabbit hole, again. Though, this first part was the easy one, most of it was over in a week or so.

2 `keymgr` as secure loader

This section provides a detailed analysis of the `keymgr` secure loader.

2.1 Hardware anchorage attempt

In its early initialization steps, `keymgr` calls the `map_mr_obj` function, as shown in listing 15. At a functional level, this function is very similar to the `memmap` function seen in the previous section.

```
map_mr_obj("MRC0000", 0x2A);
map_mr_obj("MRC0001", 0x2B);
map_mr_obj("MRC0030", 0x33);
map_mr_obj("MRC0032", 0x34);
```

Listing 15. Memory resource object mapping in `keymgr`

Instead of using a mask to identify the memory region resource, it uses the name of the resource object (alongside its id). For example, "MRC0000" instructs the kernel to map, in the keymgr virtual address space, the physical memory page starting at 0xC0000000, "MRC0032" 0xC0032000, and so on.

0xC0000000 and 0xC0001000 refer to SOC region. 0xC0030000 and 0xC0032000 also are old acquaintances, they refer to a cryptographic coprocessor (later called cryptoprocessor) introduced with iLO5 generation (most probably located on the SOC as well).

To the best of our knowledge, the cryptoprocessor is undocumented at the time of this study. These components are usually provided by the SOC vendors, as security IP modules directly integrated within the SOC hardware (as for example the ARM CryptoCell module [1]). Access to developer/technical documentation for this sort of component is most often subject to non disclosure agreements (NDA) with the owner of the IP block.

In our situation, we could not identify the module used by the iLO SOC, and thus had no prior knowledge about its capabilities, hardware interfaces, API, etc. Our only guess was that the use of a cryptoprocessor by keymgr, basically a key derivation function relying on a cryptoprocessor, was a specific development from HPE. A couple of hints can be found in some paths embedded in the binary as compilation artefacts, as shown in listing 16.

```
ROM:0001155C     00000013     C    src/local_crypto.c
ROM:0001420C     00000011     C    src/aes_engine.c
```

Listing 16. Compilation artefacts in keymgr

The binaries being fully stripped of symbols, all the descriptions, variables, constants and function names, etc. given below are based on our own analysis and experiments with the cryptoprocessor. As a quick note on the methodology, to identify the primitives exposed by the cryptoprocessor we mainly relied upon:

— the interface with OpenSSL functions. The lack of symbols can be partially overcome by using the error log feature which gives a line number and a source file, thus allowing quick identification of the function.
— analogy with the outer layer of encryption (eg. use of AES-GCM)
— size input/output buffers passed to cryptographic algorithm

keymgr takes advantage of three different primitives of the cryptoprocessor:

— `SHA384` hash function
— `AES256` symmetric block cipher, used in `CTR` mode
— `AES256` symmetric block cipher, used in `GCM` mode

`SHA384` primitive The `SHA384` digest primitive is the first one being used. It is located at physical addresses `0xC0032000`. In `keymgr` it is mapped at `0x1F2A000`. The cryptoprocessor actually offers many channels in parallel. An overview is shown in figure 5.

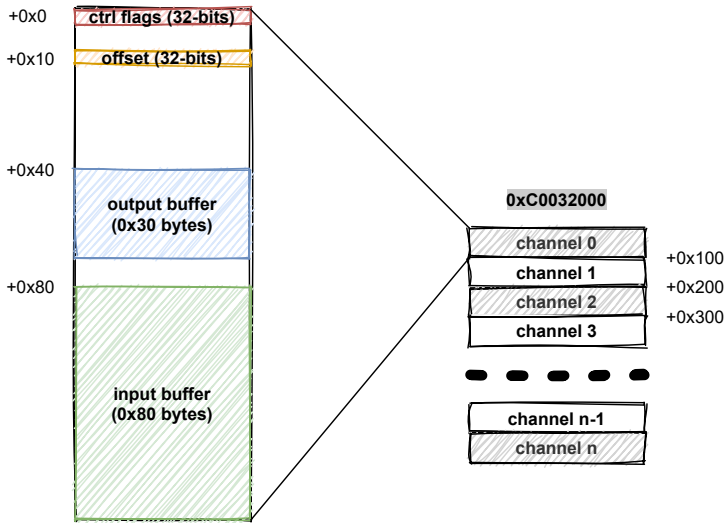

Fig. 5. Cryptoprocessor digest engines

— **`Ctrl`**: this register controls the configuration and operations of the cryptoprocessor. It is also updated to reflect the state of the cryptoprocessor.
— **`Offset`**: this register contains the number of bits written into the input buffer since the last reset of the channel. It is automatically updated on every write to the input buffer.
— **`Output buffer`**: this buffer is updated by the cryptoprocessor with the result of the digest operation. The size of the output depends upon the configuration of the channel. In `keymgr`, the cryptoprocessor is configured to compute a `SHA384` digest (`0x30` bytes).
— **`Input buffer`**: this buffer is updated by the client application to feed the cryptoprocessor with new input. The size of the hardware buffer is `0x80` bytes.

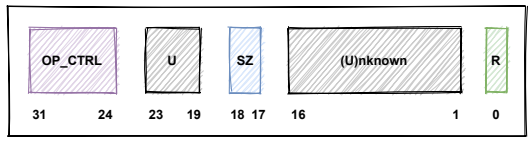

Fig. 6. Digest channel configuration register details

A more detailed description of the control register is shown in figure 6.

— `R` (bit 0): is a synchronization flag. It indicates that the output is ready.
— `SZ` (bits 17-18): this field is used to configure the digest size (e.g. `SHA256`, `SHA384`, `SHA512`)
— `OP_CTRL` (bits 24-32): this field controls the operations of the channel.

The following flags have been understood for the most significant byte (`OP_CTRL`)

— `0x80` (bit 7): Setting the most significant bit to 1, resets the state of the channel. Once written, the client application may wait for its state to flip. This synchronization mechanism signal the readiness of the cryptoprocessor.
— `0x01` (bit 0): is set to initialize a new digest, as shown in listing 17. The `coproc_crypto_channel_init` function is found in the kernel.
— `0x02` (bit 1): semantics is unclear, it seems to be used to indicate that more data is expected.
— `0x04` (bit 2): is set to finalize a digest, as shown in listing 18. The `coproc_crypto_channel_finalize` function is found in the kernel.

From listing 18, one also learns that bit 0 of the control register is used as a synchronization signal indicating that the output of the cryptoprocessor operation is available.

```
void __fastcall coproc_crypto_channel_init(int channel, int
    conf_flag)
{
  unsigned int *ptr_flags;
  unsigned int new_flags;

  ptr_flags = ((channel << 8) | 0xC0032000);
  *ptr_flags = 0x80000000;
  new_flags = conf_flag << 17;
  while ( (*ptr_flags & 0x80000000) != 0 )
    ;
  *ptr_flags = new_flags;
  *ptr_flags = new_flags | 0x1000000;
```

```
}
```

Listing 17. `coproc_crypto_channel_init` function

```
void __fastcall coproc_crypto_channel_finalize(int channel, int *
    digest_out)
{
  _BYTE *ptr_flags;
  int new_flags;

  ptr_flags = ((channel << 8) | 0xC0032000);
  new_flags = *ptr_flags | 0x4000000;
  *ptr_flags = new_flags;
  while ( (*ptr_flags & 1) != 0 )
    ;
  memcopy(digest_out, ptr_flags + 0x40, 0x40 - ((new_flags & 0
      x60000u) >> 13));
}
```

Listing 18. `coproc_crypto_channel_finalize` function

Early digest and cryptographic material seeding With this knowledge in mind, one can analyse the `coproc_crypto_initialize` function from `keymgr`. It is called very early; right after the memory resources creation.

For an easier understanding, we have split the function into three parts.

Part 1, as shown in listing 19, is simple; however, it is also quite challenging. It starts by copying twelve 32-bit words from the cryptoprocessor output buffer (on channel 0) into a local buffer (thus `0x30` bytes, 384 bits). At first, this is puzzling as it is the very first interaction of the task with the cryptoprocessor.

Besides, after inspection of the bootloaders and kernel code, no other interaction with the cryptoprocessor channel 0 could be found. The answer to this enigma was found by observing the counter register of the channel 0. Indeed the counter indicates that `0x8000` bytes were processed.

An educated guess could tell us that `0x8000` is the size of the first bootloader (`BL0`). Thus, we formulated the hypothesis that the output buffer contains an artefact from the bootrom, which computed the digest of the first bootloader, in the early steps of the boot process, during the validation of the cryptographic signature of `BL0` by the bootrom (RSA4096 key). The hypothesis was later validated by comparing the value of the output buffer with the actual SHA512 digest of the last `0x8000` bytes of the firmware update file.

`coproc_crypto_initialize` then call `coproc_crypto_KAT` which performs a self-assessment of the cryptoprocessor (AES and SHA384 primitives), using test vectors or known-answer tests (KAT).

Interestingly, at the end of this part, the value of the saved digest is written back in the input buffer of the cryptoprocessor after the initialization of a new digest transaction.

One can observe that the flag `SHA384_DIGEST_MORE_DATA` (`0x2000000`) is set. Still, experimentally we observed that the value of the bytes written before this flag is set does not influence the final digest value.

```
for ( i = 0; i < 0xC; ++i )
    coproc_state[j] = SHA384_DIGEST_OUTPUT[i];

if ( !coproc_crypto_KAT() )
return 0;

SHA384_DIGEST_FLAGS = 0x80000000;
SHA384_DIGEST_FLAGS = 0x1020000;  // config & start

for ( j = 0; j < 0xC; ++j )
{
    SHA384_DIGEST_INPUT[j] = coproc_state[j];
    coproc_state[j] = 0;
}

SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_MORE_DATA;
```

Listing 19. `coproc_crypto_initialize` - part 1

Besides, a key observation here can be made by comparing the code with the one in function `coproc_crypto_channel_init` (listing 17): the synchronization mechanisms of the cryptoprocessor are ignored. Here the most significant bit (msb) of the control register is not properly checked. After writing `0x80000000`, setting the msb to `1`, one should wait for and verify that the msb is reset by the cryptoprocessor, indicating that it is ready to accept further configuration and operations.

Part 2, as shown in listing 20, fills the input buffer (`0x80` bytes) with values from the SOC (`0x1F2xxxx`) and zeroes. Most of the SOC values are also used in the computations of the passphrase of the RSA key.

```
SHA384_DIGEST_INPUT[0] = 0;
SHA384_DIGEST_INPUT[1] = dword_1F200A0 & 0xFFFF0000;
casted_val = dword_1F200AC;
SHA384_DIGEST_INPUT[2] = casted_val;
SHA384_DIGEST_INPUT[3] = casted_val;
byte_1F20AE0 = 0x10;
SHA384_DIGEST_INPUT[4] = casted_val;
SHA384_DIGEST_INPUT[5] = (dword_1F200D8 | 0x2000000) & 0xFFFFFFDF;
SHA384_DIGEST_INPUT[6] = dword_1F20B00;
```

```
SHA384_DIGEST_INPUT[7] = dword_1F20B08 & 0xFFFFFFF0;
v4 = dword_1F20B0C;
SHA384_DIGEST_INPUT[8] = dword_1F20B0C

for ( k = 9; k < 0x1F; ++k )
    SHA384_DIGEST_INPUT[k] = 0;

SHA384_DIGEST_INPUT_END = v4;
SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_DATA_END;

for ( l = 0; l < 0xC; ++l )
    coproc_state[l] = SHA384_DIGEST_OUTPUT[l];
SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;
```

Listing 20. `coproc_crypto_initialize` - part 2

As soon as the `SHA384_DIGEST_DATA_END` flag (0x4) is written in the most significant byte of the control register, `keymgr` reads the content of the output buffer and saves it into a the `coproc_state`. Again, as we saw for the initialisation phase, the synchronization mechanisms of the cryptoprocessor are ignored here. As we will see later in this paper, this omission will have consequences.

As a side note, the algorithm described here is valid for iLO5 2.3x firmware. A minor variation is used for 2.1x firmware.

Part 3, as shown in listing 21, finally calls the function `coproc_crypto_key_schedule` which is really where the cryptographic material derivation takes place.

To coarsely summarize, `keymgr` hashes a state containing hardware stored (SOC) values padded with 0. The resulting digest (384 bits) is used as the initial seed for the cryptographic material derivation function.

```
v7 = coproc_crypto_key_schedule(&coproc_state);
coproc_crypto_sha384_digest(&coproc_state, 0x30u, &coproc_state);
SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;
memset(&coproc_state, 0, sizeof(coproc_state));
```

Listing 21. `coproc_crypto_initialize` - part 3

2.2 Cryptographic material derivation

As shown in listing 22, the key scheduling function is a bit complex, thus we are going to break it down into simpler sub-pieces. At high level, the key scheduling function is responsible for the generation of three elliptic curve keys (on the curve `secp384r1`). As shown in listing 23, the `gen_ec_key` function heavily relies upon OpenSSL primitives like `EC_KEY_generate_key`.

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

rnd_meth = hw_assisted_drbg();
RAND_set_rand_method(rnd_meth);

coproc_crypto_cmd(DRBG_BUFFER_POOL.derived_keyx_1,
                "DERIVED_KEYX", 1, 0, 0, out_buffer, 0x30);
drbg_init(out_buffer, 0x30u);
EC_KEY_POOL_.ec_key1 = gen_ec_key()

arg = dword_1F200A0 & 0xFFFFFF;
coproc_crypto_cmd(DRBG_BUFFER_POOL.derived_keyx_2,
                "DERIVED_KEYX", 3, &arg, 4, out_buffer, 0x30);
drbg_init(out_buffer, 0x30u);
EC_KEY_POOL_.ec_key2 = gen_ec_key()

extra_bytes_in = extra_entropy(tmp_buffer, 0x30);
coproc_crypto_cmd(DRBG_BUFFER_POOL.derived_keyx_3,
                "DERIVED_KEYX", 7, tmp_buffer,
                extra_bytes_in, out_buffer, 0x30);
drbg_init(out_buffer, 0x30u);
EC_KEY_POOL_.ec_key3 = gen_ec_key()

drbg_rand_bytes(tmp_buffer, 0x30u);
drbg_init(tmp_buffer, 0x30u);
```

Listing 22. Key scheduling in `keymgr`

```
EC_KEY* gen_ec_key()
{
    _ECKEY *ec_key;
    ec_key = EC_KEY_new_by_curve_name(NID_secp384r1);
    EC_KEY_generate_key(ec_key);
    EC_KEY_set_asn1_flag(ec_key, OPENSSL_EC_NAMED_CURVE);
    EC_KEY_set_flags(ec_key, EC_FLAG_COFACTOR_ECDH);
    return ec_key
}
```

Listing 23. Elliptic curve key generation

According to its documentation [9], "*the private key is a random integer (0 < priv_key < order, where order is the order of the EC_GROUP object)*". However, there is a twist in `keymgr`. The default set of functions that OpenSSL uses for random number generation are replaced (`RAND_set_rand_method`) by a custom number generator build around the features exposed by the cryptoprocessor.

The custom number generator follows the NIST SP 800-90A rev1 recommendations for random number generation using deterministic random

bit generators [7] (DRBG). More specifically it is a CTR DRBG, built around the AES256-CTR (counter mode) primitive of the cryptoprocessor.

Looking at the initialization of this DRBG (or seeding), as shown in listing 24, the `drbg_seed` function starts by hashing (`SHA384`) its buffer input argument. The result is then re-treated by a very important primitive we named `coproc_crypto_cmd`. To avoid unnecessary complexity, one can describe `coproc_crypto_cmd` as a keyed digest (`SHA384` cryptoprocessor primitive), associated with an expansion function to generate an arbitrary amount of output bytes. The "key" actually is the name of the "command", here "`DRBG_SEED_LB`".

```
int __fastcall drbg_seed(void *data_in, unsigned int data_size)
{
    int seeding_count;
    EVP_CIPHER *engine;
    int res;

    coproc_crypto_sha384_digest(data_in, data_size, DRBG_SEED_CTX);
    seeding_count = DRBG_INIT_COUNT++;
    coproc_crypto_cmd(DRBG_SEED_CTX, "DRBG_SEED_LB",
        seeding_count, 0, 0, DRBG_SEED_LB_KEY, 0x30);
    engine = EVP_aes_256_ctr();
    res = EVP_CipherInit(DRBG_CIPHER_CTX, engine,
        DRBG_SEED_LB_KEY, DRBG_SEED_LB_IV, 1);
    DRBG_NB_OUTPUT_BYTES = 0;
    return res;
}
```

Listing 24. Generate elliptic curve public/private key pair

The prototype of `coproc_crypto_cmd` is shown in listing 25. The input arguments are used to fill a buffer that is fed to the `SHA384` digest:

- `context_seed` is the seed context, a buffer of `0x30` bytes (384 bits).
- `misc_param` is an integer input value, usually the index of the key that is generated.
- `cmd_context` can be seen as the argument of the command, it may be used to pass extra entropy to the function.

```
void __fastcall coproc_crypto_cmd(
    _BYTE *context_seed,
    _BYTE *cmd_string,
    int misc_param,
    _BYTE *cmd_context, int cmd_context_size,
    _BYTE *output_buffer, int output_buffer_size)
}
```

Listing 25. `coproc_crypto_cmd` prototype

For the "DRBG_SEED_LB" command, the output buffer size is 0x30 bytes. The first 0x20 are used as the key of the AES256 algorithm of the DRBG, the next 0x10 bytes are used as its initialization vector.

Now, back to the key scheduling function, one can notice that the DRBG is re-seeded with a different context prior to each elliptic curve key generation. In the prologue of the key scheduling function, coproc_crypto_cmd is called with the command "KEY_SCHEDULE". This command generates a "seeding pool" of 0x240 bytes (twelve buffers of 0x30 bytes) for the DRBG. Incidentally the "DERIVED_KEYX" command is also called three times, some with minor modifications, such as passing extra entropy bytes.

A summary of the cryptographic material derivation is shown in figure 7. Three elliptic curve keys are generated. Logically, all randomness has been removed from the process. Indeed, using a DRBG with a deterministic seeding results in fully deterministic generated keys.

At the very end of the key scheduling function, just after the third elliptic curve key generation, as shown in orange in listing 22, the DBRG is re-used to generate 0x30 random bytes that are immediately used to re-seed the DRBG itself. This state will be used later silently.

The structures and initial seeding pool are designed to generate and handle twelve keys. However, only three keys are generated and only one actively used. This may indicate further plans from HPE.

2.3 Firmware decryption

So far, we have gone through the cryptographic material derivation, the output is a pool of three elliptic curve keys. As shown in listing 26, extract from the main function, only the first key is used. load_image_type function walks through the mapped firmware looking for an image header with a type set to 0x23, i.e. the new, secure, encrypted Integrity applicative image. If found, this image is copied to its target load address.

```
res = coproc_crypto_initialize();
if ( res )
{
    ec_key = EC_KEY_POOL_.ec_key1;
    bio = BIO_new_fp(FD_STDOUT, 0);
    PEM_write_bio_EC_PUBKEY(bio, EC_KEY_POOL_.ec_key1);
    BIO_free_all(bio);
}
load_image_type(load_addr, &image_size, 0x23, 1);
decrypt_image(ec_key, load_addr, &image_size);
```

Listing 26. keymgr main function

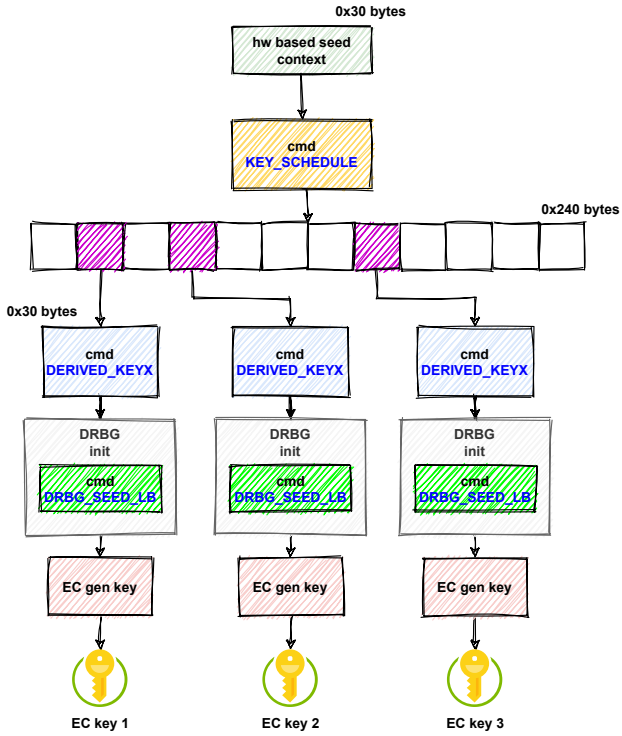

Fig. 7. Key scheduling summary

The loaded image is then processed by the `decrypt_image` function, an extract is given in listing 27. Besides, an overview of the header is shown in figure 8.

Again the code relies upon an OpenSSL primitive, in that case the function `ECDH_compute_key` [8]. According to its documentation it "*performs Elliptic Curve Diffie-Hellman key agreement. It combines the private key contained in ecdh with the other party's public key, [...] It stores the resulting symmetric key in the buffer out*". Interestingly, the function also makes use of the internal random number generated, this is the reason why the DRBG was re-seeded in the prologue of the key scheduling function.

The header of the encrypted image blob start with a `0x200` bytes buffer that contains an EC public key (point coordinate). This public key is combined with the private key previously generated to compute a shared secret (Elliptic Curve Diffie-Hellman key agreement). The shared secret, once re-treated through a `SHA384` digest, gives the encryption key and an initialization vector for the AES256-GCM cipher used to decrypt the image.

The idea is very similar to the encryption of the firmware envelope. Elliptic curve cryptography is used here instead of `RSA` to encrypt the AES key, The encrypted image structure is shown in figure 8.

```
ec_pubkey = d2i_EC_PUBKEY(0, &load_addr, 0x200);
memset(&context, 0, sizeof(context));

EC_KEY_set_flags(privkey, EC_FLAG_COFACTOR_ECDH);
pubkey = EC_KEY_get0_public_key(ec_pubkey);
buffer_size = ECDH_compute_key(derived_aes_key, 0x200, pubkey,
    privkey, 0);
EC_KEY_free(ec_pubkey);

coproc_crypto_sha384_digest(derived_aes_key, buffer_size, aes_key);
aes_engine = EVP_aes_256_gcm();
EVP_DecryptInit(&context, aes_engine, aes_key, load_addr + 0x200)
EVP_CIPHER_CTX_ctrl(&context, EVP_CTRL_AEAD_SET_TAG, 0x10, load_addr
    + *p_image_size - 0x10)
EVP_DecryptUpdate(&context, load_addr, &p_data_out_size, load_addr +
    0x20C, *p_image_size - 0x21C))
p_data_out_size;
*p_image_size = p_data_out_size;

EVP_DecryptFinal(&context, load_addr + v8, &p_data_out_size)
*p_image_size += p_data_out_size;
```

Listing 27. HPE iLO5 image decryption

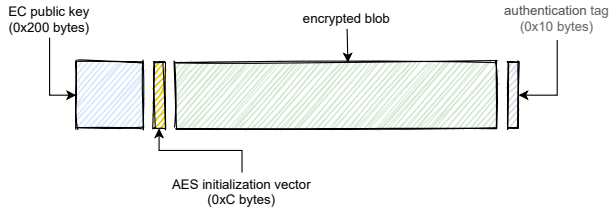

Fig. 8. HPE iLO5 encrypted image structure

3 Implementation

Our deep understanding of the key derivation algorithm enabled us to implement it in Python and C (for an easier `OpenSSL` use). However, the decrypted image is still a high-entropy blob of data, which means that we still do not generate the correct encryption key.

3.1 Interacting with the cryptoprocessor

Fortunately, one could ask the cryptoprocessor to perform operations without executing any arbitrary code by leveraging read and write primitives. We thus could use the `CVE-2018-7105` against the `SSH` service to experiment using the cryptoprocessor.

3.2 The `SHA384_DIGEST_MORE_DATA` bug

As we hinted previously, we discovered that the `SHA384_DIGEST_MORE_DATA` was not working as expected. Indeed, if the input buffer is not completely filled, its content is ignored (the internal state of the cryptoprocessor is not updated) but the counter is still updated by the amount of data written. In the "early digest" phase, this means that the `BootHash` is not taken into account while computing the hash. Instead, when the final hash is asked, the final input buffer is read as if it was a circular buffer, until `0x30 + 0x80` bytes are processed. A way to trigger the bug is pictured in figure 9.

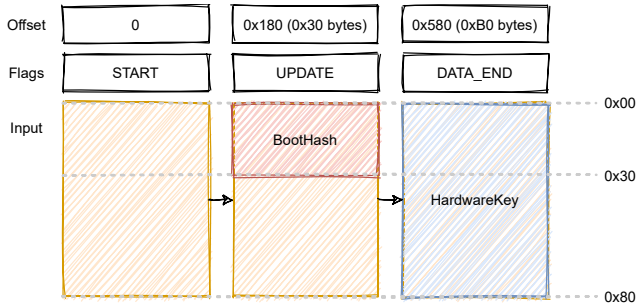

Fig. 9. Cryptoprocessor hash update bug

— **Expected behavior**:
SHA384(BootHash[0x0:0x30] || HardwareKey[0x0:0x80])
— **Real behavior**:
SHA384(HardwareKey[0x0:0x80] || HardwareKey[0x0:0x30])

3.3 The input buffer contiguity bug

We discovered a second bug, which might be linked to the first one: when the input buffer is not written contiguously, the counter is updated for each write by the amount of data written, with no consideration

about the offset in the input buffer, and the final hash is computed by contiguously reading in the input buffer as much bytes as described by the counter.

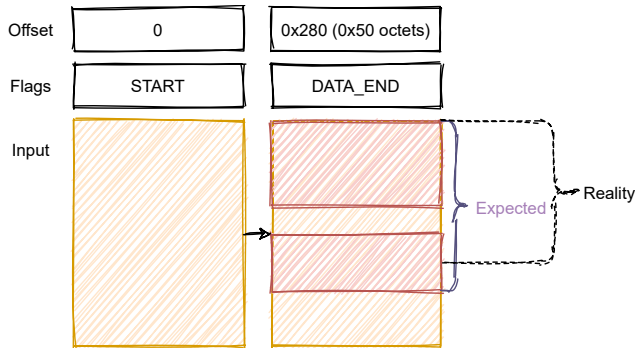

Fig. 10. Cryptoprocessor non-contiguous input buffer bug

This bug is present in the `coproc_crypto_cmd` function. When the `cmd_context` argument is `NULL`, its placeholder in the input buffer is not filled (to avoid writing null bytes where the content is already null) and the input buffer is thus written non-contiguously. A way to trigger the bug is pictured in figure 10.

— **Expected behavior**:
SHA384(INPUT[0x0:0x60])
— **Real behavior**:
SHA384(INPUT[0x0:0x50])

3.4 Adapting the implementation

With these two bugs in mind, we could fix our implementation and obtain the same outputs as when interacting with the cryptoprocessor. Still, the generated key does not allow to decrypt a correct firmware. With no more idea in mind, we had to find a way to gain debugging information.

4 Getting debugging information

While studying the `keymgr` task, we noticed that it outputted a subset of an intermediate state during the key derivation, as well as the generated public key, as seen in listing 28.

```
int __fastcall coproc_crypto_key_schedule(COPROC_STATUS *
    coproc_status) {
  // [...]
  coproc_crypto_cmd(coproc_status, "KEY_SCHEDULE", 0, 0, 0, &
      DRBG_BUFFER_POOL, 0x240);
  EVP_EncodeBlock(tmp_buffer, &DRBG_BUFFER_POOL, 0x21);
  printf("Key Schedule Validation: %s\n", tmp_buffer);
  // [...]
}

int main_task() {
  // [...]
  validation = coproc_crypto_initialize();
  if ( validation )
  {
    ec_key = EC_KEY_POOL_.ec_key1;
    bio = BIO_new_fp(FD_STDOUT, 0);
    PEM_write_bio_EC_PUBKEY(bio, EC_KEY_POOL_.ec_key1);
    BIO_free_all(bio);
  }
  // [...]
}
```

Listing 28. `keymgr` debug messages

Knowing the public EC key, we can check if our implementation failed because of the key derivation or the decryption phase. Moreover, the `Key Schedule Validation` output comes quite early in the derivation process: there are only two important parts before:

— "early digest": `SHA384` computation from `SOC` registers values;

— beginning of "cryptographic material seeding": a first call to `coproc_crypto_cmd` function with `KEY_SCHEDULE` as the `cmd_string` and a null `cmd_context`.

Both debug messages could be observed during `iLO 5` boot sequence by reading its `UART` output. During our previous study [6], we already found the `UART`, so it was just a matter of plugging a `USB-TTL` adapter and opening a `minicom` to read the debug messages, as seen in listing 29.

```
Loading 2.33.16
Key Schedule Validation: l03wN5OaDle9gyhfEJShR5jv0sKB0tfT25uk2U/
    vjxRA
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAE4StRIN6NFi6X000aNMLePDm0mYmXIpDf
03KrjkhWjZW8z3QNeyUXVNxHayZEKFL6Xk6vjkYYeJNdqg9yEzI0a2GK2emSgp4D
RNgUyUpix0jq5+lluKXWUyFQ6rBJ45Dr
-----END PUBLIC KEY-----
```

Listing 29. `keymgr` debug messages during boot sequence of `iLO 5 2.33`

Both debug values mismatched what we computed using our implementation. This indicates that we failed in a very early stage of the

`coproc_crypto_initialize` function, which is quite a good news since the code involved is fairly simple.

5 We need to try harder!

After trying to bruteforce some of the `SOC` registers values in case one of them changed since the earlier firmware, we came to the conclusion that we needed better debugging primitives to understand our failure(s).

5.1 The `JTAG` way

As our task-force was not united in a single geographical location, we had to buy a second server. `HPE` finally released a new `MicroServer` edition which includes `iLO 5`, the `MicroServer Gen10 Plus`. While reviewing the server specs on `HPE` website, a picture of the motherboard triggered our interest: there seems to be a populated debug port with the `iLO DEBUG` text printed (figure 11).

However, when we received our own `MicroServer`, the connector was not populated. We thus had to identify it as a `MICTOR 38` connector to buy and solder one on our motherboard, after having tried to directly solder copper wires on the `0.6mm` pads. To our surprise, we discovered that our previous `MicroServer` running `iLO 4` also had this debug connector, but there was not any mention of debugging capabilities printed on the motherboard.

Finally, the `JTAG` connection seems buggy: while `TMS`, `TCK` and `TDO` seem to work as expected (we are able to run an `IDCODE` scan), `TDI` has a weird behavior, and we could not manage to make it work at the time of writing this article. A logic analyzer trace shows that `TDO` output seems to be somehow replicated on `TDI`.

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

Listing 30. `JTAG IDCODE` scan

After a few days trying to make it work, we decided to explore another idea.

5.2 The `0day` way

Finding and exploiting a vulnerability Another way to debug our code was to be able to execute arbitrary code on an up-to-date `iLO 5`, to

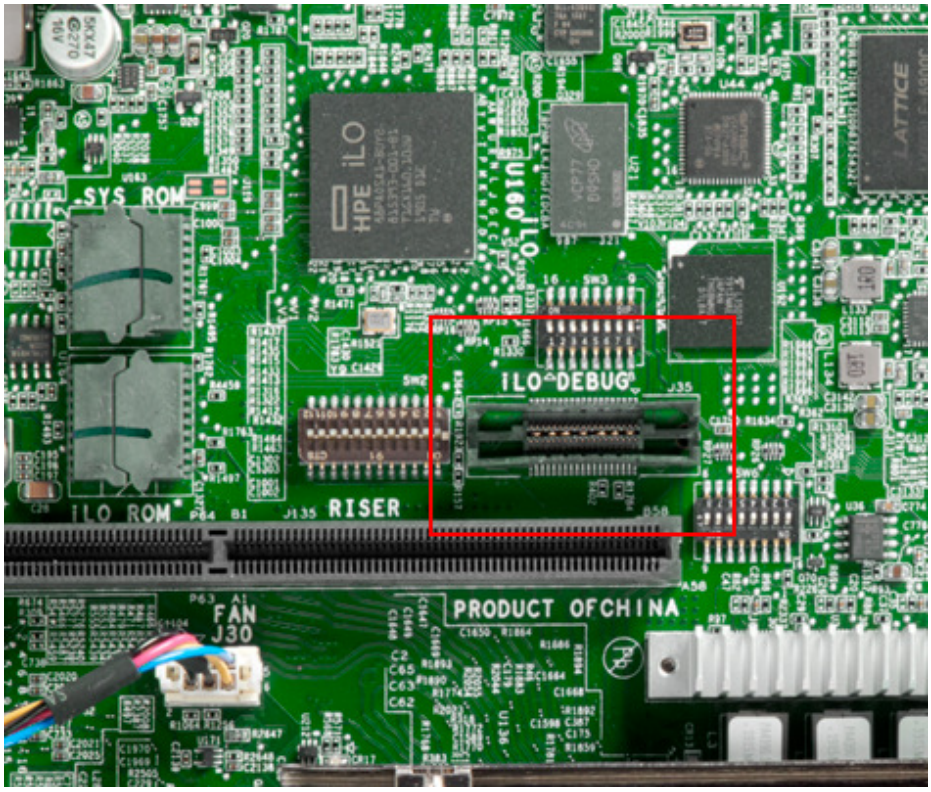

Fig. 11. `HPE MicroServer Gen10 Plus` debug connector

check if executing the exact same code as the `keymgr` task would produce different results than our implementation.

We thus started to review the attack surface reachable from the host operation system: the `CHIF` task and all the other tasks to which commands can be relayed.

Amongst all the tasks reachable from the host, the `blackbox` task is the only one which is also present in the recovery image, which is not encrypted after removing the initial envelope. We thus started to review the various command handlers, and found that command 5 seems to be a debug interface handling a bunch of subcommands and displaying results to the UART output. Some of the subcommands handlers were prone to buffer overflows, either in the stack or in the `.data` section.

As an example, here is how the `fview` subcommand is handled:

```
char v105[64]; // [sp+5C0h] [bp-C0h] BYREF
if ( argc > 1 && !strcmp(argv[1], "fview") )
```

```
{
  if ( argv[2] )
  {
    sprintf(v105, "/mnt/blackbox/data/%s", (const char *)argv[2]);
    return fview(v105);
  }
}
```

Listing 31. `fview` subcommand handling

As there is no `ASLR` nor `NX`, exploiting one of the stack buffer overflows to execute arbitrary code in the context of the `blackbox` task was not so complicated.

These vulnerabilities have been reported to `HPE` back in February 2021, a fix has been issued in March 2021 (`iLO 5` version `2.41` [3]) but at the time of writing this article, no security bulletin has been released. `HPE` plans to release security bulletin `HPESBHF04120` near May 18, 2021.

Using the vulnerability Now that we are able to execute arbitrary code in iLO, we could perform the following actions:

0. remap the `SOC` registers in our task using `memmap`;
1. execute the exact same code as `keymgr` for the "early digest" part;
2. dump the cryptoprocessor output buffer;
3. execute `coproc_crypto_cmd` function with the output as `context_seed`, `KEY_SCHEDULE` as the `cmd_string` and a null `cmd_context`;
4. dump the derivation output.

Figure 12 summarizes the process.

To our surprise, the cryptoprocessor output buffer and the derivation output contain **the exact same bytes** as what we computed offline, which are different than what is printed on the serial output.

Bonus: dumping cleartext credentials In order to appease our frustration, we decided to implement the dump of `iLO 5` users credentials using the vulnerability. Contrary to `iLO 4`, the `cfg_users.bin` file containing users credentials is now encrypted. However, the encryption key is stored in a second file, named `cfg_users_key.bin`. Dumping these two files is sufficient to retrieve cleartext credentials as seen in figure 13.

The `cfg_users.bin` contains the `IV` before the encrypted data and `cfg_users_key.bin` contains a 256-bits key. Both are used by `AES256-CBC` to decrypt the content.

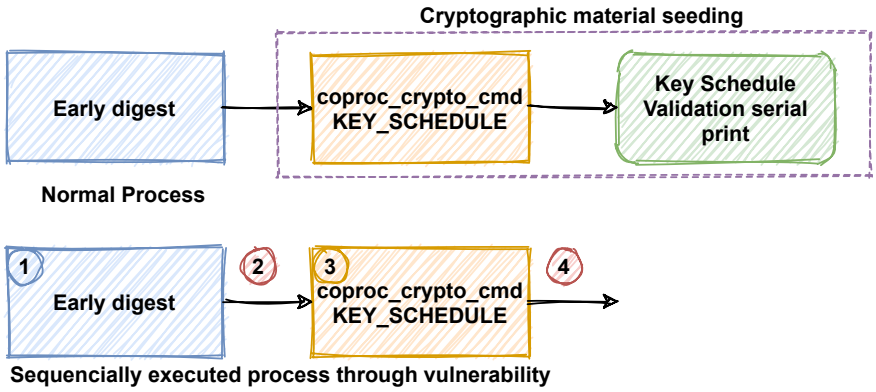

Fig. 12. Derivation process as executed through the vulnerability

```
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]
>>>
```

Fig. 13. Dumping cleartext users credentials

5.3 The `1day` way

The last remaining option is to be able to debug the `keymgr` task while it is effectively decrypting the userland image, as all our previous tries failed miserably. During our first study about `iLO 5` security [6], we successfully broke the secure boot by leveraging a vulnerability (`CVE-2018-7113`) in the kernel allowing us to run a modified userland image.

We concluded this study saying that attackers will always be able to build a "Frankenstein" firmware by using the old, vulnerable kernel with an up-to-date userland image. Now this is time to test this affirmation with a real use-case!

The software way Back in the days, we used a second vulnerability (`CVE-2018-7078`) in the `fum` task to be able to directly write the firmware on the `SPI` flash.

CVE-2018-7078 was fixed in version 1.30 (released in June 2018) for iLO 5 systems. When we tried to flash a vulnerable version (1.20) using the web administration feature, we observed inconsistent behaviour between our two target platforms:

— HPE ProLiant ML110 Gen10 (older) gracefully accepts the downgrade to a vulnerable version (1.20).
— MicroServer Gen10 Plus rejects the downgrade. The lowest version we were able to flash on this platform was 1.40.

This discrepancy has been discussed to HPE in the follow-up of our vulnerability report. Additional Active Health System (AHS) logs were shared with HPE to help them to identify the issue.

The hardware way To overcome the downgrade limitation and also to avoid having to reflash an old vulnerable firmware each time we wanted to test a new corrupted one, we directly went the hardware way (figure 14).

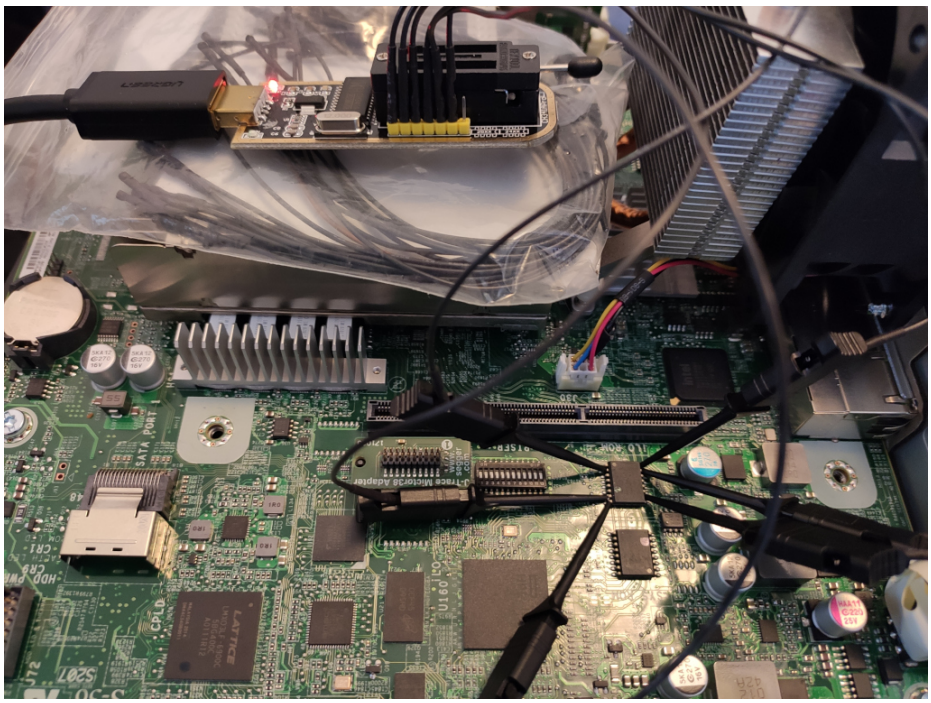

Fig. 14. Flashing custom iLO 5 firmware

While dumping for the first time the current installed firmware to be able to recover in case of failure, we noted that, despite being available in new firmware, first stage bootloaders are never flashed, and the current boot chain uses `Secure Micro Boot 1.01`.

We then tried to craft a custom firmware using a legitimate `iLO 5 2.33` decrypted firmware in which we injected:

— `Secure Micro Boot 1.01`;

— vulnerable `Integrity` kernel 1.30.

This first try was a success: `iLO` booted until the end of the `keymgr` task, because the old kernel did not implement the part used to load the final userland image.

Our second try was to modify a string in `keymgr` and inject it directly uncompressed (which would ease the next attempts): it still booted, allowing us to now inject various hooks to observe the different states of local variables and cryptoprocessor buffers during the decryption process.

We developed a minimal hooking engine using an unused function as a code cave, and could dump arbitrary base64 encoded data on the serial output. Our first move was to dump the data passed to the `coproc_crypto_key_schedule` function. This data stems from a temporary buffer containing the `0x30` first bytes of the cryptoprocessor output buffer after the "early digest" phase. The retrieved value still mismatched our implementation.

A first positive point though, if we injected the retrieved values in our implementation, we could compute the correct key to decrypt the userland image, which indicates that our problem was located before the first `coproc_crypto_key_schedule` function call.

As we wanted to ensure we got the good input values, we also dumped the whole cryptoprocessor shared memory (including input buffer, output buffer, counter and flags) and noticed a weird inconsistency: the output buffer contained the same bytes as our implementation. A closer look at the stack buffer content showed that the difference only affected the first 4 bytes.

```
Stack buffer
00000000: BC E9 73 3A DD 13 69 EB  F4 CA BC 41 B4 8D DB DF
00000010: 76 AF D7 35 82 66 4C 2D  12 99 C2 23 E5 85 09 AE
00000020: DC 67 2A A6 D0 A9 90 4F  CF AC C7 27 6E A8 FC 9A

Cryptoprocessor output buffer
00000000: 8F D6 EA 44 DD 13 69 EB  F4 CA BC 41 B4 8D DB DF
00000010: 76 AF D7 35 82 66 4C 2D  12 99 C2 23 E5 85 09 AE
00000020: DC 67 2A A6 D0 A9 90 4F  CF AC C7 27 6E A8 FC 9A
```

Listing 32. Difference between stack buffer and cryptoprocessor output buffer

While looking at the code in IDA Pro, nothing could explain such a behavior:

```
// fill input buffer
// [...]
SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_DATA_END;// sha2.digest()
for ( 1 = 0; 1 < 0xC; ++1 )
  *&stack_buffer[4 * 1] = SHA384_DIGEST_OUTPUT[1];
*&SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;
```

Listing 33. Copying cryptoprocessor output buffer

When dumping the whole cryptoprocessor shared memory, another weird behavior can be observed in the input buffer:

```
00000080: 0000 0000 0000 0600 0000 0000 0000 0000
00000090: 0000 0000 C3FF 7FBF 000D 1C85 1064 F232
000000A0: 2106 0008 0000 0000 0000 0000 0000 0000
000000B0: 8000 0000 0000 0000 0000 0000 0000 0000
000000C0: 0000 0000 0000 0000 0000 0000 0000 0000
000000D0: 0000 0000 0000 0000 0000 0000 0000 0000
000000E0: 0000 0000 0000 0000 0000 0000 0000 0000
000000F0: 0000 0000 0000 0000 0000 0000 0000 0580
```

Listing 34. Cryptoprocessor input buffer after computing digest

Values in blue are typical SHA2 padding values: a `0x80` byte followed by zeros, and finally the input size in bits in big endian (`0x580`). This observation indicates that the cryptoprocessor directly uses the shared memory as its working buffer. Therefore, the output buffer might contain intermediary hash value before the final digest is asked by setting flag `SHA384_DIGEST_DATA_END`;

We thus dumped the cryptoprocessor state just before the `digest` flag is set:

```
00000040: BCE9 733A 2CB0 47EC B74C D8D4 0850 7DBB
00000050: 7937 CD5A 4599 CA65 5920 3622 16D8 A65A
00000060: 9AA5 62B7 4B50 E3BF DF5A 26E8 8D15 CECE
```

Listing 35. Cryptoprocessor output buffer after computing digest

The first 4 bytes are the same as what we found in the stack buffer! There is a race condition between the application processor and its cryptographic coprocessor: the copy loop starts copying bytes from an intermediary state before the cryptoprocessor effectively update them. A correct implementation should have waited for the cryptoprocessor to updated its flag indicating that the output is available. The race condition has been reliably observed in 100% of our experiments with the cryptoprocessor. As a final argument, to generate the correct decryption key for the

firmware, one has to take this race condition into account. We can only make the assumption that HPE firmware encryption pipeline mirrors that odd inconsistency.

One could observe this same intermediate state by dumping the SHA-2 internal state before the `final` operation.

6 Conclusion

Now that we understood how the firmware encryption works, we could develop new tools in Python and C to handle decryption. The resulting cleartext firmware can now be correctly parsed by the tools available at [5]. A quick look at the `2.x` firmware branch indicates that 4 new tasks are present, but they do not seem to add so much attack surface.

Even though it relies upon well known and tested algorithm (AES, SHA384), the key derivation function designed by HPE and implemented in `keymgr` is surprisingly complex. No vulnerability was found in that function. However, by design, reading some specific memory area of the SOC is enough the leak the seeds and thus re-generate all the needed crypto-material.

We're still wondering what the firmware encryption pipeline looks like on `HPE` side given all the inconsistencies we encountered while trying to reimplement it. We can only make the assumption that HPE firmware encryption pipeline mirrors the various inconsistencies we have observed during our research.

About the firmware encryption, we do not see a huge gain in terms of security:

— on a vulnerability research PoV, a motivated attacker could still reverse the encryption scheme and have a look at new firmware;
— in case of a supply chain attack, the security was already supposed to be assured by the secure boot, the encryption does not raise the bar.

About the secure boot, we concluded our previous study [6] by saying that the secure boot was broken forever because of the lack of a hardware revocation; this statement was proved right by this new chapter, as we successfully built a hybrid firmware to debug the `keymgr` task.

To conclude this article, we wanted to remind readers that `iLO 5` is a critical component in a server security. However, it is still vulnerable to supply chain attacks due to the broken secure boot. Besides, as highlighted once again by the new vulnerability, it lacks all the exploit mitigations

that have been implemented in modern operating systems for about 20 years.

References

1. ARM. Security IP - CryptoCell-700 family. https://developer.arm.com/ip-products/security-ip/cryptocell-700-family.
2. Hewlett Packard Enterprise. HPESBHF03866 rev.3 - HPE Integrated Lights-Out 3,4,5 iLO Moonshot and Moonshot iLO Chassis Manager, using SSH, Remote Execution of Arbitrary Code, Local Disclosure of Sensitive Information. https://support.hpe.com/hpsc/doc/public/display?docId=hpesbhf03866en_us, 2018.
3. Hewlett Packard Enterprise. Hpe integrated lights-out 5 firmware 2.41 (26 mar 2021). https://support.hpe.com/hpesc/public/swd/detail?swItemId=MTX_20c4517c90cd43f1b5982d21c9, 2021.
4. Alexandre Gazet Fabien Perigaud and Joffrey Czarny. Backdooring your server through its bmc: the hpe ilo4 case. https://airbus-seclab.github.io/ilo/SSTIC2018-Slides-EN-Backdooring_your_server_through_its_BMC_the_HPE_iLO4_case-perigaud-gazet-czarny.pdf, 2018.
5. Alexandre Gazet Fabien Perigaud and Joffrey Czarny. Subverting your server through its BMC: the HPE iLO4 case - iLO4 toolbox. https://github.com/airbus-seclab/ilo4_toolbox, 2018.
6. Alexandre Gazet Fabien Perigaud and Joffrey Czarny. Turning your bmc into a revolving door. https://airbus-seclab.github.io/ilo/ZERONIGHTS2018-Slides-EN-Turning_your_BMC_into_a_revolving_door-perigaud-gazet-czarny.pdf, 2018.
7. NIST. SP 800-90A Rev. 1 - Recommendation for Random Number Generation Using Deterministic Random Bit Generators. https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final.
8. OpenSSL. Ecdh_compute_key, ecdh_size — elliptic curve diffie-hellman key exchange. https://man.openbsd.org/ECDH_compute_key.3.
9. OpenSSL. Ec_key_new. https://www.openssl.org/docs/man1.1.0/man3/EC_KEY_generate_key.html.
10. Fabien Perigaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its bmc: the hpe ilo4 case. RECon conference, https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2018-Subverting-your-server-through-its-BMC-the-HPE-iLO4-case.pdf, 2018.
11. Nicolas Iooss (@fishilico). Commit 430bfb9: "Add SSH exploit for CVE-2018-7105". https://github.com/airbus-seclab/ilo4_toolbox/commit/430bfb9592a543e1fbfe9f71ed930479e6809d86.