Subverting your server through its BMC: the HPE iLO4 case

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

[1] Synacktiv
[2] Airbus

Abstract. iLO is the server management solution embedded in almost every HP server since more than 10 years. It provides the features required by a system administrator to remotely manage a server without having to physically reach it. iLO4 (known to be used on the family of servers HP ProLiant Gen8 and ProLiant Gen9) runs on a dedicated ARM microprocessor embedded in the server, totally independent from the main processor. We performed an initial deep dive security study of HP iLO4 [6] and covered the following topics:

– Firmware unpacking and memory layout
– Embedded OS internals
– Vulnerability discovery and exploitation
– Full compromise of the host server operating system through DMA

One of the main outcome of our study was the discovery of a critical vulnerability in the web server component allowing an authentication bypass but also a remote code execution [6, 9]. Still, one question remains open: are the iLO systems resilient against a long term compromise at firmware level? For this reason, we focus on the update mechanism and how a motivated attacker can achieve long term persistence on the system.

1 Introduction

1.1 IPMI/BMC introduction

The Intelligent Platform Management Interface (IPMI) is a suite of computer interface functions for an autonomous computer subsystem that provides management and monitoring capabilities independently of the host system's CPU, firmware (BIOS or UEFI) and operating system.

IPMI defines a set of interfaces used by system administrators for out-of-band management. For example, IPMI provides a way to manage a computer that may be powered off or otherwise unresponsive by using a network connection to the hardware rather than to an operating system or login shell.

An IPMI sub-system consists of a main controller, called the Baseboard Management Controller (BMC) and other management controllers distributed among different system modules. BMCs have been embedded in most of HP servers for more than 10 years.

1.2 HP Integrated Lights-Out

Integrated Lights-Out, or iLO, is a proprietary embedded server management technology by Hewlett-Packard which provides out-of-band management facilities. The physical connection is an Ethernet port that can be found on most Proliant servers and microservers of the 300 and above series.

iLO has similar functionality to the Lights Out Management (LOM) technology offered by other vendors such as Sun/Oracle's LOM port, Dell DRAC, IBM Remote Supervisor Adapter and Cisco CIMC.

iLO provides remote administration features such as:

- Power Management
- Remote system console
- Remote CD/DVD image mounting
- Several monitoring indicators

On the hardware side, the iLO chip is directly integrated on the server's motherboard (see figure 1). It is composed of:

- Dedicated ARM processor: GLP/Sabine architecture
- Dedicated RAM chip
- Firmware stored on a NAND flash chip
- Dedicated network interface

On the software side, iLO provides various services for administrators to interact with, such as a web server and a ssh server.

There is a full operating system running in your server as soon as it has a connected power cord! As said before, iLO runs even if the server is turned off.

iLO has a privileged (read/write) access to the server communication buses. For example, it is directly connected to the PCI-Express bus (see figure 2).

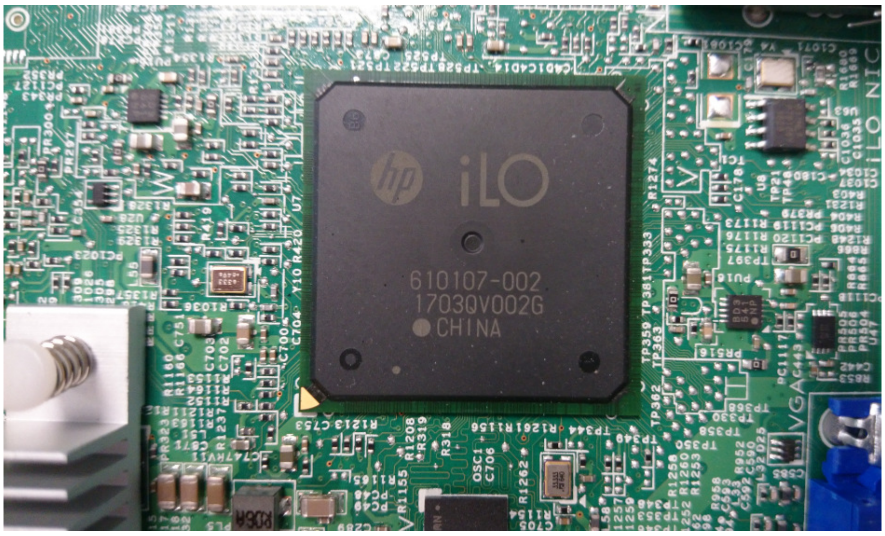


Fig. 1. `iLO` chip on server's motherboard

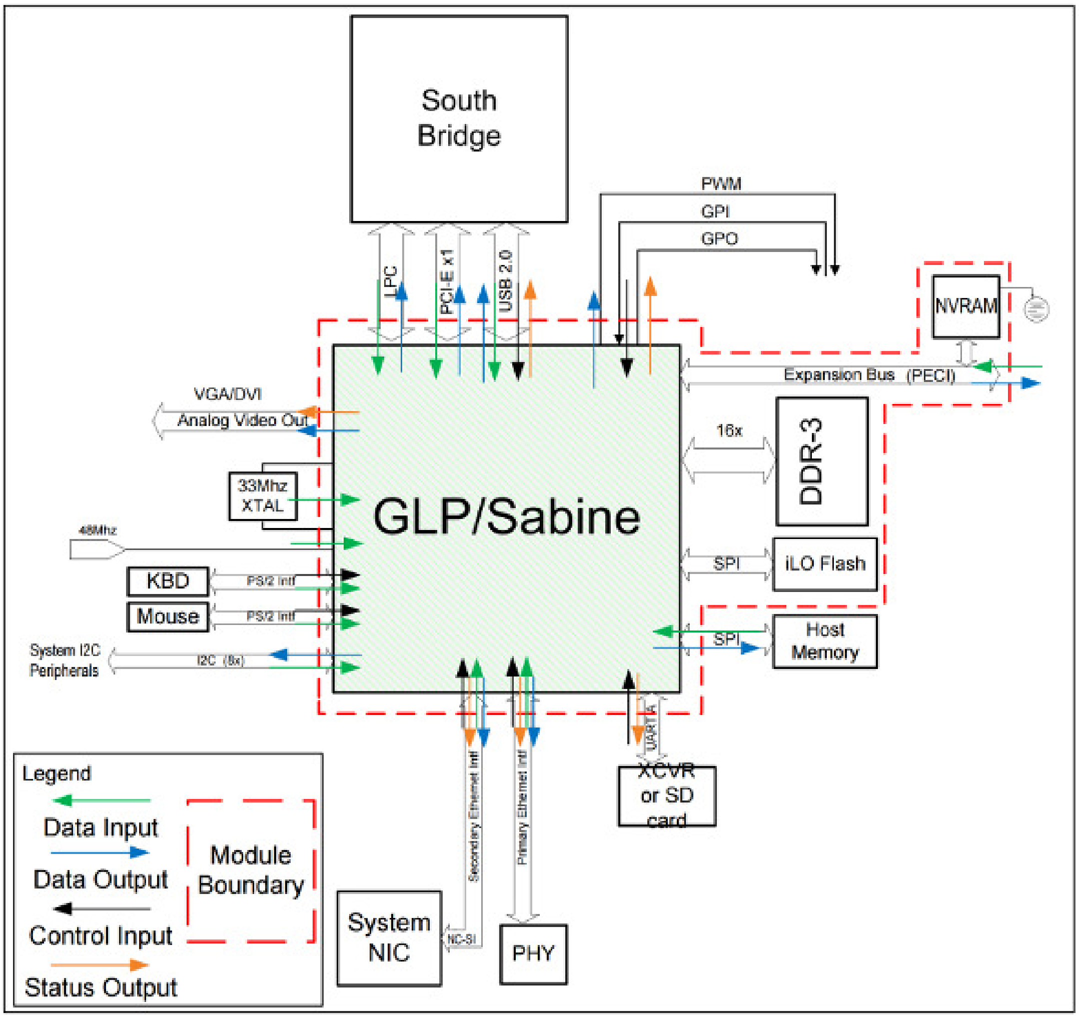


Fig. 2. `iLO` privileged hardware access

2 Context

2.1 Previous work on iLO

As a pentester/red-teamer you definitely have met iLO on your target network. Unfortunately, this interface is too rarely protected and is fully exposed. Some previous works have been done on this topic; researchers have published two main vulnerabilities:

– IPMI Authentication Bypass via Cipher 0
– IPMI 2.0 RAKP Authentication Remote Password Hash Retrieval[3]

The first vulnerability allows remote attackers to bypass authentication and execute arbitrary IPMI commands by using cipher suite 0. Indeed, this cipher suite does not require the user to provide a password. This issue has been fixed by HP in July 2013, see HP Customer Notice HPSN-2008-002[4].

The second vulnerability is an issue in the IPMI 2.0 specification on RMCP+ Authenticated Key-Exchange Protocol (RAKP) authentication. It allows remote attackers to obtain password hashes from a RAKP message 2 response. The only prerequisite to this attack is the knowledge of valid usernames. Then, the password cracking attack can be conducted offline. This flaw is present "by design" in the protocol and thus can not be easily fixed. HP has now disabled IPMI in the default configuration of iLO5.

We strongly recommend the reader to refer to these previous papers/publications:

– "*IPMI: freight train to hell*", by Dan Farmer [3]
– "*A Penetration Tester's Guide to IPMI and BMCs*" [5]

To our knowledge, at the time we performed our study (*i.e.* mostly by the end of 2016, beginning of 2017), the IPMI 2.0 password hash retrieval was the only known public vulnerability impacting up-to-date iLO4 systems.

2.2 Presence of iLO4 on Internet

iLO interface is usually exposed on the internal network, but also sometimes on the Internet. Indeed, in some cases, hosting providers can offer an access to reach the BMC systems in order to troubleshoot an issue if the connection with the host is lost.

[3] http://fish2.com/ipmi/remote-pw-cracking.html
[4] https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c03844348

A survey has been done in September 2017 and January 2018 on the exposure of `iLO4`. The simple scanner we developed has been released as part of our `ilo4_toolbox`[5]. Versions `2.53`, `2.54` and `2.55`, marked with an arrow, are versions where the vulnerability is fixed.

By performing a network scan on all public `IPv4` addresses, around 3,604 `iLO` interfaces version 4 are discovered exposed in September 2017:

```
   3 Server:HP-iLO-4/1.30 UPnP/1.0 HP-iLO/1.0
   1 Server:HP-iLO-4/1.51 UPnP/1.0 HP-iLO/1.0
 112 Server:HP-iLO-4/2.00 UPnP/1.0 HP-iLO/1.0
 140 Server:HP-iLO-4/2.02 UPnP/1.0 HP-iLO/1.0
 172 Server:HP-iLO-4/2.03 UPnP/1.0 HP-iLO/1.0
 230 Server:HP-iLO-4/2.10 UPnP/1.0 HP-iLO/2.0
 189 Server:HP-iLO-4/2.20 UPnP/1.0 HP-iLO/2.0
  29 Server:HP-iLO-4/2.22 UPnP/1.0 HP-iLO/2.0
 461 Server:HP-iLO-4/2.30 UPnP/1.0 HP-iLO/2.0
   4 Server:HP-iLO-4/2.31 UPnP/1.0 HP-iLO/2.0
 552 Server:HP-iLO-4/2.40 UPnP/1.0 HP-iLO/2.0
  14 Server:HP-iLO-4/2.42 UPnP/1.0 HP-iLO/2.0
 108 Server:HP-iLO-4/2.44 UPnP/1.0 HP-iLO/2.0
1050 Server:HP-iLO-4/2.50 UPnP/1.0 HP-iLO/2.0
 219 Server:HP-iLO-4/2.53 UPnP/1.0 HP-iLO/2.0   <--
 320 Server:HP-iLO-4/2.54 UPnP/1.0 HP-iLO/2.0   <--
```

We performed the same scan in January 2018, around 3,788 `iLO` interfaces version 4 were discovered exposed:

```
  86 Server:HP-iLO-4/2.00 UPnP/1.0 HP-iLO/1.0
 117 Server:HP-iLO-4/2.02 UPnP/1.0 HP-iLO/1.0
 144 Server:HP-iLO-4/2.03 UPnP/1.0 HP-iLO/1.0
 173 Server:HP-iLO-4/2.10 UPnP/1.0 HP-iLO/2.0
 169 Server:HP-iLO-4/2.20 UPnP/1.0 HP-iLO/2.0
  26 Server:HP-iLO-4/2.22 UPnP/1.0 HP-iLO/2.0
 297 Server:HP-iLO-4/2.30 UPnP/1.0 HP-iLO/2.0
   2 Server:HP-iLO-4/2.31 UPnP/1.0 HP-iLO/2.0
 422 Server:HP-iLO-4/2.40 UPnP/1.0 HP-iLO/2.0
   9 Server:HP-iLO-4/2.42 UPnP/1.0 HP-iLO/2.0
  83 Server:HP-iLO-4/2.44 UPnP/1.0 HP-iLO/2.0
1020 Server:HP-iLO-4/2.50 UPnP/1.0 HP-iLO/2.0
 193 Server:HP-iLO-4/2.53 UPnP/1.0 HP-iLO/2.0   <--
 571 Server:HP-iLO-4/2.54 UPnP/1.0 HP-iLO/2.0   <--
 474 Server:HP-iLO-4/2.55 UPnP/1.0 HP-iLO/2.0   <--
```

2.3 Our approach for the initial study

It is clear that `iLO` is a critical technology. By design, it provides a full remote management interface for `HP` servers. Moreover, known weaknesses exist in the authentication protocol and few people actively monitor `iLO` systems; we needed nothing more to dive into it. Our goals were to:

[5] `https://github.com/airbus-seclab/ilo4_toolbox`

- Evaluate the trust we can put in the solution/product
- Better understand the technology **and its internals**
- Better understand the exposed surface/risk

One of the main outcome of our study was the discovery of a critical vulnerability in the web server component (CVE-2017-12542, CVSSv3 base score 9.8), allowing an authentication bypass but also a remote code execution. This vulnerability has been fixed in iLO 4 versions 2.53 and 2.54.

Exploitation of this vulnerability allows an attacker to fully compromise a server and break the segmentation between the iLO and the host. Indeed, it has been demonstrated that it is possible to obtain the highest privileges on the host from the iLO system. All the details have already been published during ReCon Brussels in February 2018 [6].

The responsible disclosure timeline is provided as an indication to readers with an eye for details...

- **Feb 2017** - Vulnerability discovered
- **Feb 27 2017** - Vulnerability reported to HP PSIRT by Airbus CERT
- **Feb 28 2017** - HP acknowledges receiving the report
- **May 5 2017** - HP releases iLO 4 2.53, silently fixing the vulnerability
- **July 20 2017** - Airbus CERT contacts MITRE to request a CVE ID
- **July 28 2017** - HP PSIRT tells Airbus CERT that they are planning to release a security bulletin
- **August 24 2017** - HP releases security bulletin HPESBHF03769[6]
- **Feb 4 2018** - All details are presented during ReCon Brussels

2.4 A necessary supplement for this study

In order to answer to the first objective, namely "Evaluate the trust we can put in the solution/product", we also had to validate the security measure implemented on the firmware update process and more specifically the mechanisms set to validate the integrity of updates and their origin. Fortunately, the previous study allowed us to identify several modules and data structures involved in the process of firmware integrity verification (a brief summary is provided in section 3.1).

Besides, there are very few mechanisms or tools to validate the presence of a rootkit inside BMC systems. In case of a compromised system, people usually change hard drives, but few people check for implants installed on the hardware.

[6] https://support.hpe.com/hpsc/doc/public/display?docId=hpesbhf03769en_us

Thus, this study is focused on the update process and how a new/backdoored firmware can be installed and allow an attacker to be persistent in an environment which has been compromised.

3 iLO4 firmware integrity

3.1 Update process overview

In order to update an `iLO 4` firmware, the first step is usually to obtain an update package from the vendor website. For a `Windows` based host, it comes as an executable binary: `CP030133.exe` for `iLO 4` version `v2.44` for example. It should be noted that `pingtool.org`[7] also provides a great repository of archived firmware versions.

The following elements are based on the analysis of the update package `CP030133.exe` (`iLO 4 v2.44`). This self-extracting/script based archive is quickly dissected and contains the following content:

```
total 17M
-rwxr-xr-x 1 user None 198K Jul 21  2016 CP030133.xml*
-rwxr-xr-x 1 user None 490K Apr  1  2016 flash_ilo4*
-rwxr-xr-x 1 user None  17M Jul 21  2016 ilo4_244.bin*
-rwxr-xr-x 1 user None 9.9K Jul 21  2016 Readme.txt*
```

The relevant files are:

- `flash_ilo4`: flashing tool, `x86` code
- `ilo4_244.bin`: the actual firmware, concatenation of:
  - the *HP Signed File* header

    ```
    --=</Begin HP Signed File Fingerprint\>=--
    Fingerprint Length: 000527
    Key: label_HPBBatch
    Hash: sha256
    Signature: WtLLCUv/ergBGLM6fULxgUUvffHNPNblf5KQFUY0BKxYznzepQggzhF/UsuU2zlrd0D
    +KH0YN0OdkycgVDKjilkD1nCgPrfL0yjZLI22A0NZ0uEle3uW+Gvkj3s178Zt1RJizAYLXU/vAG47G
    0R1MjKmB8ca5tzJKxuRi1AxtRcfU7DaVtHPTPZ7ro5QL+JH7/EeBIZbi79CsHTg0kVdiPNaVlQ1eYb
    uKjLwHptuTm0AmpvPnZ6oQi8FDmtHSeEIY4nCBl7GwBTYMYVUMwDcI8HQypuwna0dAeUy4z2/xYcIu
    kbwlZNREDt4QPHZzCP52clJIRhtwsjdD2SUwj3jGA== Fingerprint Length: 000527
    --=</End HP Signed File Fingerprint\>=--
    ```

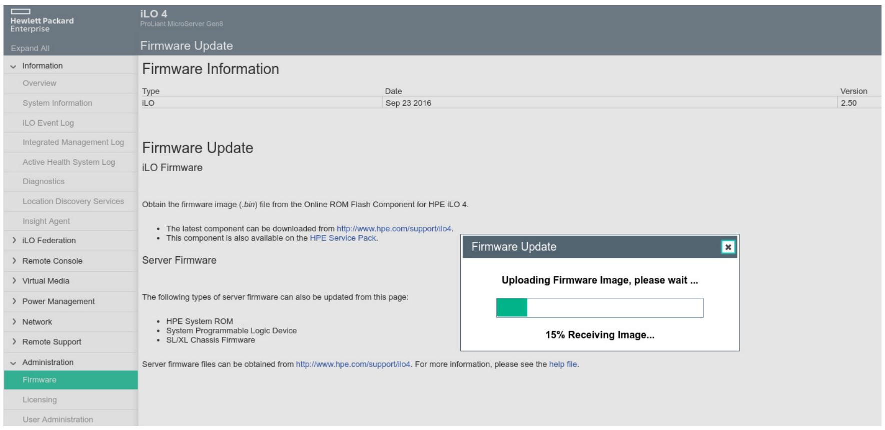
  - three certificates from `HP`
  - the `HPIMAGE` blob

From there, an `iLO` administrator can update the firmware by either:

- Running the binary `flash_ilo4` on the host (`x86`-based) system. Its purpose is to "*flash*" the binary image `ilo4_244.bin` by sending it to the `iLO` though a shared-memory communication channel.

[7] `http://pingtool.org/latest-hp-ilo-firmwares/`

– Using the web server to directly upload the `ilo4_244.bin` file, as seen in figure 3.



Firmware Information

Type	Date	Version
iLO	Sep 23 2016	2.50

Firmware Update

iLO Firmware

Obtain the firmware image (*.bin*) file from the Online ROM Flash Component for HPE iLO 4.

- The latest component can be downloaded from http://www.hpe.com/support/ilo4.
- This component is also available on the HPE Service Pack.

Server Firmware

The following types of server firmware can also be updated from this page:

- HPE System ROM
- System Programmable Logic Device
- SL/XL Chassis Firmware

Server firmware files can be obtained from http://www.hpe.com/support/ilo4. For more information, please see the help file.

Firmware Update

Uploading Firmware Image, please wait ...

15% Receiving Image...

Fig. 3. Firmware update through the web server

In both cases, the firmware file will finally be handled by a userland `task` of the `iLO` system called `fum`. `iLO4` systems rely upon the `Integrity` operating system developed by Green Hills Software[8]. In this context, a `task` is a userland process, with its own set of threads and virtual memory mappings. For example, the web server and the `SSH` server each run in a separated `task`.

When the `fum` task receives the firmware file, it looks for the `HP Signed File` header containing the signature and hash algorithm; then it checks its validity using its own embedded `RSA` public key:

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAteyCedpzasCIZeLkygK/GsUB29BY6wR0zcw/N5M/PitwnkNLn/yb
i7FKQIfoH7wRLzPSLWUORRKRy5OvfRwiw+6ezxlgjp/IvM75mI56KoanlyRw04FZ
mjfHKndMTCMaozBLUpIgfCr33NsAI4EcIG/edp7fgzUMr/T4xE0lyHxzCi0q70HP
BjuQ+CKrwbCPfvxOEA3vw+/fQq0f5RhZ+ihAKZyzcAzLVW0SI4gEvzm0L3uUolmM
lX/QAAWPA5fJfkGQAARS+I8pyb/sz9eaXb+JB/ukuGffwzPuqyKGcGilNIKsFKF4
8+QBYCutnDOFy7uekLLb9GUuKjWiDe8DOwIDAQAB
-----END RSA PUBLIC KEY-----
```

If the signature is correct, the userland and kernel parts of the firmware are written on the flash. Depending on a physical switch on the server, the bootloader will also be written. This physical switch is only checked in software and does not prevent from writing to a specific zone of the flash. After the flashing operation has completed, the `iLO` reboots.

[8] `https://www.ghs.com/products/rtos/integrity.html`

During the boot chain, each component of the firmware is checked by its parent:

- the bootloader checks the kernel signature
- the kernel checks the userland signature

However, the signature of the bootloader is not checked at boot time. For now let's consider the signature is correct, we can then proceed to the `HPIMAGE` blob.

3.2 `HPIMAGE` blob

The binary `HPIMAGE` binary blob is the actual data that is written on the `NAND` flash chip. Let's start dissecting the `HPIMAGE`, starting with the blob header:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  48 50 49 4D 41 47 45 00 01 01 00 00 9D 7B 31 2F  HPIMAGE......{1/
00000010  E3 C9 76 4D BF F6 B9 D0 D0 85 A9 52 01 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 E0 07 07 13  ...............
00000040  32 2E 34 34 00 00 00 00 00 00 00 00 00 00 00 00  2.44............
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060  69 4C 4F 20 34 00 00 00 00 00 00 00 00 00 00 00  iLO 4...........
[...]
00000490  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000004A0  01 00 00 00 29 32 EC AE CC 69 D8 43 BD 0E 61 DC  ................
000004B0  34 06 F7 1B 00 00 00 00                          4.......
```

Listing 1. Dump of an `HPIMAGE` header

The key elements are:

- Magic: `HPIMAGE`
- Size: `0x4B8`
- Version (`2.44`) and two `GUID`s
- Size of mapped firmware without header: `0x1000000` bytes

The two `GUID`s (in red on listing 1) are interesting with regards to the update process. Their semantics can be understood by reversing the `fum` task binary (see the `Python` definition of the `FlashEntry` structure presented on listing 2). Indeed they respectively indicate the update type and target device type (see listing 3 and 4). For this file, they correspond to an `iLO 4 Firmware` type, dedicated to an `iLO 4` hardware, as expected.

The `HPIMAGE` format can be used to package many different update types such as: `iLO 4 Firmware`, `System ROM`, `CPLD-JTAG`, `Language Pack`, *etc.* One can note that the minimum version field is always set to zero, thus it is possible to downgrade firmware.

Finally, looking at the end of the file, one can also found a footer (see figure 5).

```
class FlashEntry(LittleEndianStructure):
    _fields_ = [
        ("name_ptr", c_uint32),
        ("unknown", c_uint32),
        ("guid", c_byte*0x10),
        ("type", c_uint32),
        ("min_ver", c_byte),
        ("field_1D", c_byte),
        ("field_1E", c_byte),
        ("field_1F", c_byte),
        ("field_20", c_uint32),
        ("field_24", c_uint32),
        ("field_28", c_uint32),
        ("field_2C", c_uint32),
        ("field_30", c_uint32),
        ("field_34", c_uint32),
        ("field_38", c_uint32)
    ]
```

Listing 2. FlashEntry structure

```
> parsing flash types:
    iLO 4 Firmware - guid 9d7b312fe3c9764dbff6b9d0d085a952 - type 0x01 - min ver 0x0
        System ROM - guid 2e8d14aa096e3e45bc6f63baa5f5ccc4 - type 0x05 - min ver 0x0
        Custom ROM - guid 916b239911c283429ca97423f25687f3 - type 0x06 - min ver 0x0
         CPLD-JTAG - guid 9a43adb1d19dc141a4962da9313f1f07 - type 0x07 - min ver 0x0
        Carbondale - guid 3bad180a84cb0c479050cafb33371a14 - type 0x08 - min ver 0x0
               PIC - guid 90aa533689703a45899c792827a50d67 - type 0x0a - min ver 0x0
        EEPROM I2C - guid dffc32e2cbbc5347a99bf6b11c6eb074 - type 0x0b - min ver 0x0
             Files - guid 18077fda4c441c49b9bfb5a9ccc5e6e8 - type 0x0c - min ver 0x0
     Language Pack - guid 0c4c1027c53a91498afbd1f3cd166fb4 - type 0x0d - min ver 0x0
    iLO (Moonshot) - guid a8d1685fab9795408c68bc3e1125268b - type 0x01 - min ver 0x0
   CPLD (Moonshot) - guid 8384790bfcabcc4c914e26c4fb948cff - type 0x07 - min ver 0x0
```

Listing 3. List of supported HPIMAGE update types

```
> parsing device types:
         iLO 4 - flags 0x008 - guid 2932ecaecc69d843bd0e61dc3406f71b - min ver 0x0
     Server ID - flags 0x001 - guid 0000000000000000000000000000ffff - min ver 0x0
          BIOS - flags 0x002 - guid 00000000000000000000000001ffffff - min ver 0x0
   BootBlock 0 - flags 0x080 - guid 00000000000000000000000001ffffff - min ver 0x0
   BootBlock 1 - flags 0x100 - guid 00000000000000000000000001ffffff - min ver 0x0
    Carbondale - flags 0x004 - guid 0000000000000000000000000000cdff - min ver 0x0
     Power PIC - flags 0x010 - guid 00000000000000000000000000504dff - min ver 0x0
   NMVe BP PIC - flags 0x200 - guid 000000000000000000000000ffffffff - min ver 0x0
      OEM Data - flags 0x040 - guid 4cb0f50e84b9984295f04b3fffffffff - min ver 0x0
           PS1 - flags 0x020 - guid fffffffffff0000000000cf38db966ea - min ver 0x0
           PS2 - flags 0x020 - guid fffffffffff0000000000cf38db966ea - min ver 0x0
           PS3 - flags 0x020 - guid fffffffffff0000000000cf38db966ea - min ver 0x0
           PS4 - flags 0x020 - guid fffffffffff0000000000cf38db966ea - min ver 0x0
```

Listing 4. List of supported HPIMAGE update targets

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00FFFFC0  76 20 30 2E 31 2E 37 39 2B 20 32 35 2D 4A 75 6E  v 0.1.79+ 25-Jun
00FFFFD0  2D 32 30 31 35 00 FF FF FF FF FF FF FF FF FF FF  -2015...........
00FFFFE0  FF FF FF FF 00 00 01 00 00 00 00 00 00 00 00 00  ................
00FFFFF0  6B 09 7C 77 B3 00 00 2B BC FB 00 00 69 4C 4F 34  ............iLO4
```

Listing 5. Dump of an HPIMAGE footer

The key elements of the footer are:

- a "mirrored" blob header: `iLO4` magic at the end (`0x40`-byte long)
- `0xFBBC`: negative offset from the end of the file (`0x444`)
- This offset points to the **cryptographic parameters**, `0x404`-byte long. The `Python` definition of the `SignatureParams` structure is presented in the following listing:

```
class SignatureParams(LittleEndianStructure):

    _fields_ = [
        ("sig_size", c_uint),
        ("modulus", c_byte * 0x200),
        ("exponent", c_byte * 0x200)
    ]
```

The cryptographic parameters we just discovered are a key element of the integrity verification process. Let's see how they are used.

3.3 Module integrity check

`0x10000` bytes from the end of the file, one can find the `HPIMAGE` bootstrap code or bootloader (here in blue):

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00FEFFE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00FEFFF0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00FF0000  09 00 00 EA 7C 03 00 EA E1 07 00 EA D5 03 00 EA  ................
00FF0010  E7 03 00 EA FE FF FF EA 66 03 00 EA 0A 04 00 EA  ................
00FF0020  7C 0E FF FF A8 02 FF FF 10 80 00 D0 68 07 00 EB  ................
```

This is `ARM` code! More precisely it is an `ARM` bootloader.

When the `iLO` system boots up, this bootloader is responsible for loading (and integrity checking) modules (or sub-images) from the `HPIMAGE` blob. They are concatenated to the `HPIMAGE` header as a set of `IMG_HEADER`:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  69 4C 4F 34 20 76 20 32 2E 34 34 2E 37 20 31 39  iLO4 v 2.44.7 19
00000010  2D 4A 75 6C 2D 32 30 31 36 1A 00 FF FF FF FF FF  -Jul-2016......
00000020  08 00 00 10 F8 0A 00 00 57 5F 10 00 E0 02 68 01  ................
00000030  D3 EA D0 00 FF FF FF FF 00 00 00 00 FF FF FF FF  ................
00000040  68 3C 5A 2A E9 DF A1 6A C2 D6 96 43 85 54 4E D0  ................
[...]
```

Key elements:

- `iLO4` magic (in red)
- Version string (in blue)
- Images are signed (`RSA` signature)

– Three images for this firmware (kernel *main*, kernel *recovery*, userland)
– Possibly compressed (LZ-*like* algorithm found in the bootstrap code)

Once reversed, the IMG_HEADER structure can be defined using the ImgHeader Python class:

```
class ImgHeader(LittleEndianStructure):

    _fields_ = [
        ("ilO_magic", c_byte * 4),
        ("build_version", c_char * 0x1C),
        ("type", c_ushort),
        ("compression_type", c_ushort),
        ("field_24", c_uint),
        ("field_28", c_uint),
        ("decompressed_size", c_uint),
        ("raw_size", c_uint),
        ("load_address", c_uint),
        ("signature", c_byte * 0x200),
        ("padding", c_byte * 0x200)
    ]
```

The following listing presents the formatted output of the extraction tool for an example of ImgHeader structure:

```
[+] iLO Header 0: iLO4 v 2.44.7 19-Jul-2016
  > magic              : iLO4
  > build_version      :  v 2.44.7 19-Jul-2016
  > type               : 0x08
  > compression_type   : 0x1000
  > field_24           : 0xaf8
  > field_28           : 0x105f57
  > decompressed_size  : 0x16802e0
  > raw_size           : 0xd0ead3
  > load_address       : 0xffffffff
  > field_38           : 0x0
  > field_3C           : 0xffffffff
  > signature
0000  68 3c 5a 2a e9 df a1 6a c2 d6 96 43 85 54 4e d0   h<Z*...j...C.TN.
0010  c3 a4 e1 6f cb 2d 0f b6 0c 28 cd 31 88 db 07 6c   ...o.-...(.1...l
[...]
```

Having reverse-engineered and re-implemented the decompression algorithm, one has the surprise to discover an ELF file for the module above! The following listing shows the dump of the extracted ELF header.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010  02 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00  ..(.........4...
00000020  A0 A2 67 01 00 0C 17 00 34 00 20 00 66 02 28 00  .........4. .f.(.
00000030  68 02 67 02 01 00 00 00 F4 4C 00 00 00 00 00 00  h.g.............
```

What we have found so far is:

– A HPIMAGE blob is signed, verified by the x86 code (flashing tool)
– Collection of IMG_HEADER images
– Each of them is signed, verified by the ARM bootloader at startup, using the embedded public key (from the cryptographic parameters).

3.4 Signature check reimplementation

To validate our findings, it is possible to re-implement the integrity check. First, one need to extract the content of the update package (see listing 6).

The fingerprint is computed on the raw (possibly compressed) data and includes the first `0x40` bytes of the image header. In order to verify the `RSA` signature, the modulus and exponent are found in the cryptographic parameters structure; the signature is found in the dedicated field of the `ImgHeader` structure.

The `Ruby` code from listing 7 illustrates the re-implementation of the signature verification algorithm; its output is presented on listing 8.

```
$ ll ./extract/
total 39M
-rw-r--r-- 1 ilo ilo  63K Mar 15 16:55 bootloader.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 bootloader.hdr
-rw-r--r-- 1 ilo ilo 2.2K Mar 15 16:55 cert0.x509
-rw-r--r-- 1 ilo ilo 1.7K Mar 15 16:55 cert1.x509
-rw-r--r-- 1 ilo ilo 1.4K Mar 15 16:55 cert2.x509
-rw-r--r-- 1 ilo ilo  23M Mar 15 16:55 elf.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 elf.hdr
-rw-r--r-- 1 ilo ilo  14M Mar 15 16:55 elf.raw
-rw-r--r-- 1 ilo ilo  512 Mar 15 16:55 elf.sig
-rw-r--r-- 1 ilo ilo 1.2K Mar 15 16:55 hpimage.hdr
-rw-r--r-- 1 ilo ilo  320 Mar 15 16:55 ilo4_244.bin.map
-rw-r--r-- 1 ilo ilo 770K Mar 15 16:55 kernel_main.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 kernel_main.hdr
-rw-r--r-- 1 ilo ilo 471K Mar 15 16:55 kernel_main.raw
-rw-r--r-- 1 ilo ilo  512 Mar 15 16:55 kernel_main.sig
-rw-r--r-- 1 ilo ilo 770K Mar 15 16:55 kernel_recovery.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 kernel_recovery.hdr
-rw-r--r-- 1 ilo ilo 471K Mar 15 16:55 kernel_recovery.raw
-rw-r--r-- 1 ilo ilo  512 Mar 15 16:55 kernel_recovery.sig
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 sign_params.raw
```

Listing 6. Directory listing of extracted files

```ruby
# read stored signature and compute fingerprint on data (sha512)
def fingerprint(path, basename)
    puts "[+] compute #{basename} fingerprint\n"
    digest = Digest::SHA2.new(bitlen=512)

    # read header
    File.open("kernel_main.hdr", 'rb'){|fd|
        digest << fd.read(0x40)
    }

    # read blob
    File.open("kernel_main.raw", 'rb'){|fd|
        blob = fd.read()
        # append blob size and data
        digest << [blob.size].pack('L')
        digest << blob
    }
    puts "\n> digest:\n#{digest.hexdigest}"
endr

# verify the signature
def verify_sig(s, n, e)
    puts "[+] verify signature\n"
    puts "\n> s:\n#{s.to_s(16)}"
    puts "\n> n:\n#{n.to_s(16)}"
    puts "\n> e:\n#{e.to_s(16)}"

    m = s.to_bn.mod_exp(e, n)
    puts "\n> m:\n#{m.to_s(16)}\n"

    sig = [m.to_s(16)].pack("H*").unpack('C*')
    raise '[x] invalid sig' unless (sig.shift == 0x01)

    loop do
        b = sig.shift
        break if (b != 0xFF)
    end

    puts  "\n> output:\n#{sig.map{|i| "%02x" % i}.join()}\n\n"
end
```

Listing 7. Integrity check implementation

```
>ruby signature.rb ./extract/kernel_main.sig
[+] load crypto parameters
    > signature size: 4096
[+] load signature
[+] verify signature

> s:
a9a9c82179e1429485c6251a1cb2649f4a0fb2bff1fae8f028b4a26fda59d6e690d2431c422a3f
[...]
0626a93674e524be3c4971ab267deb87b332d80035f9b61457b6a46677c184ea83d55944a0b3f9
ad8e24b81e

> n:
d34b4cc0d6d3a0e01fc1d06909c5ba303ffd320492ac3c2418843c03d8e4402c387353405bf51d
[...]
04f92553bdc4f3363113114dceb7dbabfe4d013be144bd82db756969f476690b0036734e6236f5
0bb186d28b

> e:
10001

> m:
01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
[...]
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00BB017DE214F82D0C189B9CB50548219B8A316C9611
1666E318229A5E47C2BB351E9CCA0FF79F30D525F0D96BE88D2C372FA10B1638F791267AE3E132
679AEE65

> output:
bb017de214f82d0c189b9cb50548219b8a316c96111666e318229a5e47c2bb351e9cca0ff79f30
d525f0d96be88d2c372fa10b1638f791267ae3e132679aee65

[+] computed kernel_main fingerprint

> digest:
bb017de214f82d0c189b9cb50548219b8a316c96111666e318229a5e47c2bb351e9cca0ff79f30
d525f0d96be88d2c372fa10b1638f791267ae3e132679aee65
```

Listing 8. Integrity check output

iLO4 does not implement any kind of hardware root of trust. If one is able to bypass the "*HP Signed file*" envelope signature check; then the bootloader code only relies upon the cryptographic parameters it embeds in order to verify the integrity of the modules it loads. If an attacker was able to write its own firmware directly on the flash chip, they could remove the signature checks or embed its own public key.

4 Web server vulnerability

Once the firmware update file format has been understood, its various components can be loaded in a disassembler for a proper security study.

We focused on the web server, as it is usually enabled to allow an easy iLO administration.

It supports both HTTP and HTTPS and runs four concurrent threads to handle connections. Once a client is connected, one of the threads starts parsing the data it receives line by line, by using several string parsing functions from the libc, such as strstr(), strcmp() and sscanf().

We noticed a bad usage of sscanf() when parsing the Connection header, as highlighted in the following listing:

```
else if ( !strnicmp(request, http_header, "Content-length:", 0xFu) )
{
  content_length = 0;
  sscanf(http_header, "%*s %d", &content_length);
  state_set_content_length(global_struct_, content_length);
}
else if ( !strnicmp(request, http_header, "Cookie:", 7u) )
{
  cookie_buffer = state_get_cookie_buffer(global_struct_);
  parse_cookie(request, http_header, cookie_buffer);
}
else if ( !strnicmp(request, http_header, "Connection:", 0xBu) )
{
  sscanf(http_header, "%*s %s", https_connection->connection);
}
```

The connection buffer from the https_connection object is only 16 bytes long. Providing a Connection header larger than 16 bytes triggers a buffer overflow allowing to overwrite the content of the object.

We identified the object layout in memory, and found two interesting values to overwrite: the localConnection boolean, which indicates if a connection comes from the network or directly from the host; and the vtable, which holds the object's virtual functions pointers. These values are described in the following listing:

```
struct https_connection {
        ...
        0x0C: char connection[0x10];
        ...
        0x28: char localConnection;
        ...
        0xB8: void *vtable;
}
```

Indeed, a very simple and stable exploitation consists in sending a `Connection` header containing 29 random characters. The overflow will reach the `localConnection` boolean, setting it to a non-zero value. This is sufficient to allow unauthenticated access to several pages, including the `Rest API` endpoint.

Gaining arbitrary code execution is a bit harder, as we have to overwrite the `vtable` pointer to make it point to a known place containing arbitrary function pointers. The first observation we made was that there was no defense-in-depth mechanism such as `NX` or `ASLR`. We then noticed that each web server thread uses a working buffer located in the binary `.data` section, in which each line received is stored before being parsed. We thus are able to control this working buffer content, and can use it to store a fake `vtable` and a shellcode, gaining effective code execution.

We developed a proof-of-concept exploit reading the content of the file containing the cleartext users credentials (`i:/vol0/cfg/cfg_users.bin`):

```
$ python exploit_get_users.py 192.168.42.78 250
[*] Connecting to 192.168.42.78...
[+] Connected
[*] Assembling shellcode...
[*] Preparing shellcode headers...
[*] Preparing fake vtable...
[*] Preparing fake vtable headers...
[*] Preparing XML request...
[*] Sending 1094d bytes...
[+] Request XML sent
[*] XML data retrieved
[*] Found iLO version 2.50
[*] Preparing request 2...
[*] Sending 109f9 bytes...
[+] Request 2 sent
[+] User 01: [Administrator] [Administrator] [G......7]
[+] User 02: [admin] [admin] [passw0rd]
```

5 `iLO` to host

Once we compromised the `iLO` system through its web server, our objective was to pivot from there and gain access to the host operating system. During our investigations and analysis of the system, we took a look at a specific task: the **Channel Interface** (`CHIF`) task.

5.1 Access to the host memory

While reversing the CHIF task, we found mentions of *Windows Hardware Error Architecture* (WHEA [4]) records parsing in the log messages of the task:

```
whea: invalid info from SMBIOS type_229 : offset=%X, size=%X
whea: found whea_info at %p
whea: NO $WHE found!
[...]
whea: sawbase access failed
[...]
whea : re-running whea HostRAM detect
```

From a functional point of view, WHEA events are generated at host operating system level. Later on, a task of the iLO system is trying to parse them. It means a communication channel exists between the server main processor/memory and the iLO system.

Now, what is "*SMBIOS type_229*"? *System Management BIOS* (SMBIOS [2]) defines a set of interfaces (data structures and access points) used to expose information from the system firmware (BIOS). Various types of information are defined; type 0 describes BIOS Information for example. Types 0 through 127 are reserved and defined in the specification. Types 128 through 256 are OEM specific information.

Type 229 is OEM defined and thus undocumented up to our knowledge. Still, it is possible to dump the SMBIOS interfaces from the host operating system (here a Linux):

```
# dmidecode -t 229
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.

Handle 0xE500, DMI type 229, 100 bytes
OEM-specific Type
    Header and Data:
        E5 64 00 E5 24 44 46 43 00 50 FE F1 00 00 00 00
        00 04 00 00 24 43 52 50 00 50 F9 F1 00 00 00 00
        00 00 05 00 24 48 44 44 00 30 F9 F1 00 00 00 00
        00 20 00 00 24 4F 43 53 00 F0 F8 F1 00 00 00 00
        00 40 00 00 24 4F 43 42 00 F0 F7 F1 00 00 00 00
        00 00 01 00 24 53 41 45 00 E0 F7 F1 00 00 00 00
        00 10 00 00

0000  24 44 46 43 00 50 fe f1 00 00 00 00 00 04 00 00   $DFC.P..........
0010  24 43 52 50 00 50 f9 f1 00 00 00 00 00 00 05 00   $CRP.P..........
0020  24 48 44 44 00 30 f9 f1 00 00 00 00 00 20 00 00   $HDD.0....... ..
0030  24 4f 43 53 00 f0 f8 f1 00 00 00 00 00 40 00 00   $OCS.........@..
0040  24 4f 43 42 00 f0 f7 f1 00 00 00 00 00 00 01 00   $OCB............
0050  24 53 41 45 00 e0 f7 f1 00 00 00 00 00 10 00 00   $SAE............
```

Each entry seems 16 bytes long. The highlighted bytes look like 64-bit pointers. The following listing provides a C structure definition of these "type 229" entries based on our analysis:

```
struct entry229 {
    char tag[4];
    void *pointer64;
    int flags;
}
```

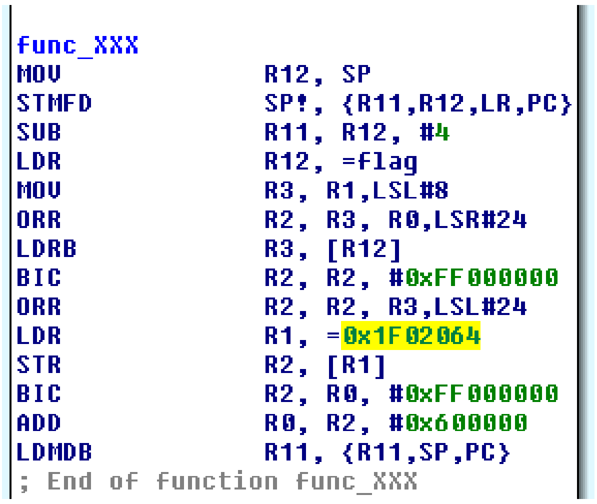
Let’s check one of these pointers in physical memory:

```
root@ilo-server-ubuntu:~# xxd -s $((0xf1f95000)) /dev/mem|head -n 8

f1f95000: 2452 4253 0000 0000 0001 0069 0813 0400  $RBS.......i....
f1f95010: 0113 0400 0101 6f00 0000 0001 6752 4f4d  ......o.....gROM
f1f95020: 2d42 6173 6564 2053 6574 7570 2055 7469  -Based Setup Uti
f1f95030: 6c69 7479 2c20 5665 7273 696f 6e20 332e  lity, Version 3.
f1f95040: 3030 0d0a 436f 7079 7269 6768 7420 3139  00..Copyright 19
f1f95050: 3832 2c20 3230 3135 2048 6577 6c65 7474  82, 2015 Hewlett
f1f95060: 2d50 6163 6b61 7264 2044 6576 656c 6f70  -Packard Develop
f1f95070: 6d65 6e74 2043 6f6d 7061 6e79 2c20 4c2e  ment Company, L.
```

Back to the CHIF task, WHEA entries are accessed using a very specific pattern; see func_XXX from the following C code:

```
char whea_header[0x18];
int *ptr_entry = find_in_smbios_229("$WHE");
if (ptr_entry) {
    int phy_ptr_low = ptr_entry[1];
    int phy_ptr_high = ptr_entry[2];

    void *whea_ptr = func_XXX(phy_ptr_low, phy_ptr_high);
    sawbase_memcpy_s(whea_header, whea_ptr, 0x18);
    [...]
}
```

At assembly level, func_XXX involves interesting hardcoded addresses (see assembly listing on figure 4).



```
func_XXX
MOV             R12, SP
STMFD           SP!, {R11,R12,LR,PC}
SUB             R11, R12, #4
LDR             R12, =flag
MOV             R3, R1,LSL#8
ORR             R2, R3, R0,LSR#24
LDRB            R3, [R12]
BIC             R2, R2, #0xFF000000
ORR             R2, R2, R3,LSL#24
LDR             R1, =0x1F02064
STR             R2, [R1]
BIC             R2, R0, #0xFF000000
ADD             R0, R2, #0x600000
LDMDB           R11, {R11,SP,PC}
; End of function func_XXX
```

Fig. 4. Hardcoded address 0x1F02064

The function is equivalent to the following `C` code:

```c
void *func_XXX(void *ptr_low, void *ptr_hi) {
    char flag = 2;
    int magic = (flag<<24) |
                (((ptr_hi << 8) | (ptr_low >> 24)) & 0x00ffffff);
    *(0x1f02064) = magic;
    return (ptr_low & 0x00ffffff) | 0x600000;
}
```

We have few answers and many questions about `func_XXX`:

– The passed 64-bits pointer is truncated to a 16MB boundary
– An unknown flag is set to 2
– What is mapped at `0x1F02064`?
– What is mapped at `0x600000`?

5.2 Memory regions

The `Integrity` kernel offers an interesting concept of `Memory Region`. A `Memory Region` object is used to map a physical memory region into the virtual space of a task. The `C` code proposed demonstrates how a memory region is instantiated:

```c
sprintf(mr_name, "MR%X", mr_physical >> 12);
RequestResource(&mr_object, mr_name, "!systempassword");
```

`RequestResource` initializes and sends a request to the kernel. It is made of the following elements:

– A verb, e.g. "`procure`"
– The name of the object, e.g. "`MR80200`"
– A password, e.g. "`!systempassword`"

Each task has a list of `Memory Region` which can be mapped in its virtual memory, by calling `memmap()`. The `CHIF` task maps the following `Memory Regions`:

Physical	Virtual	Size	
0x80000000	0x1F00000	0x1000	MR80000
0x800F0000	0x1F01000	0x1000	MR800F0
0x80200000	0x1F02000	0x1000	MR80200
0x802F0000	0x1F03000	0x1000	MR802F0
0x804F0000	0x1F07000	0x1000	MR804F0
0x82000000	0x600000	0x1000000	MR82000
0xC0000000	0x1F10000	0x1000	MRC0000
0xD1000000	0x1F14000	0x1000	MRD1000

We now have our virtual ⇔ physical mapping:

- 0x1F02064 is the mapping of 0x80200064
- 0x600000 is the mapping of 0x82000000

Furthermore, 0x1F02000 is known to contain PCI registers mappings. That's something we have learned from the dbug.html page exposed by the iLO web server (see listing 9). The two highlighted addresses are close to the one we identified in func_XXX. Our assumption is that they have a related semantics and thus that the address 0x1F02064 is a memory mapping of an unknown PCI register.

```
1f01006         Fn0 PCI-E Status Reg  CSMPCISR
1f01010         Fn0 PCI-E I/O BAR
1f010ca         Fn0 PCI-E Device Status Reg
1f02078         PCI-E Err Stat Reg PERSTAT
1f020b4         Sys Flt Stat Reg SYSFAULT
1f03006         Fn2 PCI-E Status Reg CHIFPCISR
1f03010         Fn2 PCI-E I/O BAR
1f030ca         Fn2 PCI-E Device Status Reg
1f05006         Fn3 PCI-E Status Reg  WDGPCISR
1f050ca         Fn3 PCI-E Device Status Reg
1f07006         Fn4 PCI-E Status Reg UHCIPCISR
1f070ca         Fn4 PCI-E Device Status Reg
1f09006         Fn5 PCI-E Status Reg  VSPPCISR
1f09010         Fn5 PCI-E I/O BAR
1f090ca         Fn5 PCI-E Device Status Reg
1f0b006         Fn6 PCI-E Status Reg IPMIPCISR
1f0b010         Fn6 PCI-E Memory BAR
1f0b0ca         Fn6 PCI-E Device Status Reg
```

Listing 9. PCI registers mapping

We have enough knowledge to re-implement the same technique in our shellcode. Our objective is to fill a 16MB Memory Region with host memory. The following procedure can then be applied:

- Take a **host** physical memory address
- Shift it right by 24
- Add flag
- Write the value in register 0x1F02064
- ??? (Unknown behavior on hardware side)
- Profit by accessing MR82000!

Weaponizing this technique, we are able to map the physical memory of the host (main server) with read/write access. It opens a wide range of possibilities such as rebuilding the memory mapping of the system, or injecting code into the host system.

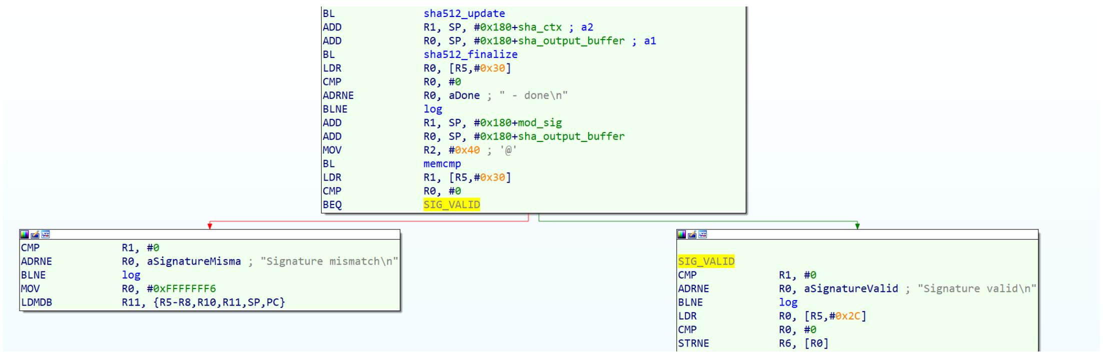
With persistence in mind, our idea is to implement a two-way communication channel over this mapped memory, offering command execution on the host server. A typical scenario for this would be an attacker using the iLO vulnerability to achieve cross-domains pivot between administration and production VLANs for example.

6 Crafting a backdoored firmware

Here, the objective is to craft a modified firmware to embed a backdoor for example. For this, many components of the update package are going to be patched. The simplest way is to patch them "in-place" in the binary file; we simply overwrite the content and fix the headers of the patched components. To facilitate this approach, our extraction script generates a map of all the offsets where the components are found:

```
[+] Firmware offset map
    >        HP_SIGNED_FILE at 0x00000000
    >              HP_CERT0 at 0x0000020f
    >              HP_CERT1 at 0x00000ab3
    >              HP_CERT2 at 0x0000112e
    >           HPIMAGE_HDR at 0x00001664
    >        BOOTLOADER_HDR at 0x00001b1c
    >            BOOTLOADER at 0x00ff1b1c
    >               ELF_HDR at 0x00001f5c
    >                   ELF at 0x0000239c
    >       KERNEL_MAIN_HDR at 0x00ef1b1c
    >           KERNEL_MAIN at 0x00ef1f5c
    >   KERNEL_RECOVERY_HDR at 0x00f71b1c
    >       KERNEL_RECOVERY at 0x00f71f5c
```

For this example we choose to insert our backdoor in the userland component, the ELF file. Patching the integrity checks is only a matter of changing a single conditional jump in the bootloader and kernel components (see figure 5).



```
BL              sha512_update
ADD             R1, SP, #0x180+sha_ctx ; a2
ADD             R0, SP, #0x180+sha_output_buffer ; a1
BL              sha512_finalize
LDR             R0, [R5,#0x30]
CMP             R0, #0
ADRNE           R0, aDone ; " - done\n"
BLNE            log
ADD             R1, SP, #0x180+mod_sig
ADD             R0, SP, #0x180+sha_output_buffer
MOV             R2, #0x40 ; '@'
BL              memcmp
LDR             R1, [R5,#0x30]
CMP             R0, #0
BEQ             SIG_VALID
```

```
CMP             R1, #0
ADRNE           R0, aSignatureMisma ; "Signature mismatch\n"
BLNE            log
MOV             R0, #0xFFFFFFF6
LDMDB           R11, {R5-R8,R10,R11,SP,PC}
```

```
SIG_VALID
CMP             R1, #0
ADRNE           R0, aSignatureValid ; "Signature valid\n"
BLNE            log
LDR             R0, [R5,#0x2C]
CMP             R0, #0
STRNE           R6, [R0]
```

Fig. 5. Signature check implementation

A description of the bootloader patch regarding the integrity check is provided in Python:

```python
# Patch signature check : BNE XX -> MOV R0, #0
PATCH = {"offset": 0x38BC, "size": 4, "prev_data": "4000001A", "
    patch": "0000A0E1"}
```

The high-level methodology is simple:

- Extract (and decompress when needed) all the components
- Patch integrity check in the bootloader
- Patch integrity check in the kernel
- Modify the ELF image to embed our backdoor code
- Re-compress modified components when needed
- Write modified components in the binary update file, update their headers
- Flash iLO with modified firmware

7 iLOshell

Our high-level objective is to craft a backdoored firmware exposing a two-way communication channel offering command execution on the host server. For this, we will reuse existing iLO features as much as possible. Using the web server endpoint seems the most efficient and reasonably stealth way to do so.

The idea is to hook or reuse existing handlers of the web server to expose the following functionalities:

- Communication channel setup
- Command execution over the communication channel (send command and receive answer)
- Communication channel removal

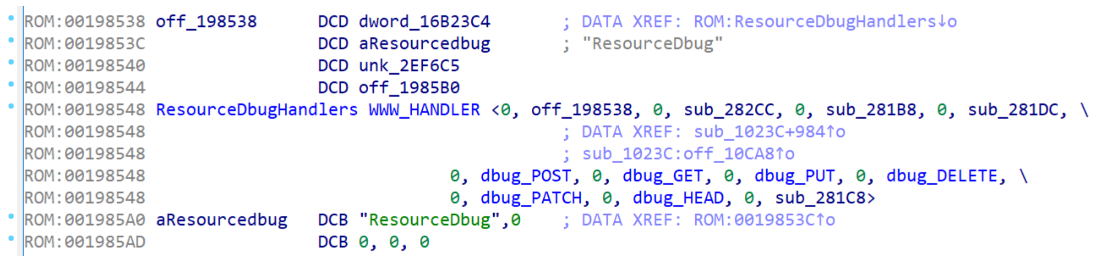
7.1 Backdooring the firmware

The web server code can be found in the webserv.elf section of the ELF userland Integrity image. A large number of handlers are exposed by the web server, a few of them are given below for illustration purpose:

- /dbug.html
- /dispatch
- /favicon.ico

- `/html/admin_manage.html`
- `/html/admin_security_HPsso.html`
- `/html/help.html`
- `/html/iLO.ico`
- `/html/info_blade.html`
- ...

Each of these handlers is described internally by a structure which mostly contains callbacks for `HTML` methods: `POST`, `PUT`, `DELETE`, `GET`, `HEAD`, *etc.* (see figure 6).



```
ROM:00198538 off_198538      DCD dword_16B23C4      ; DATA XREF: ROM:ResourceDbugHandlers↓o
ROM:0019853C                 DCD aResourcedbug      ; "ResourceDbug"
ROM:00198540                 DCD unk_2EF6C5
ROM:00198544                 DCD off_1985B0
ROM:00198548 ResourceDbugHandlers WWW_HANDLER <0, off_198538, 0, sub_282CC, 0, sub_281B8, 0, sub_281DC, \
ROM:00198548                                        ; DATA XREF: sub_1023C+984↑o
ROM:00198548                                        ; sub_1023C:off_10CA8↑o
ROM:00198548                             0, dbug_POST, 0, dbug_GET, 0, dbug_PUT, 0, dbug_DELETE, \
ROM:00198548                             0, dbug_PATCH, 0, dbug_HEAD, 0, sub_281C8>
ROM:001985A0 aResourcedbug   DCB "ResourceDbug",0   ; DATA XREF: ROM:0019853C↑o
ROM:001985AD                 DCB 0, 0, 0
```

Fig. 6. *Dbug* handler callbacks definition

The callbacks seem like a perfect place to insert our backdoor code, relying upon the web server features to handle the lower-level (socket level) communications.

7.2 Linux Kernel Shellcode

On the `iLO` system, our backdoor code runs in the web server task as a hooked handler. We need to inject code in the host system (a `Linux` system for this example), to be able to: run arbitrary commands, wait for commands completion and return the outputs.

The technique we have used so far to inject code into the host system is to overwrite unused kernel functions and then to hijack an entry of the syscall table in order to redirect the execution flow to our injected shellcode. Our code is thus executed in kernel mode. This is enough for one-shot execution like spawning a shell, however we now want to be persistent and to execute commands at userland level as well.

Two technical issues need to be solved:

- Kernel persistence
- Run code in userland from kernel, wait for its completion and retrieve its output.

The first point is easily solved using kthread. Once executed our kernel shellcode will migrate its code into a newly created kthread.

To solve the second point, we reuse the technique presented by Ben Seri and Alon Livne [7]. It simply relies on the dedicated Linux kernel primitive: call_usermodehelper[9].

```
int call_usermodehelper ( const char * path,
    char ** argv,
    char ** envp,
    int wait);
```

This helper gracefully allows us to execute a command in userland. Passed with the appropriate value, the wait parameter allows us to wait for command completion. For the sake of simplicity the command outputs its result into a file that is then read from kernel-land.

7.3 Communication channel

The communication channel between the iLO system and the host system is built upon a shared memory page. It takes advantage of the ability of the iLO to read arbitrary physical memory of the host. At high- level:

- iLO-side backdoor writes a message about new commands to execute
- Linux-side backdoor executes commands and writes the outputs into the shared memory

In order to setup the shared memory region, the kernel shellcode will allocate a new 1MB memory region, retrieve its physical address, and write it in a memory location related to itself. As the iLO knows the shellcode physical address, it will be able to retrieve the shared memory address.

On the iLO side, the physical memory address will be retrieved so that it can be mapped for read and write accesses.

We define the channel structure to describe the memory page:

```
struct channel {
        int available_input;
        int input_len;
        char input[4096];
        int available_output;
        int output_len;
        char output[];
}
```

[9] https://www.kernel.org/doc/htmldocs/kernel-api/API-call-usermodehelper.html

On the iLO side, when a new shell command is received, it gets written to the input buffer, the input_len value is updated and the available_input flag is updated to 1. The iLO then waits for the available_output flag to be 1, and sends back the output buffer content according to the output_len size.

On the Linux kernel side, the backdoor thread waits for input data by monitoring the available_input flag. It then calls the call_usermodehelper and redirects the command output to a temporary file. After the command completion, the temporary file is read and deleted, and its content is written to the output buffer. Finally, the output_len field is updated, and the available_output flag is set.

To be able to control the Linux kernel shellcode, we also defined a magic value that can be written in the available_input field. Such magic value can be used to terminate the kernel thread and free the shared memory region once we want the backdoor to be removed.

```
$ python backdoor_client.py 192.168.42.78
[+] iLO Backdoor found
[-] Linux Backdoor not detected

==============================================================

Welcome to the iLO Backdoor Commander.

    detect_backdoor(): checks for the backdoor presence on iLO and
        the Linux host
    install_linux_backdoor(): installs the Linux kernel backdoor if
        not present
    cmd(CMD): executes a Linux shell command
    remove_linux_backdoor(): removes the backdoor

Example:
    ib.detect_backdoor()
    ib.install_linux_backdoor()
    ib.cmd("/usr/bin/id")
    ib.remove_linux_backdoor()

=============================================================


Python 2.7.14+ (default, Mar 13 2018, 15:23:44)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
    information.
(InteractiveConsole)
>>> ib.install_linux_backdoor()
[*] Dumping kernel...
[+] Dumped 1000000 bytes!
[+] Found syscall table @0xffffffff81a001c0
[+] Found sys_read @0xffffffff8121e510
[+] Found call_usermodehelper @0xffffffff81098520
[+] Found serial8250_do_pm @0xffffffff81528760
[+] Found kthread_create_on_node @0xffffffff810a2000
[+] Found wake_up_process @0xffffffff810ad860
[+] Found __kmalloc @0xffffffff811f8c50
[+] Found slow_virt_to_phys @0xffffffff8106c6a0
[+] Found msleep @0xffffffff810f0050
[+] Found strcat @0xffffffff8140c9c0
[+] Found kernel_read_file_from_path @0xffffffff812236e0
[+] Found vfree @0xffffffff811d7f90
[+] Shellcode written
[+] iLO Backdoor found
[+] Linux Backdoor found
>>> ib.cmd("/usr/bin/id")
[+] Found shared memory page! 0xe8200000 / 0xffff8800e8200000
uid=0(root) gid=0(root) groups=0(root)

>>> ib.cmd("head /etc/shadow")
root:!:16758:0:99999:7:::
daemon:*:17268:0:99999:7:::
bin:*:17268:0:99999:7:::
sys:*:17268:0:99999:7:::
sync:*:17268:0:99999:7:::
games:*:17268:0:99999:7:::
man:*:17268:0:99999:7:::
lp:*:17268:0:99999:7:::
mail:*:17268:0:99999:7:::
news:*:17268:0:99999:7:::

>>> ib.remove_linux_backdoor()
```

Listing 10. iLO backdoor client

8 Detecting firmware compromise

So far we have seen that the lack of hardware root of trust leaves the system widely vulnerable to a persistent backdoor at firmware level. As a defender, one could use the same privileged access to the iLO system offered by the exploitation of the web server vulnerability to read the content of the flash and attempt to validate its content.

For this purpose, a script was developed to automatize the process of flash dumping using the RCE vulnerability and comparing to known "good" digests.

```
$ python exploit_check_flash.py 192.168.42.78 250
[*] Connecting to 192.168.42.78...
[+] Connected
[+] Request XML sent
[*] XML data retrieved
[*] Found iLO version 2.50
[+] Request 2 sent
[*] 0x00000000 bytes...
[*] 0x00000400 bytes...
[*] 0x00000800 bytes...
[...]
[*] 0x00fff800 bytes...
[*] 0x00fffc00 bytes...
[+] Flash contains iLO4 version 250

$ python exploit_check_flash.py 192.168.42.78 250
[*] Connecting to 192.168.42.78...
[+] Connected
[+] Request XML sent
[*] XML data retrieved
[*] Found iLO version 2.50
[+] Request 2 sent
[*] 0x00000000 bytes...
[...]
[*] 0x00fffc00 bytes...
[-] Unknown firmware dumped! This might indicate a backdoor!
```

Listing 11. Firmware integrity check

This is a best effort attempt to provide a simple and practical way of checking the firmware integrity. Still, as always with backdoor/rootkit detection, it is a race to the lowest levels. In this example, we perform a read of the content of the flash from a userland task. This userland uses an interface provided by the `SpiService` service, which in turn makes syscalls to the kernel. In case of a compromised firmware, one of these components may hook the read function and hide sensitive modifications.

9 Conclusion

`BMC`, and `iLO` systems in particular, are complex and powerful. They offer many services and features, at the cost of a significant attack surface. During the course of this study, the authors discovered a critical vulnerability in the web server component of `iLO4`. Although fixed by the vendor, it offers a trivial remote authentication bypass and full compromise of both the `iLO` and the host systems.

If they are not actively used, completely disabling the feature is a good practice. Otherwise, administrators should take great care to keep their systems up to date whenever possible. Network-level isolation should be put in place to ensure that `iLO` systems can only be accessed from dedicated administration `VLAN`s.

We use the web server vulnerability and its related code execution primitive as a foothold on the `iLO` system; trying to install ourselves persistently on the system. As demonstrated in this paper, `iLO4` systems offer perfect, highly stealth, long term persistence capabilities to a motivated attacker; mostly due to the lack of hardware root of trust and to our privileged access to the SPI service. Indeed, thanks to our code execution primitive we were able to bypass the signature check performed by the installed firmware and to flash our rogue firmware. From there, the chain of trust relies upon the bootloader, which we have compromised.

It also means that in case of a compromise, wiping and reinstalling the host operating system is not sufficient: the hardware should be considered untrusted as well. This sensible security gap is advertised to be fixed with the release of iLO5 systems and Proliant Gen10 servers, bundled with a feature named *silicon root of trust*.

Platform security awareness is slowly gaining more and more attention. Long term efforts such as the CHIPSEC framework [8] or more recently published projects like Titan from Google [1] are good illustrations. Each independent computational unit is a potential target for the attackers and thus has to be taken into consideration in the security model.

The authors would like to thank the Synacktiv and Airbus Digital Security teams for their insightful reviews and comments.

References

1. Google Cloud Platform Blog. Titan in depth: Security in plaintext. https://cloudplatform.googleblog.com/2017/08/Titan-in-depth-security-in-plaintext.html, 2017.
2. Distributed Management Task Force Inc. (DMTF). System Management BIOS (SMBIOS) Reference Specification Version: 3.1.1. https://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.1.1.pdf, 2017.
3. Dan Farmer. IPMI: freight train to hell. http://fish2.com/ipmi/itrain.pdf, 2013.
4. Microsoft. Introduction to the Windows Hardware Error Architecture. https://docs.microsoft.com/en-us/windows-hardware/drivers/whea/introduction-to-the-windows-hardware-error-architecture, 2017.
5. HD Moore. A Penetration Tester's Guide to IPMI and BMCs. https://blog.rapid7.com/2013/07/02/a-penetration-testers-guide-to-ipmi/, 2013.
6. 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.
7. Ben Seri and Alon Livne. Exploiting BlueBorne in Linux-based IoT devices. https://www.blackhat.com/docs/eu-17/materials/eu-17-Seri-BlueBorne-A-New-Class-Of-Airborne-Attacks-Compromising-Any-Bluetooth-Enabled-Linux-IoT-Device-wp.pdf, 2017.
8. CHIPSEC. CHIPSEC: Platform Security Assessment Framework. https://github.com/chipsec/chipsec, 2014-2018.
9. Common Vulnerabilities and Exposures (CVE). CVE-2017-12542. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12542, 2017.