Backdooring your server through its BMC : the HPE iLO4 case

Fabien Périgaud, Alexandre Gazet & Joffrey Czarny

Rennes, 13-15 Juin, 2018

SYNACKTIV
DIGITAL SECURITY

AIRBUS

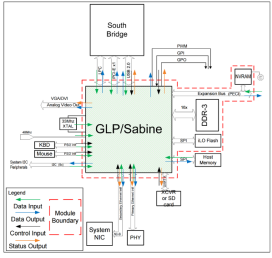
Plan

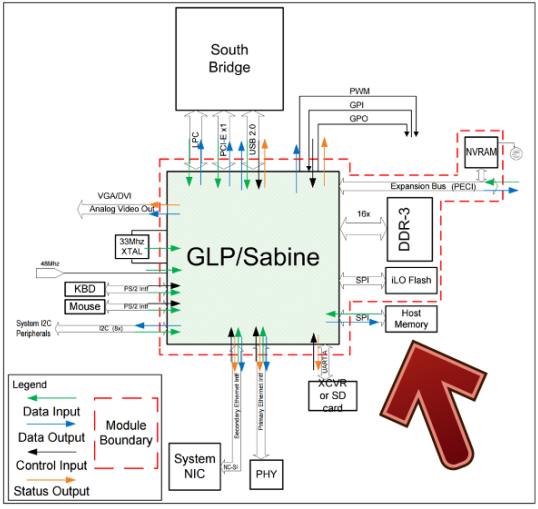
- Baseboard Management Controller (BMC) embarqué sur la plupart des serveurs HP depuis 10 ans

Niveau matériel (1/2)

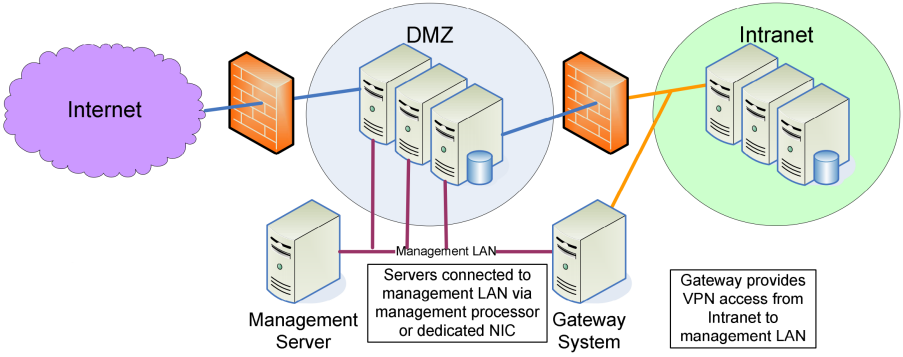
Système **autonome/indépendant** :

- Processeur `ARM` dédié : architecture `GLP/Sabine`
- Micrologiciel stocké sur une flash `NAND`
- Module `RAM` dédié
- Interface réseau dédiée
- Système d'exploitation complet et applicatif
- **Démarré dès qu'une alimentation éléctrique est branchée**



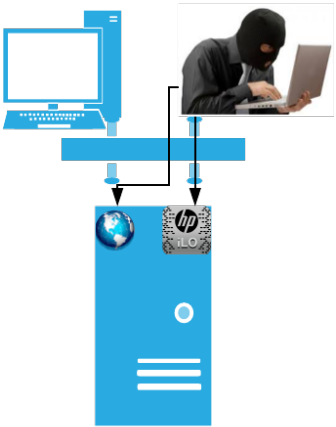


iLO est directement connecté sur le bus PCI-Express



Source : *Managing HP servers through firewalls with Insight Software*[1]

1.ftp://ftp.hp.com/pub/c-products/servers/management/hpsim/hpsim-53-managing-firewalls.pdf


iLO
Server

Démo

- Analyse du format des mises à jour du micrologiciel
- Extraction des composants : bootloader, noyau, image userland, signatures, *etc.*
- Analyse du noyau `Integrity`
- Reconstruction de l'espace d'adressage mémoire des modules userland (équivalent des processus)
- Analyse du Service Web (interface web d'administration)
- Temps total de l'étude, environ 5 mois-hommes

Publication et outillage

- https://recon.cx/2018/brussels/talks/subvert_server_bmc.html
- https://github.com/airbus-seclab/ilo4_toolbox

Une vulnérabilité critique identifiée

- CVE-2017-12542, CVSSv3 9.8
- **Contournement du mécanisme d'authentification** et **exécution de code arbitraire à distance**
- Corrigée dans la version 2.53 (buggy) et 2.54 d'iLO 4

Compromission complète du serveur

- Exécution de code arbitraire sur l'iLO, dans le contexte du serveur web
- Chemin d'attaque iLO vers hôte

Vulnérabilité située dans le serveur web

- Traitement des requêtes `HTTP` ligne par ligne
- Usage massif de fonctions de traitement de chaînes de caractères
  - `strstr()`
  - `strcmp()`
  - `sscanf()`
- Traiter des chaînes de caractères en `C` est **compliqué et/ou risqué**

Comment utiliser correctement `sscanf()` ?

```
1  else if ( !strnicmp(request, http_header, "Content-length:", 0xFu) )
2  {
3    content_length = 0;
4    sscanf(http_header, "%*s %d", &content_length);
5    state_set_content_length(global_struct_, content_length);
6  }
7  else if ( !strnicmp(request, http_header, "Authorization:", 0xEu) )
8  {
9    sscanf(http_header, "%*s %15s %16383s", method, encoded_credentials);
10   handle_authorization_credentials(method, encoded_credentials);
11 }
12 else if ( !strnicmp(request, http_header, "Connection:", 0xBu) )
13 {
14   sscanf(http_header, "%*s %s", https_connection->connection);
15 }
```

Débordement de tampon

La vulnérabilité permet de faire déborder le tampon `connection` d'un objet `https_connection`.

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

La vulnérabilité permet de faire déborder le tampon `connection` d'un objet `https_connection`.

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

Double ration de frites !

- Écrasement du booléen `localConnection` : permet l'accès à l'`API REST` **sans authentification**

  `curl -H "Connection: AAAAAAAAAAAAAAAAAAAAAAAAAAAA" :)`

La vulnérabilité permet de faire déborder le tampon `connection` d'un objet `https_connection`.

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

Double ration de frites !

- Écrasement du booléen `localConnection` : permet l'accès à l'`API REST` **sans authentification**

  `curl -H "Connection: AAAAAAAAAAAAAAAAAAAAAAAAAAAA" :)`
- Écrasement du pointeur vers la `vtable` : **exécution de code arbitraire**
  - Pas de `NX` ni `ASLR`
  - Tampon de travail du serveur web à une adresse fixe

Analyse de l'un des modules : CHIF

- Gère la communication bidirectionnelle entre l'hôte et l'iLO
- Dispose de fonctionnalités pour récupérer les informations WHEA
- Accès direct en lecture à la mémoire centrale du serveur

Analyse du mécanisme

- 16Mo de mémoire de l'hôte peuvent être mappés dans la mémoire de l'iLO via l'utilisation d'un registre PCI inconnu
- L'écriture dans cette zone mappée est répercutée dans la mémoire de l'hôte
- Réimplémentation de ce mécanisme dans un shellcode exécuté dans le contexte du serveur web de l'iLO

Réalisations :

- Compromission complète de la plateforme
- Exécution de code arbitraire sur le `iLO` **et** sur l'hôte
- Primitives d'accès en `RW` à la mémoire de l'hôte depuis le `iLO`

Notre objectif

- Persistance de la compromission
- Résistance à une réinstallation de l'hôte
- Furtivité

Moyen

Backdoor du micrologiciel `iLO`

- Mécanismes de mise à jour :
  - Page dédiée sur l'interface web d'administration
  - Depuis l'hôte via un binaire
- Les mises à jour sont signées
- L'intégrité est vérifiée en deux points :
  - Dynamiquement, lors de la mise à jour, binaires vérifiés par le `iLO` en cours d'exécution
  - Au démarrage, **toutefois pas d'ancrage matériel**

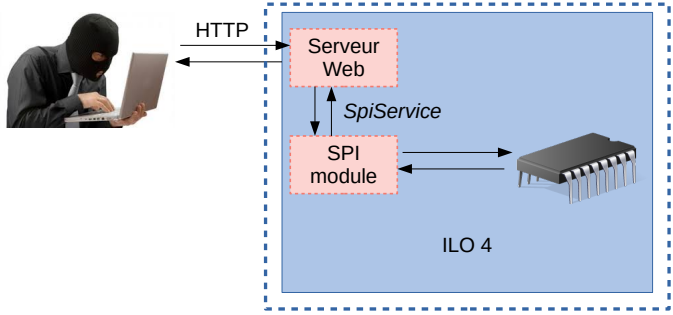
- Les modules exposent des services
- Ces services sont instanciables sous la forme d'objet

Le service `SPI`

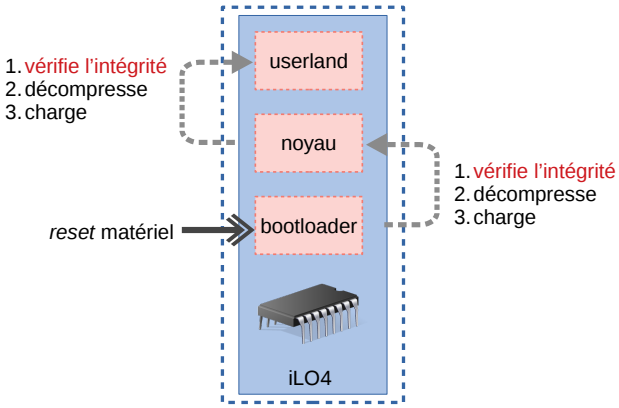
- "*SpiService*" dans le module `spi`
- Primitives de lecture/écriture directes dans la flash SPI

Attaque

- Appel au "*SpiService*" depuis un shellcode injecté dans le serveur web
- Réécriture directe du micrologiciel dans la flash
- Contournement de la vérification dynamique de la signature

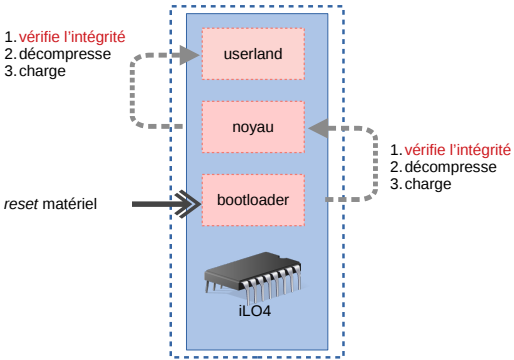


À ce stade, un micrologiciel compromis est écrit dans la flash


1. vérifie l'intégrité
2. décompresse
3. charge
userland
noyau
1. vérifie l'intégrité
2. décompresse
3. charge
reset matériel
bootloader
iLO4

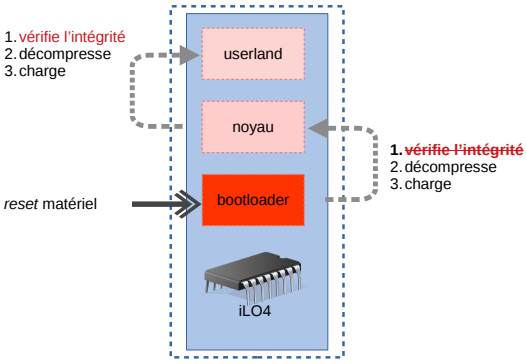
Méthodologie

- Extraction complète de la mise à jour



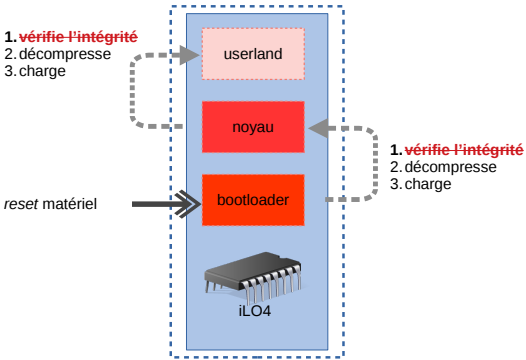
Méthodologie

- Extraction complète de la mise à jour
- Patch du bootloader



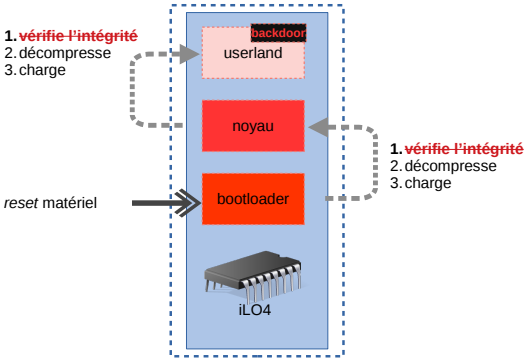
Méthodologie

- Extraction complète de la mise à jour
- Patch du bootloader
- Patch du noyau



Méthodologie

- Extraction complète de la mise à jour
- Patch du bootloader
- Patch du noyau
- Ajout d'une porte dérobée
- Reconstruction de la mise à jour
- Flash de la mise à jour



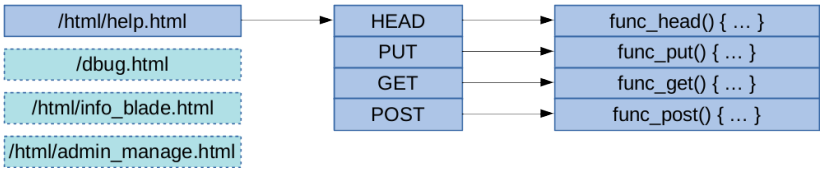
Le serveur web

- Très souvent exposé
- Dispose des primitives de communication réseau/`HTTP`
- Peut accéder à la mémoire de l'hôte via `DMA` (prouvé)
- Binaire volumineux

Le serveur web gère plusieurs pages, telles que :

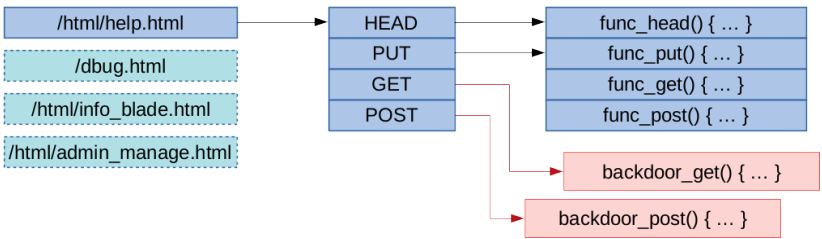
- */html/help.html*
- */dbug.html*
- */html/info_blade.html*
- */html/admin_manage.html*

Ces pages sont représentées en interne par des structures disposant de pointeurs de fonctions pour gérer les différentes méthodes `HTTP` (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`).

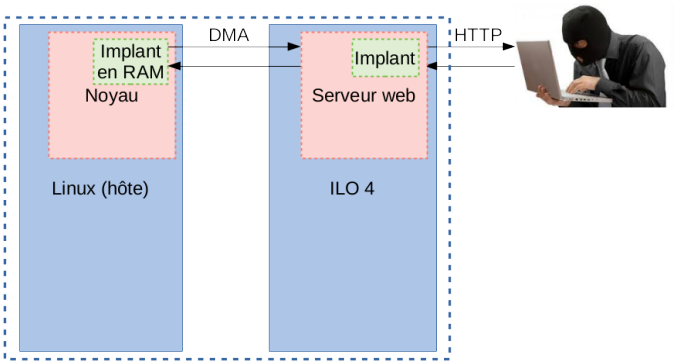


Comment insérer la porte dérobée ? (2)

- Insertion de code dans une zone non utilisée du binaire du serveur web
- Changement de pointeurs d'un gestionnaire de page pour pointer vers ce code (`GET` et `POST`)



On souhaite établir un canal bidirectionnel entre l'`iLO` et le Linux hôte, à travers le lien `DMA`.



Injection de code

- Réécriture d'un gestionnaire de requête `GET`
- Insertion du code dans une portion jugée inutile du binaire : le contenu d'un `PE` téléchargeable

Fonctionnalités

- Primitives de lecture et écriture dans la mémoire physique de l'hôte
- Ré-utilisation des fonctions du serveur web pour la gestion des requêtes

Objectifs

- Créer un nouveau thread noyau
- Allouer de la mémoire physique pour le canal de communication
- Récupérer les commandes et les exécuter
- Récupérer la sortie

Objectifs

- Créer un nouveau thread noyau
- Allouer de la mémoire physique pour le canal de communication
- Récupérer les commandes et les exécuter
- Récupérer la sortie

API noyau

- Création d'un nouveau thread : `kthread_create_on_node()` / `wake_up_process()`
- Allocation de mémoire physique : `kmalloc()` / `virt_to_phys()`
- Exécution de commandes : `call_usermodehelper()`
- Récupération de la sortie : redirection dans un fichier temporaire, puis `kernel_read_file_from_path()`

Simple structure dans une page mémoire physique partagée

- Tampon pour stocker une commande shell envoyée par l'iLO
- Tampon pour stocker la sortie de la commande, récupérée par l'iLO
- Booléens pour signaler la présence de données

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

Côté attaquant : client en Python

- Vérification de la présence des implants
- Installation et suppression de l'implant Linux
- Envoi de commandes arbitraires

Problème : les données écrites en mémoire sont parfois légèrement corrompues

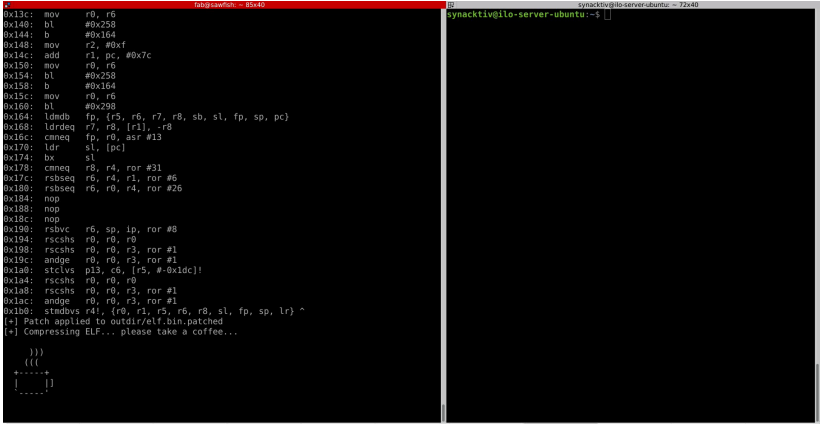
Visiblement lors de l'encodage de certains caractères...

```
if ( v13 == '%' )
{
  if ( v11 < 2 || sscanf(v5, "%d", &v19) != 1 || v19 > 0xFF )
    return 0;
  v12 = v19;
  v5 += 2;
  v11 -= 2;
  goto LABEL_21;
```

Nécessité de patcher le bug

```
# Patch query string decoding bug...
# "%d" => addrof("%02x")
PATCH5 = {"offset": 0x5D534, "size": 4, "prev_data": "25640000",
          "patch": "A8CE0400", "decode": "hex"}
PATCHES.append(PATCH5)
# ADR R1, "%d" => LDR R1, addrof("%02x")
PATCH6 = {"offset": 0x5D1A4, "size": 4, "prev_data": "E21F8FE2",
          "patch": "88139FE5", "decode": "hex"}
PATCHES.append(PATCH6)
```

Démo


fab@sawfish: ~ 85x40
0x13c: mov r0, r6
0x140: bl #0x258
0x144: b #0x164
0x148: mov r2, #0xf
0x14c: add r1, pc, #0x7c
0x150: mov r0, r6
0x154: bl #0x258
0x158: b #0x164
0x15c: mov r0, r6
0x160: bl #0x298
0x164: ldmdb fp, {r5, r6, r7, r8, sb, sl, fp, sp, pc}
0x168: ldrdeq r7, r8, [r1], -r8
0x16c: cmneq fp, r0, asr #13
0x170: ldr sl, [pc]
0x174: bx sl
0x178: cmneq r8, r4, ror #31
0x17c: rsbseq r6, r4, r1, ror #6
0x180: rsbseq r6, r0, r4, ror #26
0x184: nop
0x188: nop
0x18c: nop
0x190: rsbvc r6, sp, ip, ror #8
0x194: rscshs r0, r0, r0
0x198: rscshs r0, r0, r3, ror #1
0x19c: andge r0, r0, r3, ror #1
0x1a0: stclvs p13, c6, [r5, #-0x1dc]!
0x1a4: rscshs r0, r0, r0
0x1a8: rscshs r0, r0, r3, ror #1
0x1ac: andge r0, r0, r3, ror #1
0x1b0: stmdbvs r4!, {r0, r1, r5, r6, r8, sl, fp, sp, lr} ^
[+] Patch applied to outdir/elf.bin.patched
[+] Compressing ELF... please take a coffee...
synacktiv@ilo-server-ubuntu: ~ 72x40
synacktiv@ilo-server-ubuntu:~$

Comment détecter la compromission d'un iLO ?

- Récupération du micro-logiciel courant via un shellcode lisant la mémoire flash
- Comparaison à une liste d'images connues
- https://github.com/airbus-seclab/ilo4_toolbox
- Quid d'une porte dérobée modifiant à la volée les données lues ?

- Pas d'ancrage matériel[2], combiné au contournement d'une partie des mécanismes de vérification d'intégrité : **persistance possible et démontrée**
- Accès DMA à la mémoire de l'hôte transformé en canal de communication double-sens
- Les preuves de concept proposées requièrent l'exploitation d'une vulnérabilité et l'exécution de code arbitraire sur l'iLO
- Vulnérabilité rapportée à l'éditeur et corrigée en mai 2017, **patchez !**
- iLO4, un outil d'administration critique :
  - Désactivé si pas utilisé
  - Isolé dans un VLAN dédié

2.Problème supposé corrigé dans la dernière génération de serveur avec la version 5 d'iLO, disponible depuis mi-2017, *cf.* "*silicon root of trust*", https://support.hpe.com/hpsc/doc/public/display?docId=a00018320en_us

Merci pour votre attention

Questions ?

Pour nous contacter :
fabien [dot] perigaud [at] synacktiv [dot] com - @0xf4b
alexandre [dot] gazet [at] airbus [dot] com
snorky [at] insomnihack [dot] net - @_Sn0rkY