Analyse de firmwares de points d'accès, rétro-ingénierie et élévation de privilèges

Victorien Molle, Romain Bellan et Florent Fourcot
victorien.molle@wifirst.fr
romain.bellan@wifirst.fr
florent.fourcot@wifirst.fr

Wifirst
26 rue de Berri, 75008 Paris

Résumé. Grâce à une analyse et rétro-ingénierie des points d'accès Ruckus et Aerohive, il est possible de construire des firmwares modifiés acceptés par le matériel. Dans le même temps, ces équipements comportent également des portes dérobées permettant un accès à l'utilisateur privilégié (root) sur le système que nous détaillons.

1 Introduction

Ruckus et Aerohive sont des sociétés produisant du matériel réseau, en particulier pour les réseaux sans fil. Nous avons étudié le firmware [1] de leurs points d'accès (AP), et nous avons cherché à comprendre les mécanismes de validation d'un firmware afin de créer notre propre système personnalisé.

Ces recherches ont également permis de découvrir des portes dérobées permettant de s'échapper de l'interface en ligne de commande (CLI) restreinte pour accéder à une interface d'administration [2]. Celles-ci sont à chaque fois présentes dans le but de simplifier les opérations de débogage des fabricants mais posent des vraies questions de sécurité.

Une version enrichie avec des images et plus de détails de ce document est disponible sur le site du SSTIC avec la présentation.

2 Extraction et analyse des firmwares : méthodologie

Dans cette section, nous présentons les méthodes utilisées pour obtenir les premières informations sur les systèmes et permettre de se lancer dans des analyses et des modifications de firmware. Nous prenons pour exemple les points d'accès Ruckus, mais les mêmes méthodes ont été utilisées pour les points d'accès Aerohive.

1. Système d'exploitation interne.
2. Shell root.

2.1 Extraction du firmware

Afin d'extraire le contenu du firmware [3] Ruckus, nous avons utilisé un utilitaire nommé *binwalk* [4], capable de reconnaître et traiter des signatures connues dans un fichier (LZMA [5], SquashFS [6], pour ne citer qu'eux). Il permet ainsi de gagner du temps dans l'analyse en divisant en blocs un fichier firmware. *binwalk* divise ainsi le firmware en trois parties distinctes :

1. un en-tête (format inconnu pour *binwalk*) ;
2. un noyau Linux (au format LZMA) ;
3. un système de fichier (au format SquashFS).

Nous désignerons le système de fichier par le terme rootFS, système de fichier racine du point d'accès.

2.2 Reconstruction du firmware : modifications du rootFS et tentative naïve

Sur un système sans aucune protection, il est parfois possible de reconstruire un firmware après une simple modification du rootFS et de faire accepter la mise à jour par l'équipement par un simple téléversement. Nous avonc donc tenté cette approche, mais la protection du système nous en a empêché. La protection intègre donc une validation du firmware. Dans ce cas, nous somme contraints pour évoluer d'étudier les binaires responsables de cette protection.

2.3 Analyse statique de l'en-tête

Afin de pouvoir comprendre le format de l'en-tête, nous avons utilisé IDA [3] pour effectuer une analyse statique des binaires impliqués lors de la procédure de mise à niveau du firmware. Il est alors important de noter que l'architecture des points d'accès Ruckus repose sur un CPU de type MIPS [6].

Grâce à cette analyse, nous avons pu reconstituer la majorité des éléments constituant l'en-tête. Nous avons été aidés pour la compréhension de chaque élément par des informations accessibles via la ligne de commande du point d'accès qui est assez bavarde sur l'état du firmware. Cependant, il nous restait encore à comprendre le mode de calcul de l'intégrité de l'en-tête pour être capable de générer un firmware valide.

3. Le firmware a été récupéré depuis le site du contructeur.
4. `https://github.com/ReFirmLabs/binwalk`.
5. LZMA est un algorithme de compression sans perte.
6. SquashFS est un système de fichier compressé en lecture seule.

2.4 Analyse dynamique

Nous avions à notre disposition plusieurs versions du firmware des points d'accès Ruckus. Nous avons donc comparés les en-têtes et nous nous sommes rendu compte que seules trois informations étaient mises à jour :

— la taille de l'image (kernel + rootFS) ;
— le checksum de l'image ;
— le checksum de l'en-tête.

Si nous avons rapidement trouvé que le checksum de l'image était un simple condensat MD5 [4], nous n'avons pas immédiatement trouvé l'algorithme utilisé pour l'en-tête, nous le pensions donc non standard. Cet algorithme aurait pu être analysé entièrement en statique avec IDA, mais pour des raisons de gain de temps nous avons parallèlement utilisé un matériel basé sur la même architecture CPU pour faire du débogage distant et avoir un aperçu dynamique du fonctionnement du logiciel validant les firmwares sur le point d'accès.

Pour cela, nous avons copié l'exécutable en question sur notre routeur qui possédait un serveur GDB [7] d'où nous avons pu analyser le fonctionnement. Afin d'exécuter la partie du code qui nous intéressait, nous avons écrit un programme en C qui se chargeait de copier la partie du code en question dans un tampon rendu exécutable. Il nous suffisait ensuite d'attacher IDA sur le programme nouvellement créé et de tracer le code dynamiquement.

L'analyse a permis de déterminer que le checksum de l'en-tête était finalement calculé de façon simple. C'est une somme de contrôle additive, utilisant le même algorithme que la validation des en-têtes des paquets IP (algorithme défini dans la RFC 1071 [1]).

2.5 Détails du fonctionnement de la construction de l'en-tête

Comme vu précedemment, l'en-tête n'exige que la mise à jour de trois informations lorsque l'image est altérée. Nous avons écrit un programme qui se charge des actions suivantes :

1. récupérer la taille de l'image nouvellement créée ;
2. calculer le checksum MD5 de l'image ;
3. construire l'en-tête avec son checksum de valeur 0 ;
4. calculer le checksum de l'en-tête ;

7. GDB : The GNU Project Debugger, `https://www.gnu.org/software/gdb/`.

5. écrire le firmware en concaténant l'en-tête et l'image.

Nous sommes donc à présent capables de flasher un firmware modifié valide. Il n'y a pas de vérification cryptographique ni de protections plus avancées.

3 Analyse approfondie et élévation de privilèges

Durant l'analyse des binaires Ruckus, nous avons trouvé dans la bibliothèque *librkscli* des fonctionnalités non documentées :

— ruckus : easteregg faisant aboyer le point d'accès ;
— `!v54!` : fonction attrayante car faisant appel à une fonction nommée *cli_escape2shell*.

Une première analyse de cette fonction nous a montré qu'elle nécessitait une clef dont nous ne connaissions pas le format. Afin de simplifier les analyses, nous avons commencé par une analyse statique et poursuivi par une analyse dynamique.

3.1 Détails de `!v54!`

En analysant le code de la commande `!v54!`, nous nous sommes aperçus que cette commande lance un shell BusyBox [8] classique (`/bin/ash`). Nous avions entre les mains, la possibilité de passer root sur tous les points d'accès Ruckus à condition d'avoir un accès préalable à sa ligne de commande.

La clef demandée en entrée passe par la fonction *cli_escape2shell* qui ne modifie pas la clef et la transfère au programme *sesame*. Ce programme se charge d'effectuer des transformations sur cette clef, de construire un fichier (contenant divers paramètres) qui sera chiffré par *OMAC*, un binaire utilisé par Ruckus que nous décrivons en section 3.3.

3.2 Sésame, ouvre toi

Sesame est le programme qui se charge de dériver en plusieurs fois la clef passée lors de l'appel de la commande `!v54!`. Il se charge aussi de la vérification finale avant de lancer le shell root. La taille de la clef demandée est de trente-deux caractères qui sera par la suite traîtée par bloc de quatre caractères auquel sont appliquées diverses opérations. Le résultat des opérations est ensuite transféré à un autre binaire nommé *OMAC*.

8. BusyBox : The Swiss Army Knife of Embedded Linux, `https://busybox.net/about.html`.

3.3 Le binaire OMAC

OMAC est le binaire utilisé par Ruckus pour effectuer diverses opérations cryptographiques (AES-128-ECB, OMAC1-AES-128, MD5) sur les données générées par *sesame*. Ruckus a préféré utiliser le code de *WPA Supplicant* notamment pour la partie OMAC1 [2].

Après lecture des paramètres transmis par *sesame*, le programme *OMAC* agit sur les données en entrée afin de produire la clef de vérification finale de 16 octets.

3.4 Dernière ligne droite : génération d'une clef valide

Maintenant que nous avions tous les éléments en notre possession pour savoir comment la commande `!v54!` fonctionne, il ne nous manquait plus qu'à générer une clef valide pour chaque point d'accès. Le principe était simple : effectuer les mêmes opérations que *sesame* mais en sens inverse. Néanmoins, il fallait tout de même parvenir à retrouver la clef initiale à partir de la clef finale.

Générer la clef par force brute nécessite quelques adaptations car seuls les caractères affichables sont acceptés par la ligne de commande Ruckus, ce qui réduit grandement le nombre de possibilités. De plus, il faut que la force brute respecte l'algorithme de transformation de la clef par *sesame*.

3.5 Création du générateur de clef

Pour des raisons pratiques, nous avons entièrement recodé les fonctionnalités de *OMAC* en une bibliothèque nommée *libomac* utilisant exclusivement la *libtomcrypt* pour effectuer les diverses opérations cryptographiques [9]. Ceci nous a donc permis de nous passer de l'utilisation de *OMAC* uniquement présent sur les points d'accès et de compiler pour d'autres architectures que le MIPS.

En se basant sur cette bibliothèque, nous avons construit un exécutable prenant en entrée le numéro de série du point d'accès et générant une clef valide. La présentation entre plus dans les détails de la solution choisie.

4 Élévation de privilèges chez Aerohive

Ces recherches sur les points d'accès Aerohive ont conduit à la publication d'un bulletin de sécurité [5] rédigé par le *SIRT Aerohive*.

9. Une autre solution est d'utiliser le binaire récupérable sur les points d'accès. Cependant, cela implique de faire tourner un binaire en architecture MIPS, alors que notre version est compilable facilement pour toutes les architectures.

Tout comme Ruckus, Aerohive possède sa propre CLI. Après avoir effectué des recherches dans les différentes bibliothèques mises à disposition dans le système de fichiers, nous avons pu trouver une commande cachée, non enregistrée auprès de l'aide intégrée à la CLI, se nommant *_shell*.

Cette commande une fois exécutée, demande l'entrée d'un mot de passe. Après quelques essais ratés (ce dont nous ne doutions pas), nous avons dû nous résigner à tracer le code qui gère la commande *_shell*. Lors du traçage, nous sommes arrivés à une fonction exportée depuis une des bibliothèques d'Aerohive se nommant *ah_gen_password*. Dans le code initial, cette fonction est appelée et retourne une chaîne de caratères qui est ensuite comparée avec ce que l'utilisateur a entré. Si les deux chaînes sont identiques, alors un terminal */bin/sh* est appelé.

A l'appel de la fonction *ah_gen_password*, deux arguments sont passés en paramètres :

1. une chaîne de caractères lue depuis */etc/.ahsecret* (arg0) ;
2. le numéro de série du point d'accès (arg1).

On peut donc en déduire que la fonction prend probablement en entrée le secret et un serial et renvoie une chaîne.

La fonction effectue principalement des opérations de type hachage MD5 sur les chaînes de caractères en entrée et de la substitution d'octets sur les résultats avec une table statique définie dans la bibliothèque d'Aerohive.

Au final, le résultat est simplement un condensat MD5 retransformé en caractères imprimables (en hexadécimal).

Contrairement à ce que l'on a pu voir sur le cas Ruckus, Aerohive a choisi d'intégrer directement le générateur de clef dans les points d'accès. Une fois le mécanisme de la fonction *ah_gen_password* compris, il nous suffisait d'écrire un simple programme chargeant la bibliothèque qui contient cette fonction, et de l'appeler en lui passant les bons paramètres. À la fin de l'exécution, il nous retourne le mot de passe correct que nous pouvons utiliser pour exploiter l'élévation de privilèges. Néanmoins, l'architecture utilisée sur ce point d'accès étant un processeur ARM [10], nous avons opté pour une rétro-ingénierie complète de la fonction et des sous-fonctions appelées afin de pouvoir profiter du générateur de mot de passe sur une architecture processeur plus habituelle (de la même façon que pour la *libomac*).

10. Et également en raison de la curiosité des méthodes utilisées par les fabricants

5 Mise en place d'un firmware personnalisé

Le document numérique explique plus en détails le format du firmware utilisé par Aerohive, notamment l'utilisation d'une signature RSA empêchant l'altération du contenu. À présent que nous possédons les droits superutilisateur suite à l'élévation de privilèges précédemment expliquée, nous allons pouvoir procéder à la mise en place d'un firmware personnalisé en contournant la vérification de la signature RSA.

Plusieurs approches sont possibles, en voici quelques-unes :

1. Écrire directement sur la mémoire NAND via les binaires permettant de manipuler les partitions sur le point d'accès ;
2. Patcher la bibliothèque *libah_img.so* à l'exécution pour ignorer la vérification de la signature RSA ;
3. Exécuter le processus qui utilise une bibliothèque *libah_img.so* modifiée.

Suite à des problèmes de compilation croisée, nous n'avons pas pu réussir la méthode qui consiste à patcher la bibiliothèque *libah_img.so* en revanche, nous avons modifié la bibliothèque en amont et utilisé la méthode *LD_PRELOAD* pour charger le processus qui gère la mise à jour du firmware avec notre bibliothèque modifiée. La modification effectuée consiste simplement à continuer l'exécution de la vérification du firmware à flasher sans se préoccuper de la signature RSA. La fonction appelée afin d'effectuer la vérification retourne *0* quand la signature est valide, il nous suffisait donc de faire en sorte que le programme continue son exécution quel que soit le code de retour.

6 Conséquences post-exploitation

La finalité de nos recherches nous a menés à obtenir un accès superutilisateur sur les points d'accès Ruckus et Aerohive. Les conséquences d'un tel accès peuvent être dévastatrices. En effet, un attaquant ayant la main sur un point d'accès pourrait se positionner en man-in-the-middle et ainsi intercepter tout le trafic transitant par ce point d'accès. Au même titre, ces points d'accès requièrent une connexion permanente à Internet afin d'être supervisés via leur manager (SmartZone/HiveManager), on pourrait donc imaginer qu'un attaquant puisse installer un serveur VPN afin de récupérer la main à distance sur ce point d'accès. De plus, avoir la main sur le système de fichiers et le mécanisme de mise à jour permet d'envisager une persistance de l'attaque malgré un correctif ultérieur du fabricant.

7 Conclusion

Grâce au travail effectué sur l'édition du firmware, nous avons la possibilité de construire des firmwares personnalisés et de les faire accepter par différents matériels.

Nous pouvons aussi obtenir les droits d'administration sur les points d'accès de deux fabricants sans avoir besoin de modifier leurs firmwares, à condition d'avoir un accès utilisateur à la ligne de commande.

La génération de clef fonctionne sur l'ensemble des points d'accès Ruckus utilisant leur BSP [11] version 54. Notre générateur de clef est capable de générer plusieurs clefs valides pour un même point d'accès, ce qui prouve qu'il n'y a pas une clef unique par point d'accès. Il est également à noter qu'un appel à la commande `!v54!` ne génère pas d'alerte ni la moindre historisation dans les gestionnaires de points d'accès de Ruckus (appelés SmartZone).

Concernant les points d'accès Aerohive, nous avons aussi la possibilité de générer une clef valide pour chaque point d'accès, néanmoins, seule une clef par point d'accès est valable. De la même façon, l'appel à la commande *_shell* ne génère pas d'alerte ou d'historisation dans les gestionnaires de points d'accès Aerohive (appelés HiveManager).

Malgré le gain acquis en ayant la possibilité de personnaliser le rootFS, il nous reste encore à pouvoir générer notre propre noyau et ainsi pouvoir complètement contrôler le point d'accès.

Références

1. R.T. Braden, D.A. Borman, and C. Partridge. Computing the Internet checksum. RFC 1071, September 1988. Updated by RFC 1141.
2. Blaise Gassend. Wpa supplicant - omac1-aes-128. `http://docs.ros.org/diamondback/api/wpa_supplicant/html/aes-omac1_8c.html`, 2013.
3. Ilfak Guilfanov. Ida est un désassembleur commercial très utilisé en rétro-ingénierie. `https://www.hex-rays.com/products/ida/index.shtml`, 1987.
4. R. Rivest. The md5 message-digest algorithm, April 1992. RFC1321.
5. Aerohive SIRT. Product security advisory authenticated user privilege escalation. `https://www.aerohive.com/support/security-center/product-security-advisory-authenticated-user-privilege-escalation-jul-31-2018/`, July 2018.
6. MIPS Technologies. L'architecture mips est une architecture de processeur de type risc. `https://www.mips.com/`, 1985.

11. Board Support Package.