Recherche de vulnérabilités dans les drivers 802.11 par techniques de fuzzing

Laurent Butti and Julien Tinnès

France Télécom R&D
Laboratoire Sécurité des Services et Réseaux
38-40 Rue du Général Leclerc
92794 Issy-les-Moulineaux Cedex 9 - France
firstname.lastname@orange-ftgroup.com

Résumé Les réseaux locaux radioélectriques sans-fil 802.11 sont sous les feux des projecteurs depuis de nombreuses années de par leurs multiples failles protocolaires [1]. L'année 2006 a été l'avènement des failles dans les drivers 802.11 côté client. Nous présentons dans cet article la méthodologie adoptée en recherche de vulnérabilités dans les drivers 802.11 : le « fuzzing ». Nous exposons alors la conception et le développement d'un fuzzer 802.11 qui nous a permis de découvrir plusieurs failles critiques dans des drivers 802.11. Ces failles ouvrent potentiellement la voie à l'exécution de code arbitraire en mode noyau à distance via la voie radioélectrique.

1 Introduction

Un fait marquant de l'année 2006 a été la présentation de Johnny Cache et David Maynor intitulée « Device Drivers » [2] à la conférence BlackHat USA 2006 [3]. Elle abordait les problématiques actuelles de la recherche de vulnérabilités qui devient de plus en plus complexe de par les efforts en développement réalisés dans les systèmes d'exploitation, services et applications les plus usités : il est aujourd'hui beaucoup plus difficile de trouver une nouvelle vulnérabilité système[1] et de l'exploiter qu'il y a quelques années... Les auteurs ont alors présenté un nouvel axe de recherche : attaquer les drivers[2] de périphériques.

Durant leur présentation, ils ont diffusé une vidéo [4] relatant la compromission d'un MacBook via l'exploitation à distance d'une faille dans un driver 802.11. Cette vidéo a provoqué un vent de panique car elle représentait la première affirmation publique de la faisabilité de telles attaques et en particulier sur un système d'exploitation encore peu visé. Cependant, cette vidéo a laissé quelque peu pantois, car peu d'informations avaient transpiré et des doutes sur la véracité de l'attaque étaient alors tout à fait légitimes. Le flou autour de cette vidéo[3] a été diversement apprécié et est encore aujourd'hui difficile à analyser tant les discussions sur le sujet ont été animées [5,6]. Des éléments de réponse sont disponibles depuis peu [7,8].

Toujours est-il que cette présentation a surtout eu le mérite d'attirer l'attention sur des failles potentiellement critiques de par leur accessibilité via la voie radioélectrique. Un attaquant serait alors capable d'exécuter du code arbitraire avec des privilèges ring0[4] et qui plus est à distance ! Il

[1] on ne parle pas ici des multiples vulnérabilités orientées Web (PHP, SQL injection, XSS...) qui sont légion.

[2] dans cet article nous utiliserons la dénomination anglaise « driver » au lieu de « pilote ».

[3] mais aussi autour de la gestion de la publication de la vulnérabilité.

[4] niveau de privilège le plus élevé, en particulier pour l'accès direct aux périphériques, typiquement attribué au noyau [10].

suffit alors de remarquer que la majorité des ordinateurs portables sont équipés de chipsets 802.11 et que la plupart d'entre eux l'utilisent[5], le risque est donc extrêmement critique. Bien entendu, les mécanismes de protection classiques tels que pare-feu, anti-virus et prévention d'intrusion système ne peuvent rien faire car l'exécution de code arbitraire est réalisée en mode noyau ! Serait-il alors possible à distance, de compromettre des systèmes d'exploitation via des failles drivers 802.11 ? ! ?

2 Problématique

Un driver est généralement du code développé en C/C++ et les erreurs classiques de programmation dans ces langages permettent (entre autres) des attaques de type débordement de tampon. Cette constatation permet d'affirmer que les drivers n'ont pas de raison spécifique d'être épargnés.

Par conséquent, la première difficulté est de trouver ces erreurs qui peuvent se révéler être des failles de sécurité exploitables. Il est en effet important de se rappeler que de nombreuses conditions sont nécessaires pour qu'une erreur de programmation soit exploitable.

En sécurité, la complexité est souvent source de tracas. Dans le cas que l'on étudie dans cet article, la complexité de la norme 802.11 et de ses nombreuses extensions (802.11i pour la sécurité, 802.11e pour la qualité de service et toutes celles à venir...) est une source potentielle d'erreurs de programmation que ce soit dans les drivers et firmwares des cartes clientes ou dans les firmwares des points d'accès. La découverte de ces erreurs est par contre souvent difficile que ce soit dans des drivers dont le code source est fermé[6] ou ouvert[7].

De manière intrinsèque les drivers 802.11 sont donc susceptibles de présenter des erreurs de programmation, mais pas forcément triviales à repérer. Il est donc nécessaire d'adopter une approche meilleur rapport « qualité-prix » dans la recherche de vulnérabilités.

3 Le fuzzing

3.1 Définitions

Ce terme difficilement traduisible[8] en français est tout aussi difficilement définissable. Parmi les nombreuses définitions disponibles sur Internet, les deux suivantes méritent notre attention :

- « Le fuzzing est une technique pour tester des logiciels. L'idée est d'injecter des données aléatoires dans les entrées d'un programme. Si le programme échoue (par exemple, en crashant ou en générant une erreur), alors il y a des défauts à corriger. » (source : Wikipedia).
- « Fuzz Testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed or semi malformed data injection in a automated fashion. » (source : Open Web Application Security Project (OWASP)).

Le concensus dans la définition du fuzzing semble reposer sur le principe suivant : « technique de tests logiciels pour découvrir des erreurs de programmation ». La question ouverte étant : quelles sont les différences entre le fuzzing et les techniques classiques de tests logiciels [11].

[5] ou l'ont activé par défaut.
[6] dans ce cas, cela nécessiterait du « reverse engineering » à la fois long et coûteux lorsque l'on n'a pas d'indice où chercher.
[7] dans ce cas, cela nécessiterait un audit de code approfondi.
[8] tellement difficile que nous utiliserons sans retenue le verbe « fuzzer » !

En bref, il est très difficile de définir le fuzzing tant ce terme semble être né sous l'effet (pervers) de la mode ! L'industrie de la sécurité informatique est en effet friande de ces termes...

Dans la suite de cet article, nous considererons le fuzzing comme une technique de recherche d'erreurs de programmation quels que soient les tests réalisés (aléatoires, en fonction du protocole à tester...).

3.2 Que doit-on attendre du fuzzing ?

Le principal intérêt du fuzzing réside dans un rapport favorable entre facilité[9] de création de l'outil (le fuzzer) et les bénéfices escomptés (vulnérabilités découvertes). En effet, grâce à des techniques basiques de fuzzing il est envisageable de découvrir des erreurs de programmation grossières et ce très rapidement.

Par contre, il est difficile de réaliser des tests complètement exhaustifs via du fuzzing car l'espace des tests possibles est généralement trop important par rapport aux délais impartis pour les tests. Il ne faut pas voir le fuzzing comme une technique élaborée d'évaluation de la sécurité mais plutôt comme une brique fatalement incomplète de l'évaluation de logiciels.

3.3 Quelques exemples et applications concrêtes

Le fuzzing est très à la mode depuis 2006, en témoignent les « Month of Browser Bugs (MOBB) », « Month of Kernel Bugs (MOKB) », « Month of Apple Bugs » et « Month of PHP Bugs » [12,13,14,15] qui se sont reposés grandement sur ces techniques pour pouvoir publier un nouveau bug par jour.

L'exemple du fuzzer de systèmes de fichier *fsfuzzer* [16] publié lors du MOKB est très significatif. Son auteur a pu découvrir de très nombreuses failles grâce à des principes assez rudimentaires[10] à la vue du code source de l'outil... En effet, souvent un fuzzer se révèle être une option très efficace dans la recherche des principales erreurs de programmation. Pour la recherche des erreurs subtiles, cela est malheureusement beaucoup moins pertinent.

4 Quelques rappels sur 802.11

4.1 Machine à états 802.11

La machine à états 802.11 en figure 1 comporte trois états :
- état 1 : état initial, non authentifié, non associé ;
- état 2 : authentifié[11], non associé ;
- état 3 : authentifié, associé.

Lorsqu'un client 802.11 communique avec un point d'accès 802.11, les phases permettant le passage d'un état à l'autre sont les authentifications, associations, déassociations et déauthentifications. Afin de pouvoir communiquer correctement, les machines à états des deux entités doivent être synchronisées, i.e. dans le même état, si cela n'était pas le cas, des procédures de déauthentification ou de déassociation permettent alors aux deux entités de se resynchroniser.

[9] si l'outil est peu évolué.

[10] pas de tests spécifiques au système de fichier visé, uniquement changement aléatoire d'octets dans les entêtes.

[11] au sens 802.11, i.e. authentification « open » ou « shared key », ne pas confondre avec les mécanismes de sécurité tels que 802.11i.

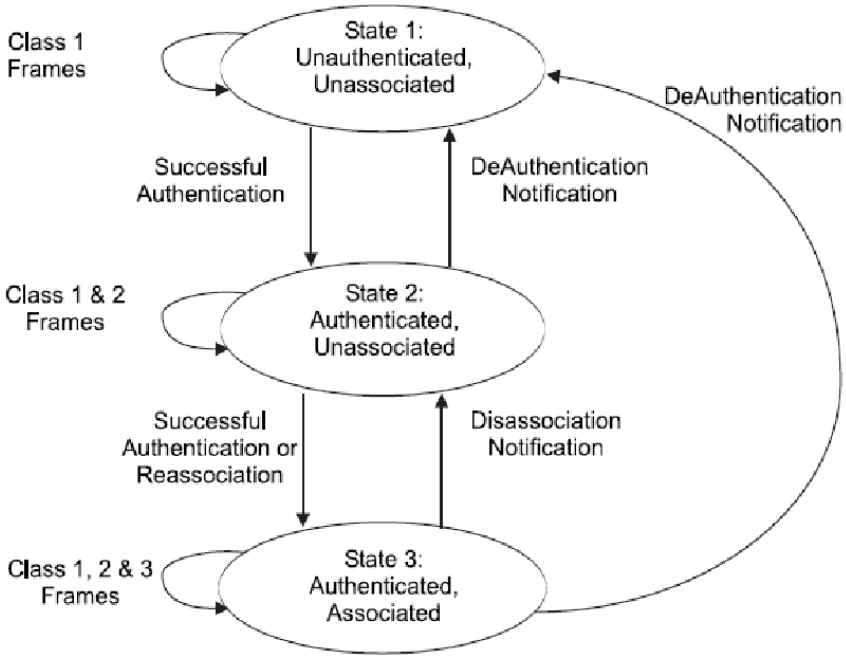

Fig. 1: Machine à états 802.11

Du point de vue du client, l'état 1 est l'état dans lequel le client recherche les points d'accès, l'état 2 est l'état dans lequel le client est authentifié auprès du point d'accès et l'état 3 est l'état dans lequel le client est associé au point d'accès. A la suite d'une association réussie, le client peut communiquer[12] via le point d'accès auquel il est associé (et vice-versa). Les transitions entre ces états sont réalisées grâce à des trames de type « management » que l'on peut apparenter à des trames de signalisation.

Le standard 802.11 définit ces états primordiaux qui doivent être implantés dans les driver et firmware 802.11 mais bien entendu de nombreux sous-états existent de manière implicite pour que l'on puisse réellement avoir un mécanisme 802.11 fonctionnel. Par exemple, un point d'accès n'acceptera que les demandes d'association vers des noms de réseau qui auront été préalablement configurés. En terme d'implantation logicielle, de nombreux autres sous-états seront nécessaires... Ceci rend d'autant plus difficile les techniques de fuzzing car le but du jeu est de tester le maximum de chemins possibles[13] dans l'exécution du code d'un driver ou d'un firmware.

4.2 Les différents types de scan 802.11

La norme 802.11 définit deux méthodes pour la découverte les points d'accès par les clients :

– durant le scan actif, le client émet des trames « probe request » en broadcast sur chacun des canaux 802.11 afin d'obliger les points d'accès à répondre par des trames « probe response »,

[12] i.e. envoyer et recevoir des trames de données.
[13] « code paths » en anglais.

il écoute alors sur chacun de ces canaux les balises et réponses des points d'accès (les trames « beacon » et « probe response ») ;

– durant le scan passif, le client écoute sur chacun des canaux les trames « beacon » émises par les points d'accès[14].

Grâce à ces deux méthodes, le client est capable de construire une liste des points d'accès environnants. Le service Wireless Zero Configuration sous Windows et l'utilitaire *iwlist* des *wireless-tools* [17] sous Linux reposent sur ces principes. Ces deux méthodes sont déterminantes dans le fuzzing 802.11, car la première étape dans le fuzzing des drivers 802.11 clients sera alors de créer des trames probe reponse ou beacon spécifiques qui déclencheront (éventuellement) un dysfonctionnement dans le driver 802.11. A noter aussi que certaines implantations logicielles des drivers et firmwares peuvent être légèrement différentes à propos des méthodes de découverte des points d'accès, ce sera point qui sera abordé plus tard dans cet article car il a eu un impact sur la conception de notre fuzzer 802.11.

5 Conception d'un fuzzer 802.11

5.1 Que peut-on fuzzer ?

Les trois états de la machine à états 802.11 sont susceptibles d'être fuzzés. Cependant il est beaucoup plus facile de fuzzer l'état 1 plutôt que les états supérieurs. En effet, pour fuzzer l'état 3 (typiquement via les « association responses ») il est nécessaire d'arriver à passer sans encombre les états 1 et 2. Or, ceci n'est pas facile si l'on utilise une carte en mode monitor[15] (et non en mode station) car il serait alors obligatoire d'émuler les différentes étapes en mode monitor, mais aussi et surtout avoir la lourde tâche de renvoyer les acquittements des trames 802.11 dans des délais extrèmement courts[16].

Dans cet article nous présenterons essentiellement nos travaux sur le fuzzing de l'état 1 côté client comme présenté dans la figure 2. Si des failles exploitables sont découvertes, alors cela implique qu'elles sont utilisables lorsque le client est en mode recherche de points d'accès. L'exploitation d'une faille driver 802.11 ne nécessite donc pas que le client 802.11 soit associé à un quelconque point d'accès, mais tout simplement d'écouter des trames beacon ou probe response préalablement formatées par l'attaquant.

Ceci rend ces attaques très dangereuses par rapport à des attaques classiques qui consistaient soit à forcer des clients à s'associer à des points d'accès malveillants [18] ou à injecter des données dans les communications des couches supérieures [19,20]. Il faut bien comprendre que dans notre cas, on attaque le driver (ou le firmware) 802.11 et non les couches supérieures.

Par ailleurs, il ne faut pas oublier que les technologies 802.11 sont de plus en plus omniprésentes : ordinateur portable, téléphone cellulaire, assistant électronique personnel, imprimante, appareil photo... Tant de portes potentiellement ouvertes et si difficilement maîtrisables...

[14] ces trames de balise sont émises régulièrement par les points d'accès (dans la plupart des cas toutes les 100 millisecondes) et comprennent toute les informations nécessaires au client pour commencer une procédure d'association auprès du point d'accès (nom de réseau, canal, capacités cryptographiques...).

[15] mode spécifique qui permet de remonter et d'injecter des paquets en RAW.

[16] en pratique de l'ordre de quelques dizaines de millisecondes.

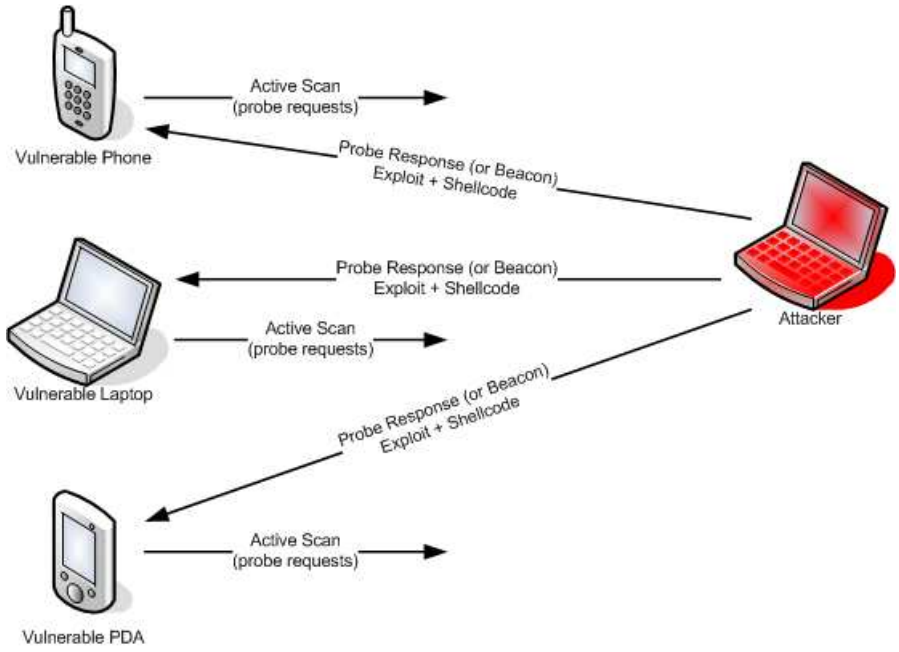

FIG. 2: Schéma des attaques de clients 802.11

5.2 Principales problématiques à gérer

Cette partie présente quelques problématiques auxquelles nous nous sommes confrontés lors de la réalisation d'une architecture de fuzzing de clients 802.11.

Fuzzing des états de la machine à états 802.11. Fuzzer l'association d'un client à un point d'accès est plus complexe que fuzzer la recherche des points d'accès par un client. En effet, des contraintes fortes au niveau MAC, telles que les procédures d'acquittement des trames reçues dans des délais très courts[17], rendent très difficile et très perfectible l'utilisation de l'injection RAW[18] pour réaliser ces tests. En effet, comme il faut réussir le passage de l'état 1 à l'état 2, il faudrait émuler les réponses d'un point d'accès en renvoyant les trames d'authentification et d'association en réponses à celles du client, mais aussi de renvoyer les trames d'acquittement dans des délais très courts. Même si cela reste réalisable en C, l'idéal est plutôt de réaliser des modifications au niveau d'un driver point d'accès 802.11 qui alors permettrait de fuzzer le driver client 802.11. Nous avions rencontré ce type de problématique lors du développement de l'outil Raw Glue AP [21] qui est une preuve de concept de la faisabilité d'émulation d'un point d'accès complètement en mode monitor afin de capturer les clients 802.11 en émulant un point d'accès virtuel[19].

L'approche la plus rentable consiste donc à fuzzer la recherche de points d'accès par les clients 802.11. Ce choix permet alors de développer tout un ensemble de tests qui seront lancés via une

[17] en pratique de l'ordre de quelques dizaines de millisecondes.
[18] en mode monitor.
[19] car en mode monitor.

carte 802.11 en mode monitor ce qui simplifie grandement la technique car ne nécessite pas de modifications au niveau du driver réalisant l'injection de paquets.

Fuzzing des firmwares. Cet article n'a abordé que les drivers 802.11. Cependant, selon les chipsets 802.11 certaines opérations sont réalisées dans les firmwares. C'est par exemple le cas des cartes Prism2.5 et Prism54 qui réalisent la recherche des points d'accès dans le firmware. Typiquement, le traitement des probe request et probe response est réalisé dans le firmware du chipset et non dans le driver. Par conséquent, découvrir un dysfonctionnement à ce niveau revient à découvrir un dysfonctionnement dans le firmware [24].

La principale difficulté ici est alors de détecter ce dysfonctionnement lorsque l'on n'a pas beaucoup d'informations remontant du firmware (pas de log, pas de débogage[20]...). La deuxième difficulté serait d'exploiter aussi ce firmware vulnérable ! En conséquence, on est surtout en droit d'attendre des attaques de type déni de service sur cette catégorie de dysfonctionnements.

Temps nécessaire versus espace des tests possibles. Conceptuellement, l'espace des tests possibles à réaliser est gigantesque. Si le fuzzing peut consister à simplement remplacer certains octets par des valeurs aléatoires alors pour chaque octet d'un paquet 802.11 on aura 256 possibilités... Pour n octets, nous avons donc 2^{8n}, ce qui rend le fuzzing complètement aléatoire difficilement réalisable[21] en pratique. Nous avons donc préféré une approche de fuzzing de protocole afin de ne tester que des champs spécifiques avec des valeurs préalablement choisies. Plus le fuzzer (ou plutôt le concepteur de ce dernier) a une bonne connaissance du protocole à tester, meilleures seront les chances d'avoir des résultats intéressants. Il suffira alors de faire varier de manière la plus « intelligente » possible[22] les champs qui auront été jugés comme étant intéressants à fuzzer. Le concepteur du fuzzer doit penser en terme d'erreurs possibles d'implantations logicielles[23] afin de spécifier les tests les plus pertinents.

A contrario, le fuzzer 802.11 publié à la conférence BlackHat US 2006 [22] a par contre eu une approche différente car à la vue du code source, il consistait à réaliser des tests de manière aléatoire.

Les deux approches semblent complémentaires, mais on a privilégié la première pour des raisons d'optimisation des temps de passage des tests.

Forcer le mode scan. Cela peut paraître évident à première vue, mais il est nécessaire que la carte soit en permanence en mode scan pour être certain que les trames envoyées par le fuzzer seront attrapées puis interprétées par le driver 802.11. Généralement, sans forcer le mode scan, le driver réalise de lui-même des scans réguliers selon un certain intervalle pré-défini. Tout cela est dépendant de l'implantation logicielle de ce dernier.

Pour ce faire, sous Windows, nous avons utilisé le célèbre outil de scanning NetStumbler [23], et sous Linux, nous avons utilisé l'outil *iwlist* qui réalise des appels systèmes `SIOCIWSCAN` et `SIOGIWSCAN` lorsqu'il est exécuté en utilisateur privilégié (uniquement `SIOGIWSCAN` en utilisateur non privilégié, dans ce cas, pour avoir des résultats on se repose sur les fonctionnalités de scan en arrière plan[24] des drivers).

[20] en particulier sur les cartes clientes.
[21] et donc probablement peu efficace.
[22] par exemple en bordure des valeurs extrèmes.
[23] s'il avait eu la lourde tâche de développer un driver 802.11.
[24] souvent appelés *bgscan* dans les paramètres de configuration.

Assurance de la bonne réception des trames. Lors du scan, la carte 802.11 écoute pendant un délai plus ou moins court sur chacun des canaux 802.11. Si les paquets de test sont envoyés sur le canal 1 alors que la carte écoute sur le canal 11, alors ce test ne sera pas valable car les paquets ne seront pas interprétés par le driver 802.11. Il est généralement plus satisfaisant d'envoyer de nombreuses fois le même test unitaire pendant une durée de quelques secondes sur un seul canal qui sera dans tous les cas écouté par la carte en mode scan. Bien entendu, il faut s'assurer que la carte reste en mode scan durant toute la durée des tests.

Détection des problèmes (bugs). Lors des tests de fuzzing, des dysfonctionnements au niveau du driver 802.11 peuvent apparaître[25]. Dans le cas des drivers 802.11 sous Windows que nous avons pu tester, des écrans bleus de la mort (BSOD) nous ont permis rapidement de détecter un crash système. Dans ce cas précis de crash, plusieurs possiblités de détection peuvent s'offrir à nous. La plus simple consiste à vérifier que la machine fuzzée répond toujours sur une interface Ethernet, si ce n'est pas le cas, alors il est facile d'identifier le dernier test qui aura été fatal. Sous Linux, c'est différent, car la trace d'un bug dans un driver 802.11 se retrouvera dans la journalisation noyau (BUG, oops...). Il est alors possible de réaliser une recherche de ces chaînes de caractères dans la journalisation noyau et d'identifier alors le dernier test qui aura été fatal.

D'autres possibilités génériques sont envisageables telles que l'écoute des probe requests. En effet, si la carte n'émet soudainement plus de trames de type probe request alors qu'elle est toujours en mode scan actif, alors il est fort probable qu'un dysfonctionnement a été détecté. L'avantage de cette dernière approche est son aspect générique (toutes plate-formes). Le désavantage est qu'il faut être précis dans les délais d'attente des trames de type probe requests afin d'éviter les faux positifs.

A noter, que bien entendu, nous parlons ici d'erreurs critiques. Il est possible que des tests de fuzzing entraînent des bugs dans les drivers 802.11 qui ne soient pas détectables en pratique car aucune information n'aura transpiré (pas de crash système ou pas de journalisation noyau). Ceci est malheureusement extrèmement difficile à identifier. L'exemple le plus typique sont les bugs de type « fuite d'informations » où il est possible de faire lire au driver des zones mémoires contigües au paquet 802.11 stocké en mémoire du processus, cependant, comme ces zones mémoires sont lisibles et généralement accessibles, cela n'entraîne pas de dysfonctionnement critique.

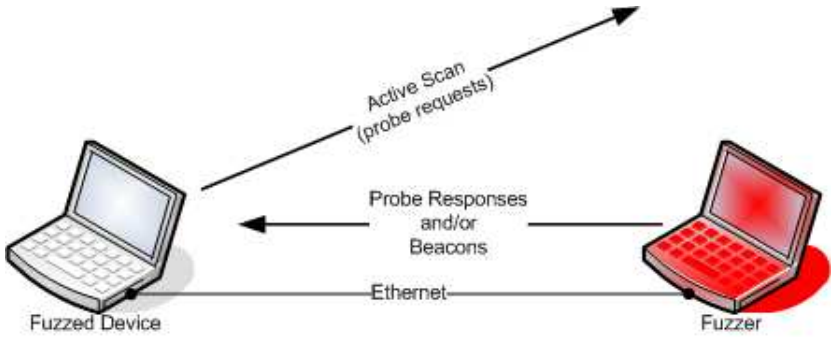

FIG. 3: Architecture de fuzzing 802.11

[25] en fait c'est même souhaité !

Dans l'architecture en figure 3 que l'on a déployée, nous utilisons une connectivité Ethernet pour vérifier l'apparition de dysfonctionnements afin de stopper l'application de fuzzing sur le test pertinent.

5.3 Développement

Choix du matériel. Le chipset Atheros et son driver Open Source madwifi [25] sous Linux sont réputés pour être très flexibles[26] au niveau de la création des trames 802.11. En effet, le fuzzing impose le changement de champs spécifiques des trames 802.11 qui peuvent dans certains cas être gérés directement par le firmware. Si tel était le cas, alors les tests seraient faussés du fait de la non maîtrise de certains champs 802.11.

Les limitations possibles à cause du firmware du chipset sont généralement au niveau de :
- la valeur du numéro de séquence des trames,
- la valeur du BSS Timestamp dans les trames beacon et probe response,
- la valeur du duration id des trames,
- la capacité à envoyer des trames fragmentées,
- la capacité à envoyer des trames de contrôle.

Le chipset Atheros supporte toutes les fonctionnalités précédentes sauf l'usurpation du numéro de séquence des trames qui n'est pas supporté en standard dans le driver madwifi. Il est possible de patcher le driver pour supporter cette fonctionnalité [26], mais elle impose alors d'avoir le champ « retry » positionné à 1 pour toutes les trames injectées, ce qui est génant si l'attaquant veut se cacher de logiciels de détection d'intrusion 802.11 évolués.

Choix du langage de développement. Même s'il est parfaitement réalisable de développer un fuzzer en C/C++ [22], il apparaît plus judicieux d'utiliser un langage de haut niveau pour la création des paquets 802.11. Nous avons choisi le langage Python pour sa simplicité d'usage.

La contrainte initiale concernant le langage de développement a été relative à la rapidité d'exécution de l'outil. En effet, dans certaines approches il est nécessaire d'attendre les probe requests pour répondre avec des probe responses. Cette méthode impose un traitement rapide du paquet reçu puis de l'émission du paquet réponse dans des délais très courts de par le fait que la carte 802.11 qui scanne les réseaux 802.11 n'attend que très peu de temps les réponses sur le canal qui aura été sondé.

Cette approche au début de la conception de l'outil a finalement été abandonnée car il n'était pas possible de répondre suffisamment rapidement à tous les coups ce qui entrainait alors des problèmes de fiabilité[27] des tests. Nous avons donc choisi une méthode à la fois brutale et efficace : saturer la voie radioélectriques de trames beacon et probe response pour chacun des tests de fuzzing réalisés. Par conséquent, nous n'avons plus la contrainte de répondre rapidement, il suffit simplement de connaître le temps nécessaire pour un chipset de parcourir tous les canaux pour être sûrs que ces trames seront analysées par le driver. A noter que cette méthode fonctionne car aucun mécanisme de type « cookie » n'est utilisé dans les trames probe request et probe response pour ajouter une notion de session à la procédure de recherche de points d'accès. Enfin, certains drivers n'acceptent les beacons que si des probe responses sont aussi envoyées (ce qui est un comportement réellement

[26] car le firmware est très minimaliste et les opérations MAC 802.11 sont réalisées au niveau du driver.

[27] i.e. diminuer au maximum les faux négatifs et augmenter au maximum la reproductibilité des dysfonctionnements découverts.

dépendant de l'implantation logicielle), par conséquent, cette approche un peu brutale révèle ici toute son efficacité ! Ces problématiques ont aussi été observées dans l'excellent papier de Johnny Cache, HD Moore et skape [27].

Choix d'outils existants ou développement de zéro. La première idée était de se reposer sur *Scapy* [28] et de fonctions spécifiques de type "receive and send". Cependant, comme la contrainte originelle forte était la rapidité de réponse à des trames de type probe requests (capture de la trame et injection à la volée d'une trame adaptée de type probe response), il s'était avéré que *Scapy* était trop lent pour répondre dans les délais impartis. Nous avons donc choisi de developper la création de paquets et l'injection de paquet en Python, de manière la plus légère et la plus « codée en dur » possible (remplir les champs invariants tels qu'adresse MAC destination...) afin d'optimiser au mieux les temps de réponse. Cette technique n'a été satisfaisante qu'en partie du fait de la difficulté d'une réponse rapide, mais reste intéressante dans certains environnements[28] où il est difficile de forcer le scan actif.

Comme expliqué dans la section précédente, l'utilisation de la méthode brutale pourrait être implantée via *Scapy* très facilement. Ce sera probablement pour une prochaine version de notre fuzzer !

5.4 Quels champs 802.11 fuzzer ?

Cette partie décrit les champs des trames 802.11. Une bonne connaissance du protocole permet d'identifier les champs des entêtes 802.11 les plus intéressants à fuzzer. Il faut garder en tête les erreurs possibles d'implantations logicielles et comment les déclencher par du fuzzing.

Champ	**Taille**	**Information**
Frame Control	16 bits	Identification de la trame 802.11 et attributs
Duration/ID	16 bits	Mise à jour du NAV et/ou Association ID
Address 1	48 bits	Première adresse MAC
Address 2	48 bits	Deuxième adresse MAC
Address 3	48 bits	Troisième adresse MAC
Sequence Control	16 bits	Numéro de séquence et de fragmentation
Address 4	48 bits	Quatrième adresse MAC (mode WDS[29])
Frame Body	0-2312 octets	Corps du message
FCS	32 bits	Vérificateur d'intégrité (CRC32)

Tab. 1: Format d'une trame 802.11

Dans la table 1, le Frame Control est l'élément critique car il définit le type de trame 802.11. Les autres champs ne semblent pas très prometteurs en terme de fuzzing car il s'agit en particulier des adresses MAC, du duration/ID et du numéro de séquence.

Dans la table 2, le Frame Control est un élément primordial car il définit le type de trame 802.11. Selon les valeurs de protocol, type et subtype, la trame pourra être une trame de management (beacon, probe request...) ou autre. Les valeurs de ces champs sont définis par la norme 802.11 ainsi

[28] typiquement tout ce qui n'est pas un ordinateur portable ou un assistant numérique personnel.

Champ	Taille	Information
Protocol	2 bits	Pour le standard 802.11, la valeur est 0
Type	2 bits	Identification du type de trame 802.11
Subtype	4 bits	Identification du type de trame 802.11
To DS	1 bit	Identification de la direction pour les données
From DS	1 bit	Identification de la direction pour les données
More Frag	1 bit	Fragmentation 802.11
Retry	1 bit	Trame 802.11 retransmise
Pwr Mgt	1 bit	Mode économie d'énergie annoncé par les stations
More Data	1 bit	Paquets en attente côté point d'accès
WEP	1 bit	Positionné à 1 si le frame body utilise WEP
Order	1 bit	Positionné à 1 si le service StrictlyOrdered est activé

Tab. 2: Entête du frame control d'une trame 802.11

que par ses extensions (trames spécifiques pour la QoS, trames de management de type « action » ...). Il apparaît alors intéressant de fuzzer les champs protocol, type et subtype car ils permettent au driver d'identifier le type de trame et donc la charge sous-jacente. Si l'on fuzze de manière aléatoire ici, alors les tests auront pour but d'identifier des dysfonctionnements dans la partie initiale du driver qui consiste en premier lieu à identifier le type de trame avant de l'analyser.

Champ	Taille	Information
Frame Control	16 bits	Identification de la trame 802.11 et attributs
Duration/ID	16 bits	Mise à jour du NAV et/ou Association ID
DA	48 bits	Adresse destination
SA	48 bits	Adresse source
BSSID	48 bits	Adresse du point d'accès
Sequence Control	16 bits	Numéro de séquence et de fragmentation
Frame Body	0-2312 octets	Corps du message
FCS	32 bits	Vérificateur d'intégrité (CRC32)

Tab. 3: Entête de trame de management 802.11

Dans la table 3, les trames de management ont un frame control prédéfini par le standard 802.11. Il impose par exemple un maximum de trois adresses MAC. Les différentes trames de management (beacon, probe request...) sont définies par le subtype permettant alors au driver d'analyser la charge de manière cohérente (les entêtes de trames de beacon et de probe request sont par exemple différents).

Dans la table 4, les trames de beacon et de probe response nous interessent tout particulièrement car nos travaux ont porté sur le fuzzing de l'état 1 des drivers 802.11 clients. Ces trames sont parfaitement identiques seul le frame control permet de les différencier. Les champs timestamp, beacon interval et capability sont obligatoires pour constituer une trame de beacon ou de probe response. Ces champs-là ne semblent pas extrèmement prometteurs pour le fuzzing, mieux vaut se pencher sur les éléments optionnels suivants.

Information	**Taille**	**Type**
Timestamp	64 bits	Horloge de synchronisation
Beacon Interval	16 bits	Intervalle entre beacons
Capability information	16 bits	Capacités du point d'accès

Tab. 4: Entête de trame de beacon ou probe response

Champ	**Taille**	**Type**
Type	1 octet	Type d'information element (ID)
Length	1 octet	Taille de l'information element
Value	Length octet(s)	Valeur de l'information element

Tab. 5: Entête d'Information Element 802.11

Dans la table 5, les information elements sont des éléments optionnels empilés à la suite des entêtes des trames de management tels que les trames de beacon et de probe request. Ces informations elements sont de la forme Type/Length/Value ce qui est extrèmement prometteur car il est par exemple envisageable en terme d'erreurs de programmation que la valeur annoncée de Length (dans la trame 802.11) soit différente de la longueur réelle de Value. De plus, la plupart des informations elements ont des valeurs minimale, fixe ou maximale. Ceci rend possible les erreurs de programmation classiques telles que des débordements de tampons, car le driver devrait normalement[30] vérifier que les copies dans les tampons soient cohérentes avec les tailles définies dans le standard 802.11.

Information Element ID	**Information Element Type**
0	Service Set IDentifier (SSID)
1	Rates
2	FH Parameters
3	DS Parameters
4	CF Parameters
5	Traffic Information Map
7	Country
48	RSN
50	Extended Rates
221	Vendor Specific ou WPA

Tab. 6: Quelques information elements

Dans la table 6, l'erreur la plus vraisemblable est d'allouer un tableau statique de 32 octets pour un information element particulier (le SSID) et de lire la taille réelle à partir du paquet 802.11 et de copier sans vérifier que cette taille réelle est plus grande que celle allouée pour le contenu de l'information element.

[30] évaluer ce « normalement » est une raison d'être du fuzzing !

5.5 Fonctionnalités implantées

Lors de la réalisation de notre fuzzer, nous avons donc implanté plusieurs stratégies de tests à réaliser. Il est à noter que nous nous sommes focalisé sur les informations elements car cet axe d'investigation nous paraît être le plus prometteur. Ces tests seront alors passés en batterie jusqu'à épuisement des tests ou arrêt par les mécanismes de détection de bugs décrits dans les parties précédentes.

Les statégies de tests implantées sont :
- fuzzing optimisé d'Information Elements,
- fuzzing en force brute d'Information Elements,
- fuzzing aléatoire d'Information Elements,
- fuzzing protocolaire d'Information Elements,
- fuzzing protocolaire d'Information Elements spécifiques,
  - Wi-Fi Protected Access (`WPA`), Robust Security Network (`RSN`), Wireless Multimedia Extensions (`WMM`)...
- fuzzing générique d'Information Elements propriétaires,
  - Atheros, Cisco...
- fuzzing par bibliothèque de vulnérabilités découvertes[31].

L'ensemble de ces tests sont sélectionnables en ligne de commande par l'utilisateur afin qu'il puisse réaliser les tests les plus appropriés en fonction des délais impartis pour l'évaluation du driver 802.11.

6 Fuzzing 802.11 en « deux lignes »

Jusqu'à present, nous n'avons abordé que le fuzzer que nous avons développé et qui possède des fonctionnalités évoluées en particulier concernant les Information Elements spécifiques. Cependant, il s'avère que les failles découvertes dans l'état de l'art sont pour la grande majorité découvrables avec des techniques de fuzzing non évoluées. Nous proposons dans cette section deux approches possibles et efficaces pour réaliser du fuzzing 802.11 en deuxlignes[32].

6.1 Fuzzing 802.11 avec Scapy

La fonction `fuzz()` permet de générer aléatoirement des valeurs pour les champs qui ne seraient pas renseignés lors de la création de la trame à envoyer. Il est donc très facile de réaliser un fuzzer 802.11 préliminaire grâce à *scapy*.

Fuzzer de manière (vraiment) aléatoire.

```
frame = Dot11(addr1=DST,addr2=BSSID,addr3=BSSID,addr4=None)
sendp(fuzz(frame), loop=1)
```

Les trames envoyées sont donc toutes les trames possibles 802.11 mais aussi des trames non définies dans le standard. Ce test permet d'évaluer la capacité du driver 802.11 à analyser des trames qui seront pour la grande majorité invalides.

[31] selon l'état de l'art du domaine à l'instant t.
[32] ahhh... Les fameux PEEK et POKE, un brin de nostalgie ne fait pas de mal de temps en temps [29].

Fuzzer de manière aléatoire les Information Elements dans des trames de beacons.

```
frame = Dot11( proto=0,FCfield=0,ID=0,addr1=DST,addr2=BSSID,
               addr3=BSSID,SC=0,addr4=None)
        /Dot11Beacon(beacon_interval=100,cap="ESS")
        /Dot11Elt()
sendp(fuzz(frame), loop=1)
```

Les trames envoyées sont uniquement des trames de type beacon avec des information elements aléatoires qui ne sont pas forcément de taille cohérente par rapport à ce qui est défini dans le standard 802.11. Ce test permet d'évaluer la capacité du driver 802.11 d'analyser un type de trame particulier (les beacons) avec des information elements à la fois valides et invalides.

Fuzzer de manière aléatoire les SSIDs dans des trames de beacons.

```
frame = Dot11( proto=0,FCfield=0,ID=0,addr1=DST,addr2=BSSID,
               addr3=BSSID,SC=0,addr4=None)
        /Dot11Beacon(beacon_interval=100,cap="ESS")
        /Dot11Elt(ID=0)
sendp(fuzz(frame), loop=1)
```

Les trames envoyées sont uniquement des trames de type beacon avec des information elements de type SSID qui ne sont pas forcément de taille cohérente par rapport à ce qui est défini dans le standard 802.11. Ce test permet d'évaluer la capacité du driver 802.11 d'analyser un type de trame particulier (les beacons) avec des information elements de type SSID à la fois valides et invalides.

6.2 Fuzzing 802.11 avec Metasploit

Grâce à l'intégration de la librairie LORCON [30] dans le framework Metasploit [31], il devient aisé de créer et d'injecter des trames 802.11 via des modules Metasploit. Plusieurs exemples sont présents dans la version courante qui permettent entre autres de réaliser des faux points d'accès ou de fuzzer via des probe responses.

Fuzzing de manière aléatoire dans les trames de probe responses.

```
./msfcli    auxiliary/dos/wireless/fuzzproberesp
            DRIVER=madwifing ADDR_DST=11:22:33:44:55:66
            PING_HOST=192.168.1.10
            E
```

Les trames envoyées sont uniquement des trames de type probe response avec des information elements variés qui ne sont pas forcément de taille cohérente par rapport à ce qui est définit dans le standard 802.11. Ce test permet d'évaluer la capacité du driver 802.11 d'analyser un type de trame particulier (les beacons) avec des information elements à la fois valides et invalides. Par ailleurs, le module permet de tester si l'équipement qui est fuzzé répond bien à des pings afin de continuer ou non le protocole de test.

7 Vulnérabilités découvertes par notre fuzzer

Bien entendu les tests réalisés grâce à notre fuzzer ont toujours été dans une volonté de découvrir de nouvelles vulnérabilités et non de re-découvrir des vulnérabilités connues. Par conséquent, les tests de fuzzing ont été réalisés à chaque fois avec les dernières versions des drivers selon les systèmes d'exploitation utilisés. Il nous a été aussi possible de découvrir des vulnérabilités qui ont été corrigées silencieusement d'une version de driver à l'autre, mais ces vulnérabilités ne peuvent être considérées comme inconnues puisque corrigées par l'éditeur...

Le passage de notre fuzzer sur de nombreuses cartes 802.11 a permis d'identifier plusieurs erreurs de programmation qui pour certaines apparaissent exploitables à distance. Nous nous sommes focalisés sur celle qui était de loin la plus intéressante : la première faille exploitable à distance dans un driver 802.11 sous Linux et qui plus est sur un chipset très utilisé dans la communauté « hacking », l'Atheros.

7.1 Netgear MA521 Wireless Driver Long Rates Overflow (CVE-2006-6059)

Cette vulnérabilité a été publiée le 18 novembre 2006 via le Month of Kernel Bugs [32].

L'utilisation d'un Information Element de type "Rates" trop long permet de déclencher le dysfonctionnement. Selon la trace du crash kernel, cela semble être un débordement sur le tas.

Nous n'avons pas investigué plus profondément et seulement publié le module Metasploit pour réaliser le déni de service. A ce jour, aucune mise à jour corrigeant la vulnérabilité n'est disponible.

7.2 Netgear WG311v1 Wireless Driver Long SSID Overflow (CVE-2006-6125)

Cette vulnérabilité a été publiée le 22 novembre 2006 via le Month of Kernel Bug [33].

L'utilisation d'un Information Element de type "SSID" trop long permet de déclencher le dysfonctionnement. Selon la trace du crash kernel, cela semble être un débordement sur la pile.

Nous n'avons pas investigué plus profondément et seulement publié le module Metasploit pour réaliser le déni de service. A ce jour, aucune mise à jour corrigeant la vulnérabilité n'est disponible.

7.3 D-Link DWL-G650+ Wireless Driver Long TIM Overflow (CVE-2007-0933)

Cette vulnérabilité a été publiée le 28 mars 2007 lors de la conférence BlackHat Europe [34,35].

L'utilisation d'un Information Element de type "TIM" trop long permet de déclencher le dysfonctionnement. Selon la trace du crash kernel, cela semble être un débordement[33] sur la pile.

Nous n'avons pas investigué plus profondément et seulement publié le module Metasploit pour réaliser le déni de service. A ce jour, aucune mise à jour corrigeant la vulnérabilité n'est disponible.

7.4 Madwifi Driver "giwscan_cb()" and "encode_ie()" Remote Buffer Overflow Vulnerability (CVE-2006-6332)

Cette vulnérabilité a été publiée le 7 décembre 2006 sur la liste de diffusion Daily Dave [36]. Vu la criticité de la faille découverte, nous avons averti l'équipe de développement du driver Madwifi et attendu la sortie de la mise à jour avant d'annoncer la vulnérabilité et de publier une preuve de concept d'exploitation locale du driver.

[33] forte suspicion pour un off-by-one sur `RET`...

Nous félicitons l'équipe de développement de madwifi qui a été extrèmement réactive en publiant un nouveau paquetage du driver le lendemain de notre contact avec eux.

L'utilisation d'un Information Element de type `WPA`, `RSN`, `ATH` ou `WMM` trop long permet de déclencher le dysfonctionnement. Selon la trace du "oops", il s'agit d'un débordement sur la pile.

8 Exemple d'une faille de sécurité exploitable

Dans cette partie, nous allons brièvement expliquer où se trouve la faille et comment la déclencher grâce à une trame 802.11 bien formatée. Il faut alors se rapeller qu'un seule trame 802.11 peut alors permettre l'exécution de code arbitraire à distance en mode ring0 sous Linux ! Rien que ça !

8.1 Introduction

Notre fuzzer a une brique adaptée pour le fuzzing protocolaire d'information elements spécifiques tels que `WPA`, `RSN`... C'est grâce à ces fonctionnalités que la vulnérabilité a pu être découverte, lors de tests sur les briques développées dans notre fuzzer d'information elements de type `WPA`.

En effet, le driver analyse les informations elements et vérifie que l'information element est bien un `WPA` avant de continuer l'analyse (pour entre autres connaître les méthodes d'authentification et de chiffrement supportées par le point d'accès). Pour se faire, il est nécessaire d'avoir les champs `OUI`[34], `TYPE` et `VERSION` bien renseignés pour la version de `WPA` supportée par le driver 802.11 testé.

Par conséquent, un fuzzer « basique » d'information elements n'aurait pas été capable de découvrir cette vulnérabilité car il est nécessaire que le fuzzer ait une connaissance minimale du protocole à fuzzer (dans notre cas `WPA`). C'est certainement une des raisons pour laquelle cette vulnérabilité n'a pas été découverte plus tôt ! Pourtant le code source était disponible depuis longtemps et la faille saute aux yeux en lisant[35] au bon endroit !

A noter que cette vulnérabilité est déclenchable aussi sur les information elements de type `RSN`, `WMM` et `ATH`.

```
BUG:
unable to handle kernel paging request at virtual address 45444342
 printing eip:
45444342
*pde = 00000000
Oops: 0000 [#1]
PREEMPT
CPU:    0
EIP:    0060:[<45444342>]    Tainted: P      VLI
EFLAGS: 00210282   (2.6.17.11 #1)
EIP is at 0x45444342
eax: 00000000   ebx: 41414141   ecx: 00000000   edx: f4720bde
esi: 41414141   edi: 41414141   ebp: 41414141   esp: f3f2be24
ds: 007b   es: 007b   ss: 0068
Process iwlist (pid: 3486, threadinfo=f3f2a000 task=f6f8a5b0)
```

[34] Organizationally Unique Identifier.

[35] qui a affirmé que l'Open Source était plus sûr car le code était lu ? ! ?

Dans ce "oops" d'un noyau Linux, il est remarquable que plusieurs registres sont contrôlés par l'attaquant et en particulier un registre très intéressant : `EIP` ! Il est très probable, vu que nous contrôlons `EIP` et `EBP` que nous soyions en présence d'un stack buffer overflow. Si l'attaquant contrôle `EIP`, cela ouvre immédiatement la voie à de l'exécution de code arbitraire. Le but du jeu sera donc de faire pointer `EIP` vers du code intéressant pour l'attaquant, par exemple embarqué dans la trame 802.11 qui aura permis l'exploitation de la vulnérabilité.

8.2 Descriptif des erreurs de programmation découvertes

Dans la fonction `giwscan_cb()`.

```
static void
giwscan_cb(void *arg, const struct ieee80211_scan_entry *se)
{
    struct iwscanreq *req = arg;
    struct ieee80211vap *vap = req->vap;
    char *current_ev = req->current_ev;
    char *end_buf = req->end_buf;
#if WIRELESS_EXT > 14
    char buf[64 * 2 + 30];
#endif
```

Les noyaux récents compilés avec les wireless extensions sont supérieurs à la version 14 par conséquent, le tableau statique `buf` de 158 octets est alloué.

```
<snip>

#ifdef IWEVGENIE
        memset(&iwe, 0, sizeof(iwe));
        memcpy(buf, se->se_wpa_ie, se->se_wpa_ie[1] + 2);
        iwe.cmd = IWEVGENIE;
        iwe.u.data.length = se->se_wpa_ie[1] + 2;
```

En pratique, si `IWEVGENIE` est défini, ce qui est le cas dans tous les noyaux récents, nous avons une première vulnérabilité dans le `memcpy()` qui prend comme longueur à recopier dans le buffer `buf` la longueur spécifiée dans l'information element `WPA` de la trame 802.11 ajoutée de 2. La taille maximale de `se->se_wpa_ie[1]` n'est jamais vérifiée dans aucune autre partie du code du driver par conséquent elle est complétement maîtrisée par l'attaquant. La taille maximale copiable dans `buf` est donc de 257 octets alors que sa taille est de 158 octets : nous avons donc un débordement de pile !

```
#else
        static const char wpa_leader[] = "wpa_ie=";
        memset(&iwe, 0, sizeof(iwe));
        iwe.cmd = IWEVCUSTOM;
        iwe.u.data.length = encode_ie(buf, sizeof(buf),
            se->se_wpa_ie, se->se_wpa_ie[1] + 2,
            wpa_leader, sizeof(wpa_leader) - 1);
#endif
```

En pratique, si `IWEVGENIE` n'est pas défini, la fonction `encode_ie()` est appelée avec des paramètres dont la taille n'est pas vérifiée. Si une seconde vulnérabilité existe, alors elle est présente dans la fonction `encode_ie()`.

```
<snip>

    if (se->se_wme_ie != NULL) {
        static const char wme_leader[] = "wme_ie=";

        memset(&iwe, 0, sizeof(iwe));
        iwe.cmd = IWEVCUSTOM;
        iwe.u.data.length = encode_ie(buf, sizeof(buf),
            se->se_wme_ie, se->se_wme_ie[1] + 2,
            wme_leader, sizeof(wme_leader) - 1);

<snip>

    if (se->se_ath_ie != NULL) {
        static const char ath_leader[] = "ath_ie=";

        memset(&iwe, 0, sizeof(iwe));
        iwe.cmd = IWEVCUSTOM;
        iwe.u.data.length = encode_ie(buf, sizeof(buf),
            se->se_ath_ie, se->se_ath_ie[1] + 2,
            ath_leader, sizeof(ath_leader) - 1);

<snip>
```

En pratique, avec des information elements WMM ou ATH, le fonctionnement est équivalent à des informations elements avec un noyau avec `IWEVGENIE` non défini, la fonction `encode_ie()` est appelée avec des paramètres dont la taille n'est pas vérifiée. Si vulnérabilité il y a, alors elle est présente dans la fonction `encode_ie()`.

Dans la fonction `encode_ie()`.

```
static u_int
encode_ie(void *buf, size_t bufsize, const u_int8_t *ie,
    size_t ielen, const char *leader, size_t leader_len)
{
    u_int8_t *p;
    int i;

    if (bufsize < leader_len)
        return 0;
    p = buf;
    memcpy(p, leader, leader_len);
    bufsize -= leader_len;
```

```
    p += leader_len;
    for (i = 0; i < ielen && bufsize > 2; i++)
        p += sprintf(p, "%02x", ie[i]);
    return (i == ielen ? p - (u_int8_t *)buf : 0);
}
```

Ici, la variable `ielen` est maîtrisée par l'attaquant et `p` est un pointeur sur le tableau statique `buf`. La fonction `sprintf()` permet alors de réaliser un débordement de pile en convertissant le contenu de l'information element en ASCII dans `buf`. A noter que si l'on écrase des pointeurs avec le contenu de la trame 802.11, alors le résultat est le contenu converti en ASCII.

8.3 Exploitation à distance

Vu que nous avons le contrôle de `EIP`, le but du jeu est de le faire pointer vers du code maîtrisé par nous-même. La faille s'étant déclenchée dans un contexte de processus (faisant un appel système) nous pouvons potentiellement exécuter tout code situé dans l'espace d'adressage de celui-ci, y compris les pages du noyau puisque nous sommes en mode noyau. Dans le cas d'*iwlist*, nous pouvons relativement facilement contrôler une partie de son espace mémoire pour y injecter du code, puisque ce programme va créer sur son tas des structures de données relative à son environnement 802.11 que nous controlons, au moins partiellement.

Cependant nous pouvons aussi nous affranchir du programme ayant réalisé l'appel système en injectant directement notre code dans l'information element se retrouvant dans la pile noyau (dans `buf`) au moment où le bug est déclenché. Nous avons a priori suffisamment de place pour embarquer des shellcodes intéressants car la taille de `buf` est de 158 octets. Parmi ces 158 octets seuls les 6 premiers ne sont pas utilisables car ils sont réservés pour l'identification de l'information element en tant que `WPA` (`OUI + TYPE + VERSION`).

Il nous suffit alors de localiser ce code. Une possibilité[36] consiste alors à faire pointer `EIP` vers un `jmp esp` situé dans l'espace d'adressage du processus (déjà présent ou que nous aurons injecté). Il est, là aussi, possible de s'affranchir complètement du processus ayant réalisé l'appel système car un `jmp esp` est présent dans l'avant dernière page de l'espace d'adressage de n'importe quel processus, le VDSO[37].

Notre `jmp esp` va nous emmener sur la pile noyau, juste après l'adresse de retour que nous avons réécrite, sur le premier argument de la fonction vulnérable. Pour ne pas avoir à réécrire le second argument, nous avons réécrit le premier argument avec un `jmp short` afin de sauter en arrière vers le coeur de l'information element. Un `jmp short` peut faire l'affaire car il est codé sur 2 octets, mais la contrainte est qu'il ne peut aller plus loin que -128 octets. Nous avons donc choisi de réaliser un deuxième `jmp` en arrière afin d'optimiser la taille disponible pour le shellcode dans l'information element `WPA` de la trame 802.11 déclenchant la vulnérabilité.

A partir de cette étape, nous avons alors l'exécution de code arbitraire à distance sur un Linux ayant un madwifi vulnérable. Nous ne dépendons absolument pas du programme ayant réalisé l'appel système `ioctl` déclenchant la vulnérabilité. Nous dépendons seulement de l'exécutabilité de la pile noyau et de la présence du VDSO à l'avant dernière page (certains patches de sécurité peuvent

[36] en exploitation de vulnérabilité de nombreux chemins mènent à Rome...

[37] librairie dynamique virtuelle utilisée par le processus en mode utilisateur pour réaliser des appels système. Elle est mise en place par le kernel. Il est très intéressant de noter que le `jmp esp` présent dans cette page ne fait pas partie du VDSO, mais du résidu de mémoire situé après celui-ci.

rendre aléatoire son adresse ou le supprimer), ce qui signifie que notre exploit est extrèmement stable. La problématique suivante est la réalisation d'un shellcode noyau. Cela ne rentre pas dans le cadre de cet article et nous vous proposons de vous reporter à l'article de Stéphane Duverger [39] ou à notre exploit pour de plus amples informations.

8.4 Déclenchement de la vulnérabilité

La fonction `giwscan_cb()` qui est vulnérable est appelée via `SIOCGIWSCAN`. Cet appel permet d'analyser les résultats d'un scan et de les présenter (par exemple) à l'utilisateur. Typiquement, l'utilitaire *iwlist* réalise cet appel lors de la ligne de commande `iwlist ath0 scanning` en utilisateur non privilégié. Cependant, pour que la vulnérabilité soit réellement déclenchée, il est nécessaire que les paquets spécialement forgés soient passés via la structure `const struct ieee_scan_entry *se`. Or ceci n'est possible que si un scan des réseaux 802.11 a permis de mettre à jour les données soit :

- manuellement via un appel à `SIOCSIWSCAN` (grâce par exemple à l'utilisation de la commande `iwlist ath0 scanning` en utilisateur privilégié),
- automatiquement par le driver 802.11 lors des procédures de « background scanning » configurables par des paramêtres internes (tels que `bgscan`, `bgscanidle` et `bgscanintvl`).

9 Automatisation de l'exploitation de vulnérabilités 802.11

Grâce à des techniques de prise d'empreinte de cartes 802.11, il est envisageable de selectionner l'exploit le plus adapté selon la cible. En conséquence, l'automatisation de l'exploitation de vulnérabilités de drivers 802.11 est tout a fait réalisable en particulier dans des zones denses telles que des hot spots ou des conférences.

Un outil approprié dans ce cadre est Metaploit qui, depuis l'intégration de LORCON et de ses bindings ruby, a une bibliothèque d'exploits 802.11 (en particulier en environnement Windows).

10 Conclusions

Dans cet article, nous espèrons avoir démystifié les vulnérabilités dans les drivers 802.11. Les failles découvertes sont relativement triviales à découvrir et un fuzzer moins évolué que le notre aurait pu trouver la majorité des failles découvertes à ce jour dans l'état de l'art. De plus, notre fuzzer a permis de trouver de nombreuses failles alors qu'il n'était focalisé que sur l'état 1. Un fuzzer digne de ce nom devrait gérer proprement les états pour être considéré comme efficace, or ce n'est pas complètement trivial en 802.11, contrairement à la plupart des autres protocoles réseaux classiques[38] qui n'ont pas de contraintes fortes en terme d'accès au medium.

Nos axes de réflexion actuels portent donc sur l'implantation de techniques de fuzzing 802.11 avec états aussi bien coté client que point d'accès.

[38] i.e. non directement liés à la couche physique.

11 Remerciements

Nous tenons à remercier Jérôme Razniewski pour ses travaux sur le 802.11 et sur la découverte de la faille sur madwifi, Yoann Guillot pour le fameux `metasm` [40] qui nous a facilité la tâche dans le développement de l'exploit, Raphaël Rigo pour l'analyse des vulnérabilités Windows, Franck Veysset et Matthieu Maupetit pour leur relecture attentive.

Références

1. IEEE, *Local and Metropolitan Area Networks, Specific Requirements, Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1997-1999.
2. Johnny Cache and David Maynor, *Device Drivers*,
`http ://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Cache.pdf`, 2006.
3. BlackHat Conference,
`http ://www.blackhat.com`
4. Johnny Cache and David Maynor, *Hijacking a MacBook in 60 seconds*,
`http ://www.youtube.com/watch ?v=chtQ1bcHLZQ`, 2006.
5. Billet de James W. Thompson,
`http ://he-colo.netgate.com/archives/00000465.htm`, 2006.
6. Billet de Cédric Blancher,
`http ://sid.rstack.org/blog/index.php/2006/10/02/133-se-paierait-on-notre-pomme`, 2006.
7. David Maynor, *Its V-A day....*,
`http ://erratasec.blogspot.com/2007/02/its-v-day.html`, 2007.
8. Robert Lemos, *Maynor reveals missing Apple Flaws*,
`http ://www.securityfocus.com/news/11445`, 2007.
9. Mail de Johnny Cache,
`http ://lists.immunitysec.com/pipermail/dailydave/2006-September/003459.html`, 2006.
10. Wikipedia, Rings,
`http ://en.wikipedia.org/wiki/Ring_%28computer_security%29`
11. Fuzzing Mailing List, *The definition of what a fuzzer really is... is fuzzy*,
`http ://www.whitestar.linuxbox.org/pipermail/fuzzing/2006-May/000033.html`, 2006.
12. Month of Browser Bugs,
`http ://browserfun.blogspot.com/`, July 2006.
13. Month of Kernel Bugs,
`http ://kernelfun.blogspot.com/`, November 2006.
14. Month of Apple Bugs,
`http ://applefun.blogspot.com/`, January 2007.
15. Month of PHP Bugs,
`http ://www.php-security.org/`, March 2007.
16. L.M.H., *fsfuzzer*,
`http ://projects.info-pull.com/mokb/fsfuzzer-0.6-lmh.tgz`, 2006.
17. Jean Tourillhes, *Wireless Tools for Linux*,
`http ://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html`, 1996-2007.
18. Dino A. Dai Zovi and Shane Macaulay, *Attacking Automatic Wireless Network Selection*,
`http ://www.theta44.org/karma/aawns.pdf`, 2005.

19. Toast, *airpwn*,
http ://sourceforge.net/projects/airpwn, 2004-2006.
20. Cédric Blancher, *wifitap*,
http ://sid.rstack.org/index.php/Wifitap, 2005-2006.
21. Laurent Butti, *Raw Glue AP*,
http ://rfakeap.tuxfamily.org/, 2005-2006.
22. Johnny Cache, *fuzz-e*,
http ://www.802.11mercenary.net/code/airbase-latest-svn.tar.gz, 2006.
23. NetStumbler.com, *NetStumbler*,
http ://www.netstumbler.com/, 2001-2007.
24. Joshua Wright, Seng Ooh Too, Mike Kershaw, *802.11b Firmware-Level Attacks*,
http ://802.11ninja.net/papers/firmware_attack.pdf, 2006.
25. Multiband Atheros Driver for Wireless Fidelity, *madwifi*,
http ://www.madwifi.org/, 2004-2006.
26. Aircrack-ng, *madwifi-ng injection patch*,
http ://patches.aircrack-ng.org/madwifi-ng-r1816.patch, 2006.
27. Johnny Cache, HD Moore, skape, *Exploiting 802.11 Wireless Driver Vulnerabilities on Windows*,
http ://www.uninformed.org/ ?v=6&a=2&t=sumry, 2006.
28. Philippe Biondi, *Scapy*,
http ://www.secdev.org/scapy, 2003-2007.
29. Shift Éditions, *Hebdogiciel*,
http ://fr.wikipedia.org/wiki/Hebdogiciel, 1984-1987.
30. Joshua Wright, Mike Kershaw, *Loss Of Radio CONnectivity*,
http :
www.802.11mercenary.net/lorcon/, 2006-2007.
31. Metasploit LLC, *Metasploit*,
http ://www.metasploit.com/, 2003-2007.
32. Laurent Butti, *NetGear MA521 Wireless Driver Long Rates Overflow (CVE-2006-6059)*,
http ://kernelfun.blogspot.com/2006/11/mokb-18-11-2006-netgear-ma521-wireless.html, 2006.
33. Laurent Butti, *NetGear WG311v1 Wireless Driver Long SSID Overflow (CVE-2006-6125)*,
http ://kernelfun.blogspot.com/2006/11/mokb-22-11-2006-netgear-wg311v1.html, 2006.
34. Laurent Butti, *D-Link DWL-G650+ Wireless Driver Long TIM Overflow (CVE-2007-0933)*,
http ://cve.mitre.org/cgi-bin/cvename.cgi ?name=CVE-2007-0933, 2007.
35. Laurent Butti, *Wi-Fi Advanced Fuzzing*,
http ://www.blackhat.com/html/bh-europe-07/bh-eu-07-speakers.html#Butti, 2007.
36. Laurent Butti, Jérôme Razniewski, Julien Tinnès, *Madwifi SIOCSIWSCAN vulnerability (CVE-2006-6332)*,
http ://archives.neohapsis.com/archives/dailydave/2006-q4/0291.html, 2006.
37. Madwifi, *Release 0.9.2.1 fixes critical security issue*,
http ://kernelfun.blogspot.com/2006/11/mokb-22-11-2006-netgear-wg311v1.html, 2006
38. Julien Tinnès and Laurent Butti, *madexploit.c*,
http ://archives.neohapsis.com/archives/dailydave/2006-q4/att-0298/madexploit.c, 2006.
39. Stéphane Duverger, *Exploitation en espace noyau*,
http ://www.sstic.org/SSTIC07/programme.do#DUVERGER, 2007.
40. Yoann Guillot, *metasm*,
http ://www.sstic.org/SSTIC07/programme.do#GUILLOT, 2006-2007.