afl-taenia-mt : fuzzing en mémoire pour les cibles multi-threadées et en boîte noire.

Julien Rembinski, Benjamin Dufour, Simon Lebreton et Frédéric Garreau
julien.rembinski@zaclys.net
benjamin.dufour.ssi@gmail.com

DGA-MI

Résumé. Confrontés au besoin de *fuzzer* une cible complexe, en boîte-noire et multi-threadée, dans un environnement Unix, nous avons rencontré plusieurs limites d'afl-qemu. Cet article présente les solutions mises en place pour surmonter ces limites. Il présente également la preuve de concept afl-taenia-mt qui n'est pas un outil clé en main, mais contient un exemple d'implémentation de ces solutions.

1 Introduction

Avant de commencer nos travaux, nous avons repris un état de l'art effectué en interne précédemment sur les outils de *fuzzing*. Cet état de l'art a mis en lumière deux outils : afl [1] et honggfuzz [2]. Ils permettent tous deux de *fuzzer* une cible en boîte noire en l'émulant sous qemu [4]. Dans cette configuration, les fonctionnalités les plus avancées de ces outils ne sont plus utilisables (ASAN, MSAN, *comparison unrolling*...). Afl++ est un projet en source ouverte maintenant afl et y intégrant les fonctionnalités les plus avancées des autres fuzzers. Nous avons finalement choisi afl++ pour le reste du projet.

1.1 Problématique

Notre étude concerne le *fuzzing* en boîte-noire de systèmes d'information proposant une architecture applicative complexe (nombreux processus interdépendants et multi-*threadés*, fonctionnant dans un environnement Unix. Ces cibles ont en outre de nombreuses contraintes, qui ont mis en évidence plusieurs limites d'afl :

1. Le processus cible prend beaucoup de temps à s'initialiser.
   — Mais afl réinitialise la cible à chaque exécution.
2. Le processus cible fait partie d'un écosystème de processus, réinitialiser le processus cible sans redémarrer l'écosystème mène à des instabilités. Certaines entrées d'afl peuvent provoquer des erreurs dans les autres processus de l'écosystème.

   — Mais afl ne s'intéresse pas aux autres processus.
3. Le code ciblé est enfoui dans le binaire cible.
   — Mais afl fournit ses entrées au binaire cible en ligne de commande, par stdin ou par fichier.
4. Le code ciblé peut être situé dans une bibliothèque externe.
   — Mais afl ne suit pas le code des bibliothèques extérieures au binaire cible.
5. Les données issues des entrées d'afl peuvent être manipulées par plusieurs threads du processus cible.
   — Mais afl ne sait pas différencier le code exécuté dans des threads différents.
6. Le processus cible possède des threads ne manipulant pas de données issues des entrées d'afl.
   — Mais afl ne permet pas de les ignorer.

Ces deux derniers points sont détaillés dans les paragraphes suivants.

1.2 Suivi de chemins par afl-qemu

Afl est un *fuzzer* par mutations. Il se base donc sur des entrées qu'il mute selon certaines heuristiques, déterministes ou non. L'ensemble de ces entrées est initialisé par l'utilisateur, puis complété au fur et à mesure qu'afl trouve des entrées menant à de nouveaux chemins dans le graphe de flot de contrôle. Ce mécanisme amène afl à *fuzzer* la cible en profondeur.

Pour ce faire, afl introduit des sondes dans sa cible. Dans le cadre d'une cible fournie au format binaire, sans les sources, afl se base sur qemu-user.

Qemu-user est un émulateur. Il fonctionne de la façon suivante : il traduit les instructions du binaire dans son langage intermédiaire, qui est lui même retraduit dans le jeu d'instuctions de l'architecture locale. Cela nous permet donc de *fuzzer* des binaires compilés sur l'ensemble des architectures supportées par qemu (x86, x86_64, arm, powerpc ... [3]).

Afl ajoute des sondes dans le langage intermédiaire pour chaque bloc de base de la cible, ce qui lui permet d'identifier les blocs parcourus par une exécution. La figure 1 illustre ce mécanisme. Deux entrées différentes sont fournies successivement au même binaire. Elles mènent à l'exécution de blocs de base différents. Au final, afl est capable de déterminer, que les entrées ont mené à des chemins différents dans le graphe de flot de contrôle. Elles seront donc toutes deux sauvegardées.

Ainsi afl trace les transitions entre les blocs de base rencontrées lors de l'exécution d'une entrée. Un chemin est alors un ensemble de transitions

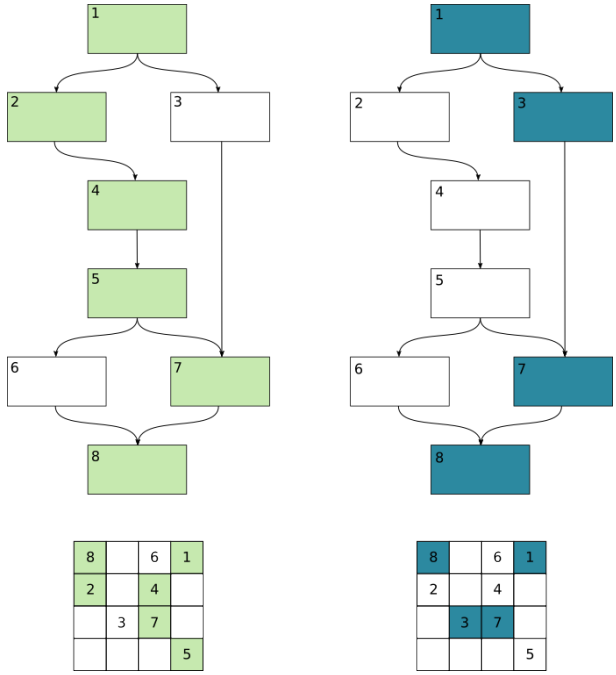

Fig. 1. Illustration du calcul d'empreinte pour les transitions rencontrées par afl

non ordonné. Cela permet d'avoir une métrique de couverture de code. Son but est de découvrir de nouveaux chemins augmentant cette couverture.

1.3 Absence de suivi intelligent des threads

Commé évoqué précédemment, afl reconstruit un chemin à partir d'un ensemble non ordonné de transitions. Malheureusement, il n'a pas la connaissance du thread qui exécute le bloc de base courant. Lorsque deux threads s'exécutent en parallèle, ils exécutent simultanément deux chemins différents au niveau du processeur. Afl trace un mélange de ces deux chemins selon l'ordre d'exécution des blocs de base.

Ainsi, les chemins tracés ne sont pas pertinents. Afl sauvegarde des entrées inutiles et a donc des difficultés à progresser dans le parcours du graphe de flot de contrôle.

Pour la même raison, les threads inutiles engendreront des chemins qui n'ont pas de lien avec l'entrée afl et viendront donc polluer les chemins légitimes.

Pour illustrer ce problème, nous avons développé un programme de test en C intégrant des vulnérabilités. L'architecture de ce programme de test est composée comme suit :

— Un thread "Main" qui lance tous les autres threads.
— Un thread "Broker" qui écoute sur le réseau et stocke les entrées reçues dans une file d'attente.
— Un thread "Parser" qui traite les données situées dans la file d'attente.
— Un thread inutile qui fait simplement du bruit.

Les vulnérabilités sont implémentées dans le thread "Parser" et comprennent :

— Un *crash* (un appel à *abort()*).
— Une boucle sans fin (un while(true)).
— Un *crash* pouvant être déclenché par une série de deux messages (non nécessairement successifs), nommé '*crash stateful*'.

La figure 2 montre une séquence d'exécution normale de cette architecture, puis la figure 3 montre ce qu'afl trace et en déduit.

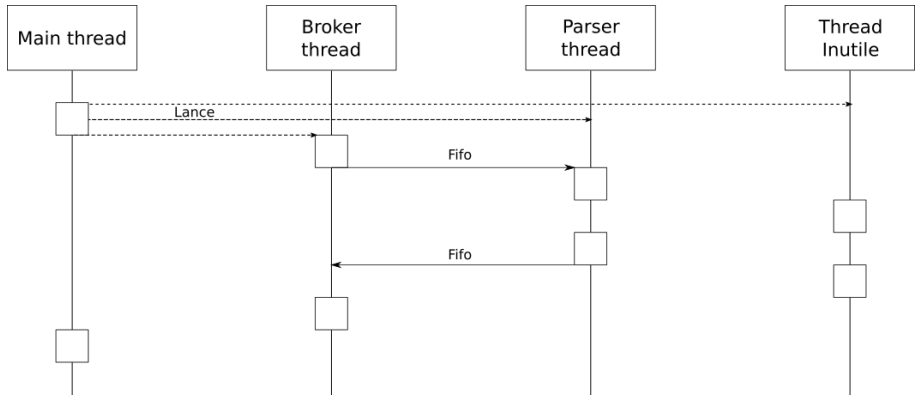

Fig. 2. Séquence d'exécution de notre architecture multi-thread de test

Afl n'a pas connaissance des threads, il considère donc que tout fait partie du même thread et assemble donc les blocs de base appartenant à plusieurs threads. Selon le comportement des threads 'inutiles', cela peut amener la même entrée à générer des chemins différents dans le binaire. Ainsi afl trouve de nouveaux chemins virtuels à chaque exécution et est donc incapable de progresser.

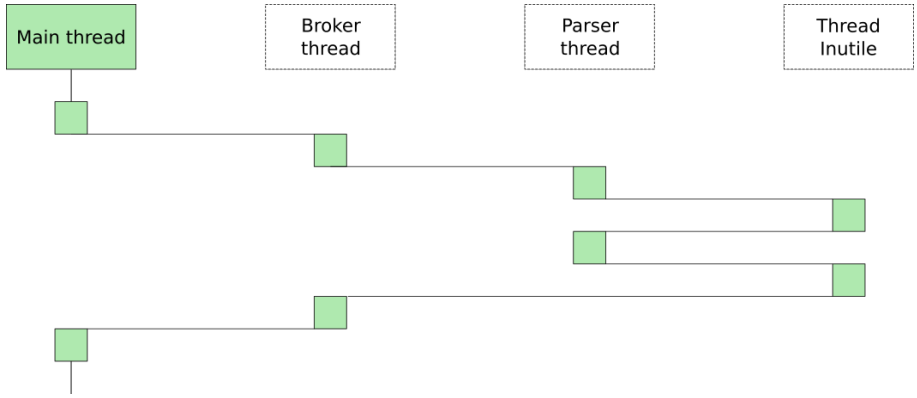

Fig. 3. Compréhension par afl de cette architecture

1.4 Illustration de l'impact sur les performances du *fuzzing*

L'architecture précédente a été dérivée en deux variantes :

— Variante multi-threadée : l'entrée est récupérée par le Broker, tous les threads présentés précédemment coexistent.

— Variante simplifiée : l'entrée est récupérée par le Parser. Le Broker et le thread inutile n'existent pas.

Pour illustrer l'impact sur les performances du *fuzzing*, nous lançons afl++ sur chacune de ces variantes. Le programme cible est simple, cinq minutes de *fuzzing* suffisent à le couvrir et le résultat est déterministe.

La figure 4 montre les résultats de ce *fuzzing* sur la variante simplifiée. Nous observons que le graphe de flot de contrôle a été majoritairement parcouru (il y a une vingtaine de chemins possibles), et que le *crash* simple et le *hang* ont été trouvés [1].

La figure 5 montre les résultats d'une heure de *fuzzing* sur la variante multi-threadée. Aucun *crash*, ni *hang* n'a été trouvé. Afl affirme avoir trouvé de nombreux chemins (plus qu'il n'y en a réellement dans le binaire), en réalité la fonction de parsing n'a été explorée que superficiellement. Ces chemins sont des mélanges des transitions des threads utiles et du thread inutile. L'information se retrouve au niveau de la stabilité vue par afl. Lorsqu'afl trouve un nouveau chemin avec une entrée donnée, il réexécute automatiquement le binaire avec cette entrée et, s'il obtient le même chemin, l'entrée est dite stable, sinon, l'entrée est dite instable. Cette stabilité de 66,20% souligne donc parfaitement notre problème.

1. L'auteur d'afl explique les différents champs de cet écran de statut sur son site [5].

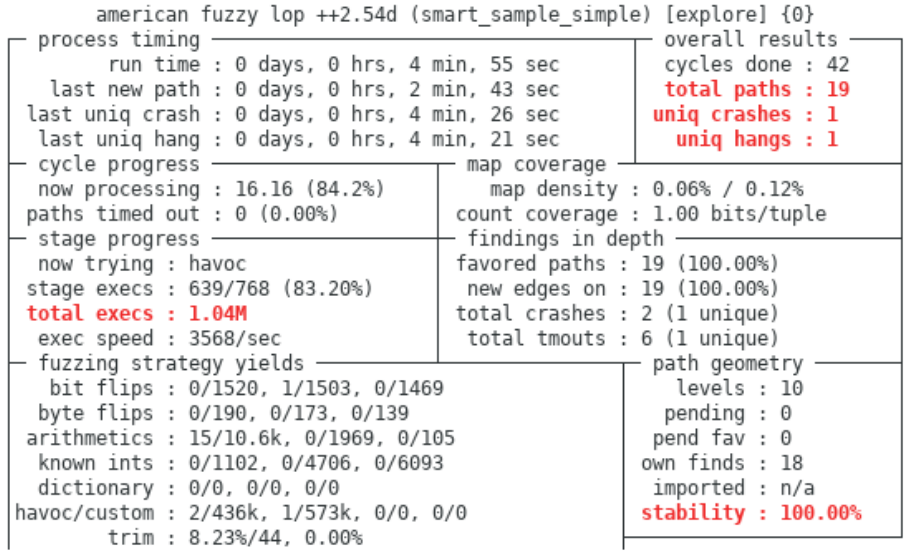

Fig. 4. Performances d'afl sur notre exemple en variante simplifiée

Il faut noter également que le changement de variante de la cible entraîne une baisse de vitesse de *fuzzing*, principalement due au redémarrage de la cible à chaque itération [2]. Mais cela n'empêche pas le *fuzzer* de trouver presque trois fois plus de chemins dans le même binaire.

2 afl-taenia-mt

Notre solution se base sur afl++ version 2.54d et ajoute les fonctionnalités suivantes :

1. *Fuzzing* en mémoire.
2. Suivi des bibliothèques externes.
3. Suivi de thread.
4. Mode *stateful*.
5. Fonctionnalité de rejeu.

2. C'est pourquoi nous avons lancé ce *fuzzing* sur une durée plus longue, il faut observer que dans tous les cas, aucun nouveau chemin n'avait été trouvé depuis plusieurs minutes.

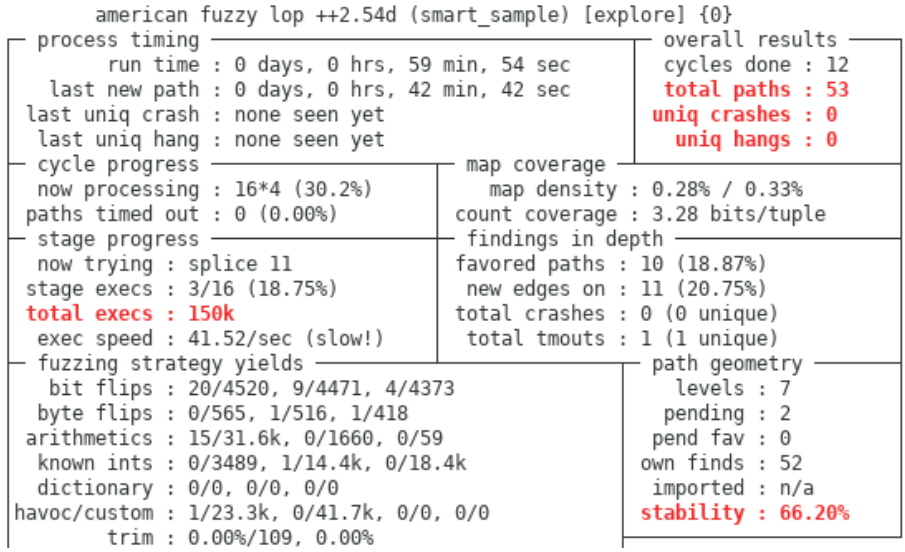

Fig. 5. Performances d'afl sur notre exemple en variante multi-threadée

2.1 Architecture

Notre solution entraîne des modifications dans l'architecture d'afl-qemu qu'il est nécessaire d'expliquer pour la suite de l'article.

Architecture d'afl avec qemu L'ensemble afl-qemu se décompose en plusieurs processus :

- afl gère l'intelligence du *fuzzing*, il fournit des entrées à partir des retours d'exécution et des transitions rencontrées.
- Le *forkserver* fait le lien entre afl et la cible instrumentée. Il redémarre cette dernière à chaque itération.
- Le processus fils de qemu, qui est créé par le *forkserver*, exécute par émulation la cible et renvoie le code de retour et les instructions rencontrées.

La figure 6 résume les interactions entre ces processus.

Architecture d'afl-taenia-mt avec qemu afl-taenia-mt injecte sa bibliothèque libtaenia dans le processus cible émulé par qemu [3]. Un canal de communication entre cette bibliothèque et les deux entités qemu est égale-

3. Elle n'est pas injectée dans le processus qemu.

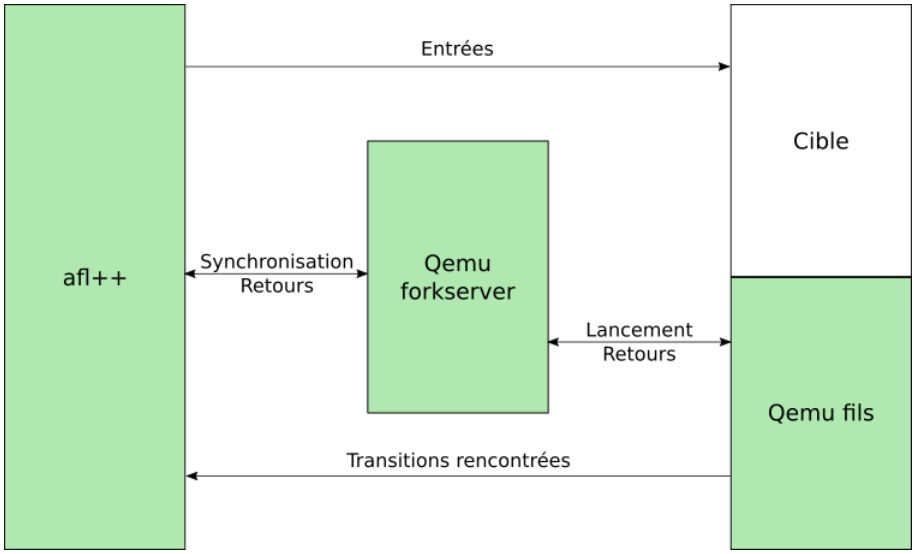

Fig. 6. Architecture d'afl

ment ajouté de façon à échanger les données relatives au fonctionnement d'afl-taenia-mt.

Le *forkserver* a été intégralement réécrit pour nos besoins. Le code du fils qemu a été modifié de façon à prendre en compte l'identifiant du thread courant, le code des bibliothèques externes et à pouvoir communiquer avec le nouveau forkserver. Le code d'afl++ a été légèrement modifié pour des raisons de débogage décrites ultérieurement.

La figure 7 présente ces modifications.

Libtaenia exécute un thread dédié à nos besoins. Pour cela, nous hookons une fonction importée et exécutée à un moment intéressant par le processus cible, idéalement à la fin de son initialisation. Pour ce faire, nous avons utilisé le mécanisme LD_PRELOAD (voir paragraphe 2.3).

Nous avons configuré afl de façon à envoyer ses entrées *fuzzées* dans un fichier en ram. Ce fichier est lu par le thread libtaenia qui exécute alors la fonction cible avec l'entrée issue de ce fichier. Qemu effectue ses retours vers afl comme lors d'un fonctionnement standard. Un espace mémoire est partagé entre les processus qemu père et fils [4]. Cet espace sert à échanger les données de configuration et de travail des options avancées qui seront détaillées ultérieurement.

4. Attention, ces processus peuvent fonctionner avec des architectures différentes. Il est donc important que la structure de cette mémoire soit indépendante de l'architecture.

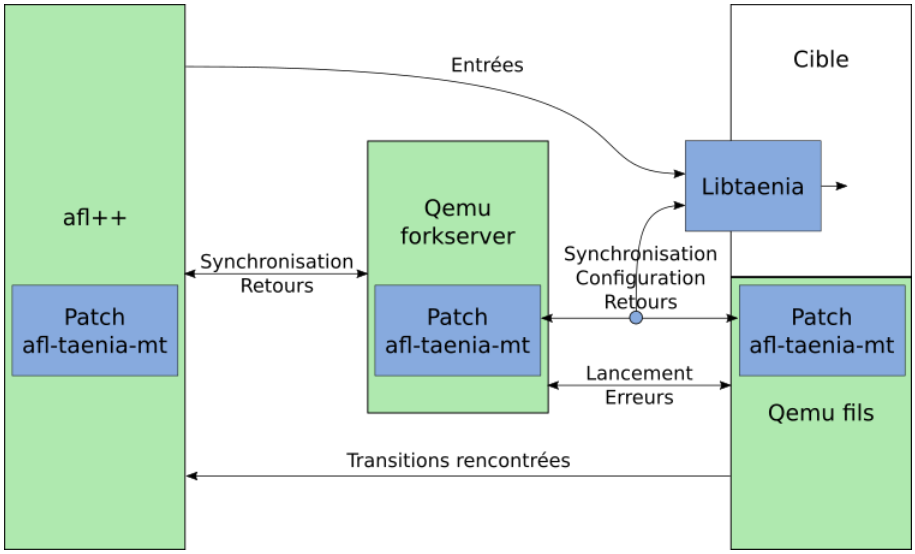

Fig. 7. Architecture d'afl-taenia-mt

Décomposition Le tableau 1 fait le lien entre les fonctionnalités et les éléments associés dans notre solution. Un fichier de configuration est utilisé

Fonctionnalité	patch d'afl++	patch du forkserver	patch de qemu fils	libtaenia	hooks	Archivage des paquets
Fuzzing en mémoire		x		x		
Mode stateful		x		x		x
Suivi de bib. externes			x	x		
Rejeu		x		x		x
Suivi de threads		x	x	x	x	
Suivi de blocs	x		x			

Tableau 1. Décomposition de l'architecture de la solution

pour configurer l'ensemble de ces mécanismes. Ces fonctionnalités seront décrites dans ce qui suit.

2.2 *Fuzzing* en mémoire

Afl ne peut *fuzzer* que des entrées explicites de la cible (stdin, fichier, ligne de commande). Pour *fuzzer* une fonction précise, il est ainsi nécessaire d'exécuter tout le code précédant un appel à cette fonction.

Pour pouvoir directement *fuzzer* des fonctions internes, identifiées comme intéressantes par rétro-ingéniérie, nous utilisons notre bibliothèque injectée libtaenia que nous exécutons dans un thread dédié. Ce dernier exécute en boucle les instructions suivantes : attendre une entrée d'afl, puis exécuter la fonction cible avec l'entrée reçue passée en paramètre.

La fonction cible est fournie à libtaenia par l'utilisateur par le biais du fichier de configuration, sous la forme de son symbole ou de son adresse. libteania se charge alors de récupérer le pointeur vers la bonne fonction pour être en mesure de l'exécuter.

Cela nécessite néanmoins que la fonction cible ait des paramètres simples, aisément rejouables, dont un buffer.

***Fuzzing* sans redémarrage** Dans son fonctionnement standard, afl redémarre le programme cible après chaque entrée fournie. Lorsqu'il utilise le forkserver, ce qui est le cas lorsqu'il est utilisé avec qemu, afl fait un *snapshot* du binaire cible sur son point d'entrée. Pour redémarrer le programme, il reprend ce *snapshot* (*fork*).

Lorsqu'il s'agit de *fuzzer* un processus dans un contexte dans lequel les processus sont interdépendants et communicants, cette méthode atteint ses limites. L'écosystème de processus risque de ne pas supporter qu'un de ses membres soit redémarré continuellement, même après sa phase d'initialisation. Par exemple, un autre processus peut envoyer des *heartbeats* régulièrement au processus cible et les processus peuvent dépendre les uns des autres.

Le *fuzzing* en mémoire permet de résoudre ce problème. Une fois le processus cible initialisé, le thread dédié à libtaenia exécute la fonction cible en boucle comme décrit précédemment. Pendant ce temps, les autres threads de la cible sont libres de continuer à échanger avec le reste de l'architecture, assurant la stabilité de l'ensemble.

Cela permet de gagner en vitesse de *fuzzing* par rapport au mode standard d'afl : il n'est plus nécessaire de redémarrer la cible à chaque exécution.

A noter : afl++ permet (option "forkserver décalé") de décaler l'adresse du *snapshot* précédement évoqué. Ainsi, il est possible de relancer le programme après une phase de réinitialisation longue et inutile pour le *fuzzing*, nous permettant donc de gagner en vitesse de *fuzzing*.

Dans le cas où la fonction cible impacte l'état global du processus, *fuzzer* en continu modifie cet état au fur et à mesure de la session, ce qui permet d'atteindre du code qu'afl ne peut pas atteindre nativement.

Cependant, cela a également un inconvénient : l'état étant modifié au fur et à mesure, les corruptions s'accumulent et peuvent mener à une instabilité de la cible, voire de l'écosystème. Ce problème est adressé par le mode stateful, détaillé plus tard dans cet article.

Suivi des bibliothèques externes La fonction ciblée peut se situer dans une bibliothèque externe. Dans son fonctionnement classique, afl ne suit pas le code situé en dehors du binaire cible. Il est possible de le modifier assez simplement pour qu'il adopte le comportement opposé, à savoir qu'il suive tout le code exécuté, et donc toutes les bibliothèques importées. Ce n'est cependant pas toujours intéressant, en particulier dans le cas de la libc. Par ailleurs, afl ne dispose pas d'assez d'informations sur la cartographie en mémoire des bibliothèques importés pour être en mesure de sélectionner celles que l'on souhaite suivre.

Puisque nous disposons de libtaenia dans la mémoire du processus, nous sommes en revanche en mesure d'obtenir cette information. Nous avons donc rajouté une fonctionnalité de suivi du code des bibliothèques externes, qui fonctionne comme suit :

— L'utilisateur fournit, via un fichier de configuration, un ensemble de bibliothèques l'intéressant.
— Libtaenia, étant dans le processus cible, récupère la cartographie mémoire de ces bibliothèques.
— Il informe ensuite afl-qemu des segments mémoires contenant le code des bibliothèques suivies par le biais d'une mémoire partagée.
— Lorsqu'une sonde est déclenchée, le code de qemu fils patché par nos soins vérifie que l'adresse courante est située dans un des segments suivis avant de marquer le chemin.

Du point de vue d'afl, le code des bibliothèques suivies devient alors indissociable du code du binaire.

2.3 Suivi de threads

Notre objectif est que les sondes de Qemu ne se déclenchent que lorsqu'elles sont rencontrées par des threads "utiles". Pour cela, nous marquons comme *suivis* les threads manipulants des données issues de celle fournie par afl. Ainsi les blocs de base exécutés par les threads qui ne touchent pas cette entrée ne sont pas considérés.

Propagation Pour propager l'attribut *suivi* parmi les threads, nous utilisons les données échangées entre les threads comme marqueurs de

contamination. Pour ce faire, nous modifions les fonctions de communication inter-threads participant au transfert des données issues des entrées. Ces fonctions sont de deux types :

— *recv()* : le thread exécutant cette fonction reçoit (ou lit en mémoire) une donnée en provenance d'un autre thread.
— *send()* : le thread exécutant cette fonction envoie (ou écrit en mémoire) une donnée vers un autre thread.

Le *suivi* commence avec le thread de libtaenia qui fournit la donnée initiale. Il est donc *suivi* par défaut. Ensuite, lorsqu'un thread envoie (écrit) une donnée :

— Si le thread est *suivi*, la donnée devient *suivie*. Cette donnée peut être la donnée d'origine ou une dérivation de celle-ci.

Inversement, lorsqu'un thread reçoit (lit) une donnée.

— Si la donnée est *suivie*, la donnée devient *non suivie*, le thread devient *suivi*. La donnée est consommée par le thread (voir les hypothèses).
— Si la donnée est *non suivie*, le thread devient *non suivi*. Nous considérons que le thread travaille sur des données ne nous intéressant pas.

Nous avons fait deux hypothèses qui explique la propagation précédente :

— Une donnée ne peut être lue qu'une seule fois. Cela permet d'éviter une explosion de la pile des données à suivre.
— Lorsqu'un thread lit une nouvelle donnée, nous considérons qu'il a fini de traiter la donnée précédente. Cela permet d'obtenir une condition de fin à notre suivi de chemin.

Cette condition de fin est essentielle. Libtaenia attend qu'il n'y ait plus ni donnée, ni thread suivi avant de demander à afl une nouvelle donnée. Sans cette condition, nous serions forcés d'utiliser un minuteur, ce qui ruinerait la vitesse de *fuzzing*.

Pour réaliser ces hypothèses, il faut *hooker* de manière très précise les fonctions *recv()* et *send()* impliquées dans les communications inter-thread. En effet, si une fonction *recv()* est oubliée, l'algorithme ne finira pas ; si une fonction *send()* est oubliée, la surface d'attaque en sera réduite.

La figure 8 présente en vert le code suivi par afl-taenia-mt dans une architecture composée de plusieurs threads : l'appel par le thread libtaenia de la fonction cible (*send()*), puis le traitement par le thread Parser [5]. Une fois l'entrée entièrement traitée, libtaenia reprend la main et envoie

5. Pour éviter les interférences, le code propre à libtaenia n'est pas suivi grâce au mécanisme permettant de suivre (ou non) le code des bibliothèques externes, cela n'est pas représenté sur cette figure.

une nouvelle entrée. Nous observons que le code suivi est épuré des blocs inintéressants [6].

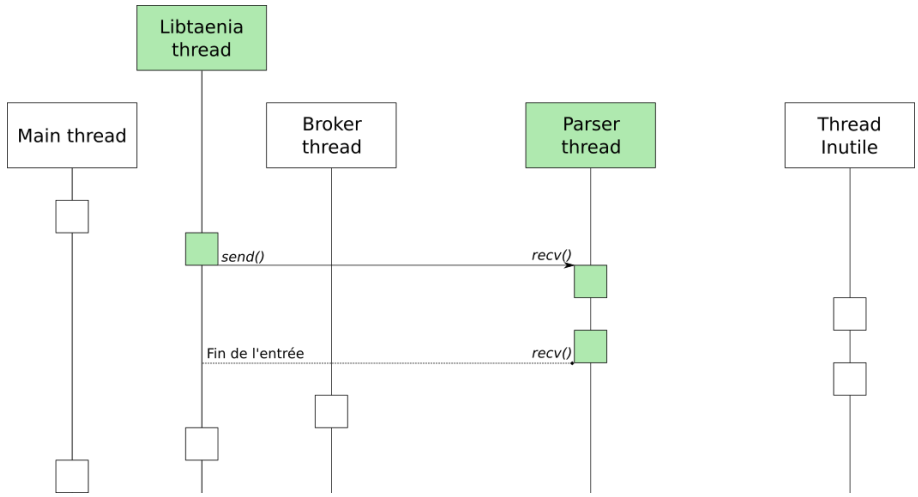

Fig. 8. Séquence d'exécution d'afl-taenia-mt avec suivi de thread

Hooks La propagation décrite précédemment nécessite d'instrumenter les fonctions *send()* et *recv()* qui permettent d'échanger des données intéressantes entre plusieurs threads. Pour cela nous utilisons le fait que la libtaenia soit pré-chargée dans le binaire cible pour *hooker* ces fonctions.

Nous souhaitons *hooker* à un endroit où la donnée est récupérable, soit dans une fonction dédiée de l'API, soit autour d'un memcpy() ou d'un socket.read() qui manipuleraient cette donnée.

Nous utilisons dans notre code plusieurs types de *hooks* selon la situation.

Hook par saut L'objectif est d'ajouter l'exécution d'un ensemble d'instructions à un endroit arbitraire du binaire cible. Pour ce faire, en supposant que l'emplacement idéal ait été identifié, on écrase quelques instructions à cet endroit par un saut vers notre code. Il faut prévoir de réécrire les octets écrasés dans notre fonction *hook()* afin de ne pas corrompre la fonction

6. Le code du Broker n'est pas suivi, son comportement est contourné par l'appel direct depuis libtaenia.

initiale. Enfin, une fois notre code exécuté, nous sautons vers le binaire cible juste après le saut initial.

Dans le cas général, nous sautons au début de la fonction, pour y exécuter notre *hook*. Éventuellement, nous en profitons pour écraser l'adresse de retour de cette fonction, pour exécuter du code à la fin.

La figure 9 schématise cela pour une architecture x86.

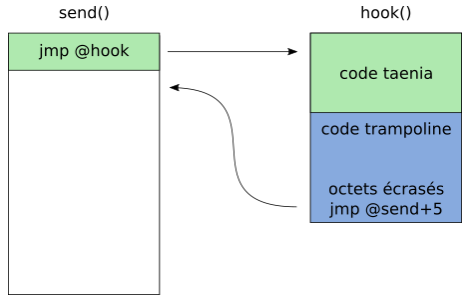

Fig. 9. Hook par jmp de la fonction *send()*

Il est également possible de hooker au milieu d'une fonction, sur une instruction particulière (par exemple le *mov* d'une donnée intéressante). Cela nécessite cependant d'être très précis, de sauvegarder puis restaurer intégralement le contexte original.

Hook par call L'objectif est de remplacer l'exécution d'une fonction du binaire cible par une fonction de libtaenia. Pour cela, en supposant que la fonction idéale ait été identifiée, on écrase l'adresse de cette fonction avec l'adresse d'une de nos fonctions. Cette dernière effectue le traitement que nous souhaitons et appellera la fonction initiale. Le retour est assuré par le ret. La figure 10 schématise cela pour une architecture x86.

Hook par LD_PRELOAD L'objectif est de remplacer une fonction importée par une de nos fonctions. Le mécanisme LD_PRELOAD du linker dynamique de Linux permet de le faire simplement. La différence avec le *hook* par appel de fonction réside dans le fait que ce dernier modifie un seul appel de fonction précis alors que LD_PRELOAD écrase la fonction dans la table d'import, donc pour tous les appels.

Appels indirects Notre solution nécessite une fonction avec une signature simple et contenant un buffer dans lequel placer nos données. Cela

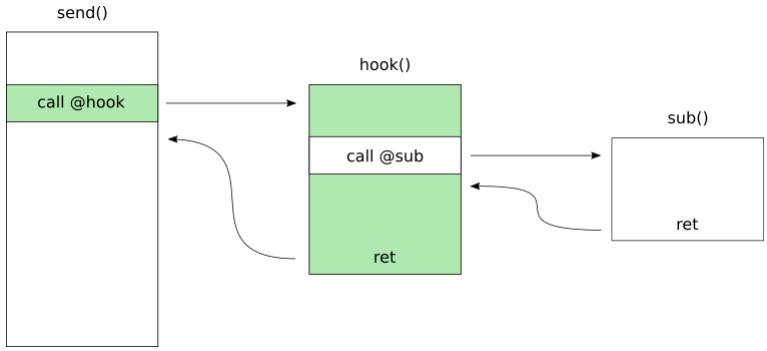

Fig. 10. Hook par call de la fonction *send()*

n'est malheureusement pas toujours le cas, il peut arriver que le traitement se fasse sans fonction explicite.

Considérons par exemple le code suivant :

```
while (1) {
    recvfrom(sockfd, buf, len, flags);
    /* Fait des trucs */
}
```

Notre solution face à ce problème est d'écraser la fonction *recvfrom* par une fonction qui attend une entrée d'afl-taenia-mt, la copie dans le buffer buf et termine.

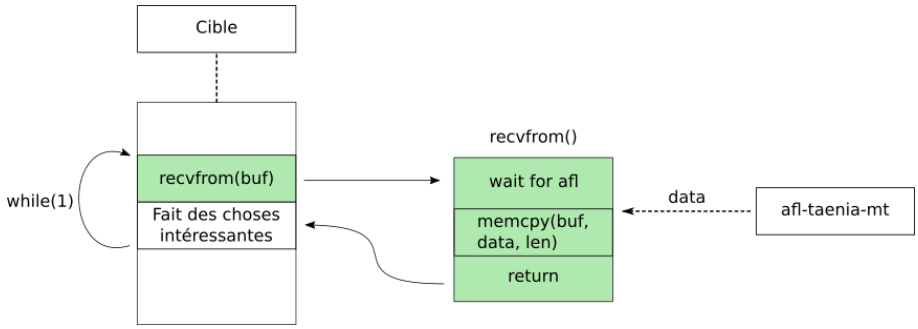

Fig. 11. Illustration de l'appel indirect

Illustration de l'impact sur les performances du *fuzzing* Nous reprenons l'exemple précédent dans son architecture de base. afl-taenia-mt permet de *fuzzer* en mémoire, nous *fuzzons* donc directement la fonction de réception du Broker (qui pourrait être appelée depuis le réseau, la ligne de commande ...).

La figure 12 montre les résultats de cette opération. Nous retrouvons des performances similaires à celle de la figure 4. Un nombre équivalent de chemins a été parcouru. Le nombre d'exécutions est également équivalent, grâce au fonctionnement en mémoire. Cependant, ici le *crash stateful* a été trouvé car afl-taenia-mt ne relance pas la cible à chaque exécution.

La stabilité n'est pas de 100% car le dernier chemin (qui est *stateful*) n'est pas retrouvé lorsque l'entrée associée est rejouée, car celle-ci est rejouée après le *crash* et donc le processus n'est plus dans le bon état.

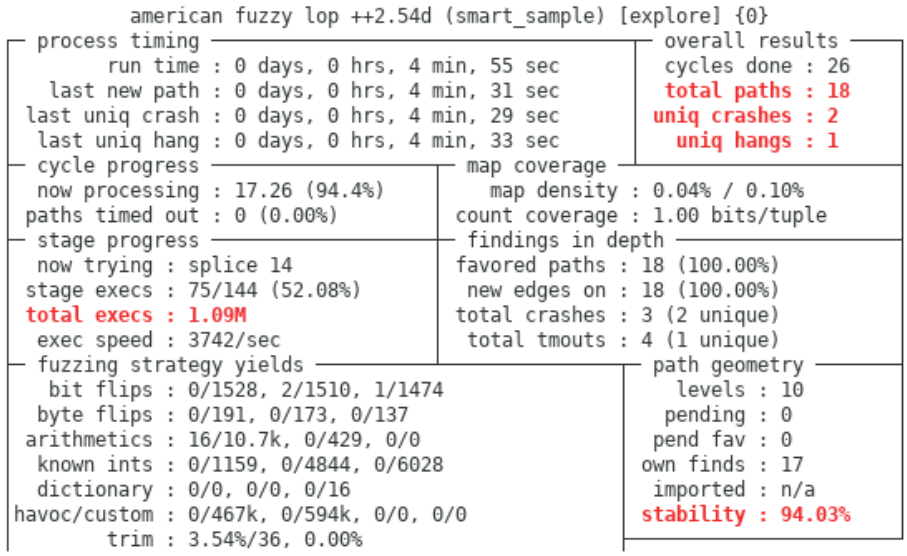

Fig. 12. Performances d'afl-taenia-mt sur une architecture multi-threadée

2.4 Mode *stateful*

Dans son fonctionnement normal, afl ne sauvegarde que l'entrée qu'il identifie comme responsable d'un comportement intéressant (*hang* ou *crash*). Comme nous l'avons vu, lors du *fuzzing* depuis la mémoire, un *crash* peut survenir du fait de la combinaison de plusieurs entrées. Le

risque est donc d'obtenir un *crash*, mais pas toutes les entrées nécessaires pour le rejouer.

C'est la raison pour laquelle nous avons créé le mode *stateful*. Il s'agit d'un compromis entre les deux approches qui fonctionne comme suit :

- L'utilisateur choisit le nombre d'entrées maximum à envoyer.
- Chaque entrée est exécutée et sauvegardée dans une archive.
- Si un comportement intéressant est relevé, l'ensemble des entrées sauvegardées y sera associé.
- Si le nombre d'entrées spécifié par l'utilisateur est atteint, le programme, ou l'écosystème, est redémarré et les sauvegardes obsolètes sont supprimées.

Ainsi, ce mode permet de dérouler le *fuzzing* par "session". Une session est composé du lancement du processus cible, d'une série de tests, puis de l'arrêt de ce processus ou de l'écosystème.

Lorsqu'un *crash* survient, s'il est non *stateful*, il est traité classiquement en analysant l'effet de la dernière entrée envoyée. Lorsqu'il est *stateful*, l'utilisateur détermine dans un premier temps le sous-ensemble minimaliste d'entrées provoquant le *crash*, puis investigue leurs effets.

Le paramètre principal du mode *stateful* est le nombre d'entrées à exécuter dans une session de *fuzzing*. Il est à la charge de l'utilisateur de configurer ce nombre sachant qu'un nombre élevé implique une vitesse de *fuzzing* accrue, mais une plus grande consommation de mémoire ; également plus de bugs *statefuls*, mais plus de difficultés à les déboguer.

Suivi de processus Puisque le processus cible est en dialogue avec d'autres processus de son écosystème, il est possible que notre entrée soit passée, au moins en partie, à un autre processus dans lequel elle peut potentiellement provoquer un *crash*. Si tel est le cas, nous souhaitons en être informés et conserver les entrées responsables pour investigations futures.

Nous avons donc intégré un mode de suivi de processus au mode *stateful*. Il consiste simplement à vérifier l'état d'une liste de processus fournie par l'utilisateur à la fin d'une session de *fuzzing*. Ainsi, si un des processus est identifié comme mort, l'ensemble des entrées de la session sont sauvegardées. On peut également faire le choix d'arrêter le *fuzzing* si le ou les processus morts sont essentiels au bon fonctionnement de l'écosystème.

L'automatisation de la réinitialisation de l'écosystème doit être assurée par un outil externe.

Il est également possible d'ajouter un test de bonne santé à la fin d'une session. Par exemple : envoyer un ping applicatif au processus cible pour identifier s'il fonctionne toujours de manière nominale. Cela peut permettre de détecter des erreurs "stables", par exemple le processus cible répond toujours la même chose, quelle que soit l'entrée fournie.

Capacités de rejeu Être en mesure de rejouer une entrée intéressante est une composante importante du processus de *fuzzing*. Cette capacité permet d'investiguer sur le comportement observé, valider sa réplicabilité et comprendre son origine. Dans le cas d'afl, les conditions de *fuzzing* sont suffisamment simples pour que la sauvegarde de l'entrée ayant provoqué un comportement intéressant suffise pour permettre le rejeu.

Dans notre cas, il est nécessaire de fournir une fonctionnalité à part entière pour plusieurs raisons :

— Le rejeu doit être effectué depuis la mémoire du processus ciblé.
— Dans le cas d'un bug *stateful*, il est nécessaire de rejouer les entrées sauvegardées dans un ordre précis pour répliquer le bug.

Nous avons donc développé un mode de rejeu qui nous permet de rejouer une entrée ou un ensemble d'entrée sauvegardées par le mode *stateful*.

2.5 Optimisations diverses

Dans le cadre de notre projet, plusieurs modifications ont été faites pour accélérer la vitesse de *fuzzing* et donc augmenter les performances du *fuzzer* de manière générale.

ramfs En premier lieu, nous avons automatisé la mise en place d'un ramfs pour contenir le projet de *fuzzing*. Cela nous a donné un gain d'environ 25% d'exécutions par seconde sur un exemple de test.

Taille du buffer d'entrée Ensuite, nous avons limité la taille du buffer d'entrée d'afl. En effet, afl stocke ses entrées dans un buffer de 1 Mo. Il peut donc générer des entrées très grandes, surtout lorsqu'il utilise ses heuristiques non déterministes ('havoc' et 'splice' [7]). Lorsqu'il fournit une nouvelle entrée à la cible, celle-ci peut être tronquée par la fonction ciblée (par exemple *read()* et *recv()* prennent en paramètre une taille qui tronque leur entrée). Mais si cette nouvelle entrée mène à un nouveau chemin, elle sera sauvegardée par afl en intégralité. Par la suite afl manipulera

7. Voir la page de l'auteur [5] pour une explication ce ces heuristiques

une entrée de grande taille, fera des mutations sur des portions non utiles et donc perdra en efficacité. Nous avons donc modifié afl pour qu'il ne sauvegarde que l'entrée tronquée à une taille définie par l'utilisateur. Sur notre exemple de test, cela a permis de gagner 32% en vitesse d'exécutions, mais cela est à nuancer car l'entrée avait une taille de 20 octets.

Suivi de parcours du graphe de flot de contrôle Pour suivre le graphe de flot de contrôle parcouru par les différentes entrées et donc déboguer correctement notre preuve de concept, nous avons modifié le patch d'afl dans qemu pour qu'il trace les blocs de base parcourus. Nous avons ensuite développé un script (python/idapython) qui prend ces traces en entrée pour reconstruire le graphe de flot de contrôle en format dot. Il est également capable de colorer les blocs dans IDAPro.

La figure 13 présente ces informations sous forme de graphe dot. Cela forme un graphe avec comme noeud les adresses des sondes, regroupées par fonction et par thread. Les transitions sont labellisées de leur numéro. Les blocs verts sont ceux qui ont été découverts par l'exécution de cette entrée.

La figure 14 présente la représentation de ces informations sur IDAPro. Les blocs de base et fonctions sont coloriés en bleu lorsqu'ils ont été exécutés, en vert lorsque leurs exécutions a été ajoutée par cette entrée, en orange s'ils sont responsables d'un *hang* et en rouge pour un *crash*.

3 Conclusion

Nous avons souhaité *fuzzer* une cible en boîte noire, complexe et multi-threadée. L'outil que nous avons choisi pour cela, afl, a plusieurs limites face à ce problème. Nous avons présenté dans cet article des solutions pour contourner ces limites, que nous avons mis en place dans une preuve de concept. Voici les problèmes rencontrés et les solutions proposées :

1. Le processus cible prend beaucoup de temps à s'initialiser.
   — *Fuzzer* en mémoire nous permet de ne pas redémarrer le processus à chaque itération.
2. Le processus cible fait partie d'un écosystème de processus, réinitialiser le processus cible sans redémarrer l'écosystème mène à des instabilités. Certaines entrées peuvent provoquer des erreurs dans les autres processus de l'écosystème.
   — Le fait de ne pas redémarrer le processus à chaque itération permet d'éviter ce problème. Cependant, la corruption peut atteindre l'écosystème. Dans ce cas, il faut privilégier l'usage

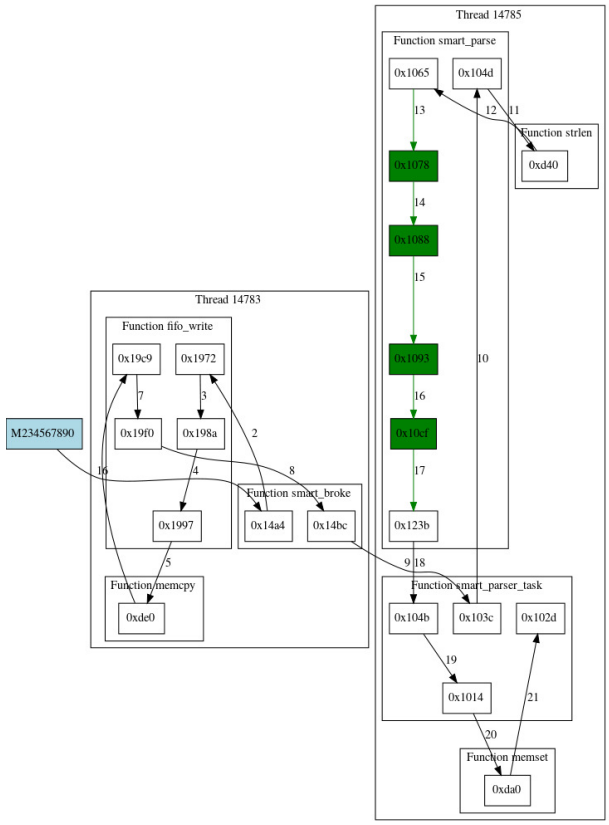

Fig. 13. Suivi d'une entrée d'afl sous dot

du mode *stateful*, avec un nombre maximum d'entrées bien configuré et un script de redémarrage de la plateforme adéquat.
— Le mode *stateful* nous permet de tester entre chaque session de fuzzing l'état des autres processus. Si un bug est détecté, nous avons alors la suite d'entrées l'ayant (probablement) causé.

3. Le code ciblé est enfoui dans le binaire cible.
   — Le *fuzzing* en mémoire permet de cibler la fonction de notre choix.
4. Le code ciblé peut être situé dans une bibliothèque externe.
   — Nous avons modifié le code d'afl pour qu'il puisse suivre les bibliothèques de notre choix.
5. Une entrée peut être manipulée par plusieurs threads.

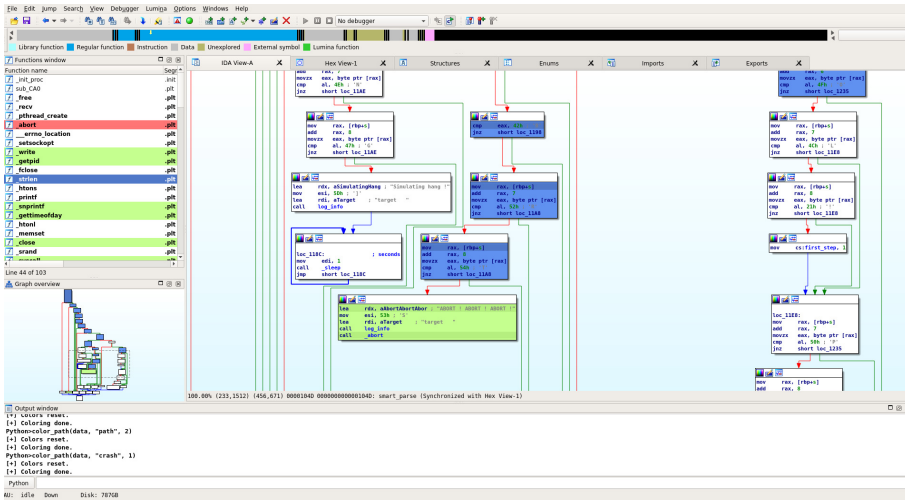

Fig. 14. Suivi d'une entrée d'afl sous IDAPro

— Le suivi de threads permet de suivre convenablement cette entrée et de démêler les threads.

6. Le processus cible possède des threads ne manipulant pas les entrées.
   — Le suivi de threads permet de les ignorer.

Cependant ces solutions nécessitent de la rétro-ingéniérie pour identifier la fonction cible, les fonctions de *send()* et *recv() hookées* ainsi que le développement des *hooks*.

Nous espérons que cela pourra être utile à la communauté ou, qu'à défaut, cet article aura donné quelques idées au lecteur.

Pérennité de la preuve de concept Nous souhaitions que notre preuve de concept ait un impact minimal sur afl++ dans le but de faciliter sa maintenance. Pour cela nous avons rendu la partie taenia aussi autonome que possible, la configuration se faisant notamment par un fichier à part.

Les modifications du coeur d'afl++ servent à insérer des logs et à faciliter le suivi des chemins. Elles ne sont pas requises au fonctionnement de la solution.

En revanche, les sources du qemu_mode ont été presque intégralement réécrites. En effet, nous avons abandonné le fonctionnement de base d'afl qui est de forker pour chaque nouvelle entrée, et nous avons modifié l'installation des sondes. Les évolutions futures provenant d'afl++ seront donc difficiles à intégrer si elles impactent ce qemu_mode.

Remerciements Nous souhaitons remercier Aymeric Leperoux pour sa participation au développement d'afl-taenia incluant déjà le *fuzzing* en mémoire. Un grand merci à toute l'équipe pour son aide technique. Merci aux relecteurs pour leurs retours, leurs aides et les idées qu'ils nous ont apportés.

Références

1. Google. github afl. `https://github.com/google/AFL`.
2. Google. honggfuzz. `https://github.com/google/honggfuzz`.
3. qemu, Fabrice Bellard. plateformes supportées par qemu. `https://wiki.qemu.org/Documentation/Platforms`.
4. qemu, Fabrice Bellard. qemu. `https://www.qemu.org`.
5. Michal Zalewski. american fuzzy lop status screen. `lcamtuf.coredump.cx/afl/status\_screen.txt`.