GUSTAVE : Fuzz It Like It's App

(feat. QEMU & AFL)

Stéphane Duverger et Anaïs Gantet
stephane.duverger@airbus.com
anais.gantet@airbus.com

Airbus

Résumé. Ces dernières années, les technologies de *fuzzing* avancé comme le *coverage-guided fuzzing* ont considérablement évolué et permis de mettre au jour de nombreuses vulnérabilités, notamment dans des applications.

Cet article a pour but de partager le travail réalisé sur GUSTAVE, un outil alliant *fuzzing* avancé et plateforme de simulation dans le but d'aider à rechercher des vulnérabilités dans des systèmes d'exploitation embarqués.

L'idée clé de notre plateforme est de tirer parti de la puissance de technologies existantes, que ce soit pour le *fuzzing* grâce à AFL ou la simulation grâce à QEMU, sans y apporter de modifications significatives. Le but est de *fuzzer* un noyau en laissant penser à AFL qu'il s'agit d'une simple application.

Nous décrivons les enjeux qui gravitent autour de la conception de GUSTAVE, son positionnement à l'état de l'art, ainsi que sa mise en œuvre sur un exemple, le système d'exploitation minimaliste `POK`. Enfin nous exposons les premiers résultats associés, ses performances, limitations et évolutions envisagées.

1 Introduction

1.1 Cible : systèmes d'exploitation embarqués

La famille des noyaux de systèmes d'exploitation embarqués, en contexte industriel, qu'ils soient ouverts ou propriétaires, est extrêmement hétérogène. Elle se compose de noyaux inspirés parfois du monde UNIX, monolithiques ou non, mais également d'architectures logicielles plus originales. Cet écosystème est assez différent du triptyque *Windows, MacOS, Linux* pour lequel de nombreuses analyses de vulnérabilités ont été réalisées et continuent de l'être.

Le fait qu'ils soient dits embarqués est intéressant au moins en deux points. D'une part, ils sont par nature assez figés et réduits à l'essentiel dans leur déploiement : moins dynamiques, souvent déterministes aussi bien du point de vue de l'espace que du temps, pas de code mort. D'autre part, cette particularité ne les rend pas moins complexes et leur surface

d'attaque peut être conséquente [1] principalement car ils implémentent des couches logicielles spécifiques [2] que l'on rencontre très peu dans des environnements IT plus classiques.

Ce dernier point nous laisse penser qu'il serait intéressant de concevoir un environnement de tests de sécurité automatisé, aisément adaptable. S'agissant de systèmes parfois inconnus, il nous est apparu naturel d'imaginer réaliser des campagnes de *fuzzing* sur des couches logicielles potentiellement peu explorées d'un point de vue sécurité.

1.2 Méthode retenue : *coverage-guided fuzzing* avec AFL

Fuzzer un noyau de système d'exploitation, en soi, n'a rien de réellement novateur. Comme mentionné précédemment, des systèmes très répandus ont vu leur implémentation mise à rude épreuve par divers outils plus ou moins avancés (voir section 2), pertinents et efficaces.

Toutefois, les technologies de *fuzzing* ont considérablement évolué ces dernières années, notamment avec la mise à disposition d'outils implémentant ce que l'on nomme le *coverage-guided fuzzing*, avec son représentant probablement le plus emblématique : *American Fuzzy Lop* (AFL) [15].

Le *coverage-guided fuzzing* repose sur l'instrumentation de la cible à *fuzzer*, permettant de mesurer les chemins d'exécution qu'elle peut prendre lorsqu'elle traite des données générées par le *fuzzer*. Ce dernier est donc capable d'analyser la quantité de code couverte par sa stratégie de *fuzzing*, lui permettant de l'adapter le cas échéant afin d'en optimiser les résultats escomptés (généralement un crash de la cible). AFL ajoute au *coverage-guided fuzzing* une forme de surveillance comportementale de la cible (*timeout, segfault, etc.*).

AFL dispose à ce jour de résultats conséquents si l'on se réfère à la section *bug-o-rama trophy case* de la page d'accueil du projet. La quasi-totalité des vulnérabilités découvertes se situe dans des programmes utilisateurs ou des bibliothèques, principalement concernant des opérations de *parsing* de fichiers. Appliquer AFL au cas du *fuzzing* d'un noyau de système d'exploitation constitue en soi un premier challenge, surtout pour des systèmes embarqués peu dynamiques.

L'idée conductrice de cette étude a été d'essayer de profiter des capacités de *fuzzing* d'AFL pour rechercher des vulnérabilités dans ces noyaux, et plus particulièrement pour détecter si les mécanismes de protection

1. Ce qui évidemment justifie une analyse approfondie de leur sécurité.
2. Nous pensons par exemple aux normes ARINC.

mémoire mis en place pourraient être compromis depuis un programme moins privilégié.

AFL agit en deux étapes. La première est celle de l'instrumentation, c'est-à-dire de la préparation de la cible à *fuzzer*. Le but de cette instrumentation est d'injecter un *shim*, c'est-à-dire un petit ensemble d'instructions, dans chaque *basic block* de la cible, afin de signaler les chemins d'exécution empruntés par la cible durant son exécution et d'enregistrer au fur et à mesure la couverture de code parcourue dans une *trace bitmap*.

La seconde étape est le processus de *fuzzing* à proprement parler. AFL génère une entrée, lance la cible (*fork*/*execve*) et surveille son exécution. L'analyse comportementale tient à la fois de l'analyse de la *trace bitmap* et des signaux reçus par le programme (`ALRM,TERM,SEGV,ABRT,...`) permettant de classifier son exécution (faute ou pas). AFL va générer, de manière efficace, de nouvelles entrées à l'aide de la trace d'exécution précédente et d'algorithmes évolutionnistes que nous ne détaillerons pas ici.

Une optimisation intéressante du processus de *fuzzing* proposée par AFL consiste en l'injection d'un *fork server* dans la cible en complément des *shim* mettant à jour la *trace bitmap*. Il permet de n'effectuer l'initialisation du programme qu'une seule fois et de recommencer le *fuzzing* directement à partir de l'endroit choisi. Ce *fork server* se synchronise avec AFL, et s'occupe de la création de processus fils pour chaque nouvelle entrée à tester. La figure 1 décrit ce procédé, que nous réutiliserons par la suite.

1.3 Enjeux du *fuzzing* de systèmes d'exploitation

Dans le cas du *fuzzing* d'un programme utilisateur, la détection de crash est relativement simple pour AFL car le système d'exploitation dispose d'un ensemble de mécanismes (mise en place d'isolation de l'espace d'adressage, de droits particuliers, *etc.*), jouant en quelque sorte le rôle de *garde-fou* mémoire, permettant à AFL de savoir si un programme utilisateur s'est bien ou mal terminé.

Tenter d'utiliser AFL pour *fuzzer* un noyau soulève deux challenges majeurs ; d'une part, AFL n'est pas initialement prévu pour s'interfacer et cibler un système d'exploitation complet ; d'autre part, la détection de crash n'est plus aussi triviale, puisqu'il n'y a plus de *garde-fou* dans le cas où le noyau aurait passé outre une ségrégation mémoire.

Le but de notre travail a été de proposer un outil prenant en compte les enjeux évoqués ci-dessus et permettant d'interfacer AFL à n'importe quel système d'exploitation. Notre approche s'attache tout particulièrement à réutiliser autant que possible des technologies existantes, fiables et éprouvées, mais également à les modifier le moins possible. Nous ne souhaitions

pas changer le *design* d'AFL, et considérons qu'il est tout à fait possible qu'il *fuzze* un noyau comme un simple programme utilisateur, notamment grâce à la plateforme de simulation générique et libre : QEMU [8].

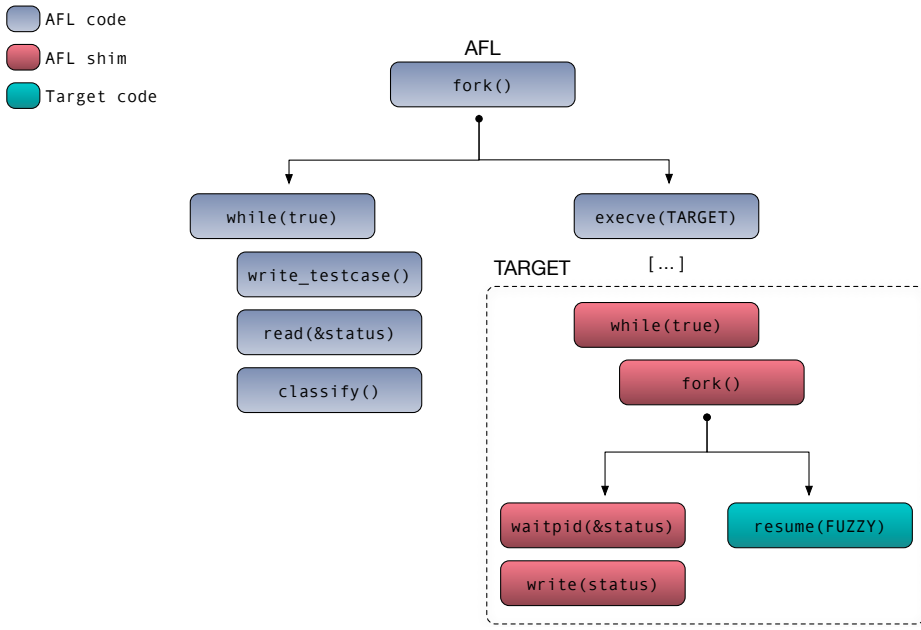

Fig. 1. AFL *fork server*.

2 État de l'art du *fuzzing* de systèmes d'exploitation

Tirer parti des solutions existantes et disponibles pour répondre à notre besoin était les maîtres-mots de notre approche. Notre liste de critères est courte mais somme toute exigeante. Dans l'idéal, nous souhaitions une approche :

— adaptable à n'importe quelle architecture matérielle ;
— profitant du potentiel du *fuzzer* d'AFL sans le modifier ;
— ne nécessitant pas de développement additionnel dans l'OS cible ;
— mise en œuvre par un outil libre, ouvert, facilement maintenable.

Comme énoncé précédemment, le *fuzzing* de noyaux de systèmes d'exploitation n'est pas nouveau et de nombreuses solutions sont disponibles [5, 7, 12, 25, 26]. Nous évoquons ci-après les solutions qui nous ont paru être les plus proches de notre besoin et semblaient nécessiter le moins d'effort possible d'adaptation : TriforceAFL, kAFL et afl-unicorn.

2.1 TriforceAFL

TriforceAFL [20] est la solution qui se rapproche le plus de nos choix technologiques : AFL et QEMU. Ce projet propose une extension d'AFL et de QEMU afin de permettre de *fuzzer* des systèmes complets. Bien que très prometteur de prime abord, leurs modifications sont assez conséquentes. C'est exactement ce que nous souhaitons éviter car cela oblige à maintenir des modifications du code source de nos deux principaux composants open-source. Sans trop entrer dans les détails, leur approche consiste en la modification du moteur d'émulation de QEMU [3] et l'ajout d'une instruction spécifique, dans le but d'effectuer l'instrumentation de la cible sans phase de recompilation et de transmettre à AFL les informations de couverture de code.

Nous y voyons au moins deux inconvénients majeurs. Le premier est qu'il empêche un éventuel usage de la virtualisation matérielle afin d'accélérer l'exécution de la cible. QEMU supporte différents *backend* d'accélération matérielle [1] dont l'intérêt est de permettre l'exécution du code cible directement sur le microprocesseur hôte [4] sans passer par un moteur d'émulation. Si l'instrumentation est effectuée dans le moteur d'émulation, l'usage de la virtualisation matérielle empêchera toute instrumentation. Le second est qu'il est plus difficile d'isoler des portions de la cible à inspecter. L'intégration de l'instrumentation dans la plateforme de *fuzzing* est, de notre point de vue, une limitation. Nous préférons être en mesure de préparer, en amont, des cibles instrumentées à des niveaux variables, et parfois à l'aide de technologies différentes de celles proposées avec le *fuzzer* d'AFL (recompilation, instrumentation binaire, etc.).

Finalement leur approche nécessite également le développement d'un pilote au sein du noyau cible, ce qui, de notre point de vue, est rédhibitoire (développement spécifique, caractère intrusif).

Nous démontrons dans la section 3 que nous pouvons atteindre un niveau de fonctionnalités similaire, sans modifier AFL, ni les *internals* de QEMU, ni nécessiter le moindre développement au sein de la cible.

2.2 kAFL

Le projet kAFL [22] était de loin le projet le plus prometteur. Il s'appuie sur des méthodes de *fuzzing* inspirées d'AFL, sur de la virtualisation matérielle et sur une extension spécifique des processeurs Intel x86 : *Processor Trace* (Intel PT [11]).

3. Tiny Code Generator (TCG).
4. Cette accélération requiert une architecture identique entre l'hôte et la cible.

Leur approche est indépendante du système d'exploitation, bien que nécessitant tout de même le développement d'une petite glu au sein du noyau. Malheureusement, leur solution est évidemment dédiée à l'architecture Intel x86. Ceci n'est pas acceptable dans notre approche ciblant des architectures potentiellement exotiques, ne fournissant pas systématiquement le même niveau de fonctionnalités matérielles (virtualisation, trace d'exécution).

Cela étant, leur implémentation optimisée des algorithmes d'AFL, leurs performances impressionnantes (environ 17k cas de test/sec) et surtout les résultats obtenus (plusieurs CVEs connues retrouvées) permettent de penser que cette approche du *fuzzing* sur des noyaux d'OS peut déclencher des vulnérabilités critiques de façon totalement automatisée.

Il convient de mentionner qu'une limitation de leur étude provient du fait qu'ils ont principalement ciblé des implémentations de pilotes de systèmes de fichiers (NTFS, HFS, EXT4), présentant les données générées par le *fuzzer* de façon brute, comme une image de système de fichiers à monter. Ils couvrent donc une surface relativement restreinte du code de noyaux de systèmes d'exploitation.

2.3 afl-unicorn

L'outil afl-unicorn [17–19], comme son nom l'indique, est fondé sur AFL et Unicorn [21]. Ce dernier est un émulateur de CPU s'appuyant très fortement sur QEMU, mais avec une approche dite plus *instrumentable*, via des *hooks python* à des endroits clés du processus de simulation. Par contre il ne traite que les instructions du CPU et ne propose pas d'émulation de périphériques, ne permettant pas la simulation de systèmes complets.

Le projet semble vouloir permettre de *fuzzer*, via AFL, des portions de programmes quelle que soit leur origine (architecture, application, noyau, firmware), dès l'instant que l'on est en mesure de pouvoir les extraire de leur environnement d'exécution global.

Des travaux très intéressants ont été présentés à ZeroNights 2018 [6]. La cible était un firmware d'une puce Wi-Fi très répandue, duquel ils ont extrait des fonctions de *parsing* qu'ils ont *fuzzées* à l'aide de cet outil. Des vulnérabilités ont été découvertes. Les entrées du *fuzzer* sont constituées, encore une fois, des données brutes directement traitées par la fonction de *parsing*.

Si l'approche semble extrêmement pragmatique (cibler un OS embarqué, trouver des failles et les exploiter), leur méthodologie nécessite par contre un effort assez conséquent de rétro-conception, ainsi que d'exécution en environnement réel afin d'être en mesure d'extraire des éléments

contextuels (tas, pile, variables globales de l'OS) permettant la simulation de la fonction extraite dans l'outil. Si, dans ce cas étudié, il n'est pas utile de simuler dans sa totalité un OS cible, et de se focaliser sur certaines fonctions uniquement, force est de constater qu'il leur est tout de même nécessaire de posséder un contexte d'exécution réel afin de fournir des informations essentielles à l'exécution en environnement simulé.

L'effort consenti à préparer un environnement simulé même dégradé, autorisant l'exécution quasi complète de l'OS cible, nous apparaît plus pertinent et prometteur dans la quantité et la qualité des vulnérabilités qu'il pourrait révéler.

Les performances annoncées par l'auteur du projet sont assez pauvres (40 cas/sec) sur des exemples simples. Cependant les résultats de la présentation de ZeroNights montrent un taux de 500 cas/sec, que nous pensons très largement lié à la nature du code *fuzzé* (une simple fonction de *parsing* isolée).

Enfin, un cas particulier sur l'émulation [5] laisse planer quelques doutes sur les choix de design de leur approche.

2.4 Conclusion

En résumé, les trois outils évoqués ne répondent qu'en partie à notre besoin, puisque soit ils sont dédiés à une architecture matérielle donnée, soit ils ne ciblent qu'une petite partie du système d'exploitation, soit ils nécessitent des modifications beaucoup trop lourdes et intrusives vis-à-vis de l'OS cible (par exemple par l'ajout d'un driver). L'ensemble de ces limitations nous ont poussés à proposer une nouvelle approche de *fuzzing* d'OS embarqués, nommée GUSTAVE.

3 Les concepts de GUSTAVE

3.1 Aperçu global

GUSTAVE s'intercale entre le système d'exploitation cible et AFL : du point de vue d'AFL, il doit faire en sorte de remonter des informations sur le comportement du système d'exploitation (crash, contournement de la ségrégation mémoire, etc.) ; du point de vue du système d'exploitation, il doit traduire les entrées brutes générées par AFL en des programmes utilisateurs faisant appel à des services exposés par le noyau, souvent via des appels système. Toutefois, cette transformation pourrait, selon la cible, aboutir à tout autre chose. Elle n'est qu'une étape du procédé.

5. `https://github.com/Battelle/afl-unicorn/issues/3`

Pour remplir son rôle, GUSTAVE se décompose ainsi :
— un traducteur des entrées AFL (*syscall generator* sur la figure 2) ;
— un environnement de simulation autour du système d'exploitation (*vCPU* sur la figure 2) ;
— la définition de propriétés de sécurité ;
— la levée d'alerte associée en cas de vulnérabilité détectée.

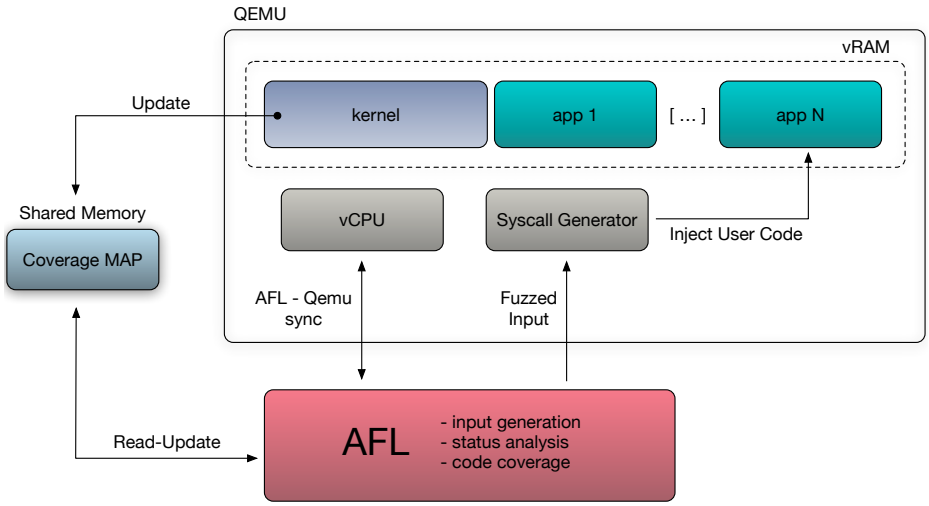

Fig. 2. Architecture générique de GUSTAVE.

Plus de détails sont donnés dans les sous-sections suivantes.

Ainsi, le scénario de *fuzzing* est le suivant : en amont, GUSTAVE prépare avec QEMU un environnement simulé dans lequel le système d'exploitation, au préalable instrumenté, va pouvoir s'exécuter. GUSTAVE met également en place les mesures de contrôle qui lui permettront d'analyser finement le comportement de la cible, en particulier pour vérifier si les propriétés de ségrégation mémoire sont respectées.

AFL commence alors par générer des données brutes (*blob* binaire). GUSTAVE traduit ensuite ces données brutes en un programme utilisateur sous forme de séquence d'appels système avec des arguments arbitraires, provenant des données d'AFL, et injecte le programme utilisateur produit dans le système d'exploitation. GUSTAVE rend la main au programme pour qu'il puisse s'exécuter. À ce stade, AFL enregistre, au fil de l'exécution du test, la couverture de code associée (cible instrumentée). En cas de vulnérabilité détectée, GUSTAVE signale à AFL qu'il y a eu un

comportement anormal afin qu'il l'enregistre comme *crash case*. AFL génère ensuite de nouvelles données brutes et le scénario réitère.

L'originalité de notre approche, en regard de l'état de l'art, réside principalement dans le fait que nous ne modifions pas AFL, permettant ainsi par exemple d'utiliser d'autres implémentations du cœur de *fuzzing* compatibles [3, 13, 14]. De plus, nous proposons une traduction plus ou moins poussée des données d'AFL, et nous élaborons des mécanismes de détection de fautes survenant au sein du noyau de système d'exploitation cible.

3.2 Étape préliminaire : instrumentation de la cible

La mise en œuvre de l'étape d'instrumentation dépend du fait qu'on dispose ou non du code source de la cible. Dans le cas où l'on a le code source, on peut recompiler la cible à l'aide d'un *wrapper* à GCC/LLVM mis à disposition par AFL. Dans le cas contraire, le problème de l'instrumentation peut être résolu soit en adaptant le moteur d'émulation de QEMU (TCG) à l'image du *qemu_mode* proposé par AFL ou du projet TriforceAFL, soit par des méthodes annexes capables d'instrumenter le binaire directement [4, 10, 16, 23, 24].

Nous avons précédemment évoqué certains arguments en défaveur des modifications opérées au niveau du TCG. Cela étant nous concevons que dans certaines situations [6], cette approche puisse être envisagée. Tout dépend de la nature de la cible.

Nous avons initialement conçu GUSTAVE à destination de développeurs de noyaux de systèmes embarqués (disposant donc du code source) mais également d'intégrateurs systèmes. Ces derniers n'ont généralement pas accès au code source du noyau embarqué mais disposent de fichiers relogeables permettant de préparer un firmware embarquant des développements spécifiques. Ces conditions d'utilisation de GUSTAVE permettent de profiter de nombreuses optimisations, notamment concernant la préparation de l'application attaquante (place libre pour l'injection de séquences d'appels système, optimisation de la stratégie d'ordonnancement des tâches, *etc.*).

Au-delà des considérations de performances, l'approche par instrumentation au niveau du TCG implique :

— de maintenir du code dans des composants susceptibles d'évoluer ;
— de filtrer dynamiquement les zones de code exclues de l'instrumentation (code utilisateur, ou autres).

3.3 Traduction des entrées : générateur d'appels système

Demander à AFL de converger seul vers la génération d'un programme ayant un sens pour notre étude, c'est-à-dire contenant une liste d'appels système avec des valeurs d'argument plus ou moins conventionnels, nous a semblé sous-optimal.

Afin d'accompagner AFL dans la pertinence des cas testés, nous avons donc fait le choix d'intercaler, entre le système cible et AFL, un traducteur qui génère lui-même une trame constante de suite d'*opcodes* effectuant des appels système, et qui se serve des données brutes d'AFL uniquement pour définir quel appel système va être généré (souvent identifié par un numéro) et quelles valeurs prennent ses arguments. Ainsi, AFL n'a qu'à produire des données brutes comme étant une suite de {ID, arguments} pour interagir avec le code noyau.

Puisque ce traducteur intervient directement au niveau binaire, son implémentation finale dépend bien évidemment de l'architecture matérielle ciblée ainsi que de la manière dont le système d'exploitation cible expose les appels système aux programmes utilisateur [6]. Cela nécessite un minimum d'analyse de la cible pour savoir comment adapter le générateur à son ABI.

3.4 Environnement d'exécution de la cible

Étant donnée l'architecture d'AFL, le programme cible ne peut pas être le noyau du système d'exploitation. En effet, ce dernier ne peut pas s'exécuter sur le système hôte Linux en tant que simple programme utilisateur. QEMU va donc être chargé de son exécution dans l'équivalent d'une machine virtuelle.

Du point de vue d'AFL, ceci signifie que le programme cible devient QEMU. Nous avons ainsi développé une *board* QEMU, qui n'est autre que l'assemblage d'un micro-processeur virtuel et de périphériques virtuels [7] couplés à une implémentation d'un équivalent de *fork server* permettant de communiquer avec AFL. Cette *board* contrôle l'exécution du noyau cible avec l'entrée *fuzzée* d'AFL, et lui transmet son état terminal à chaque lancement d'un cas de test.

La *board* est responsable de :

— la communication avec AFL via des descripteurs de fichiers (statut/contrôle/entrées *fuzzées*) ;

6. Passage des arguments par la pile, par registre, déclenchement de l'appel système via l'instruction `int N/sysenter.` en `x86`, `sc` en `PowerPC`, etc.

7. Tous ces composants sont déjà fournis par QEMU.

— la traduction vers une séquence d'appels système ;
— projeter la *trace bitmap* dans la mémoire physique de la VM, afin que le noyau puisse la mettre à jour ;
— simuler des comportements POSIX classiques (signaux, `waitpid`, *timeout*) attendus par AFL.

En termes de performances, l'exécution d'une instance QEMU responsable du démarrage d'une machine virtuelle, puis de l'initialisation de périphériques, puis du noyau de système d'exploitation cible et finalement du programme utilisateur contenant la séquence d'appels système malveillants requiert un temps beaucoup plus important que le lancement d'une simple tâche utilisateur. Les deux parties suivantes décrivent ce que nous avons mis en œuvre dans GUSTAVE pour réduire ce coût en temps.

3.5 Considérations techniques de la *board* QEMU

Les descripteurs de fichiers servant à la communication ont des valeurs fixes (198/199) et sont hérités dans QEMU depuis le processus père AFL. La *board* QEMU se synchronise avec AFL en lisant et écrivant dans ces descripteurs bloquants.

La cartographie de couverture de code, ou *trace bitmap*, est une zone de mémoire partagée (SHM) créée par AFL et dont la clé est transmise dans l'environnement du processus fils. Notre *board* QEMU la récupère via la variable `__AFL_SHM_ID` et s'y attache.

À présent, le noyau de système d'exploitation cible, et plus précisément les *shim* injectés par AFL dans ses *basic blocks*, doivent pouvoir y accéder. Cependant cette *bitmap* existe uniquement dans la mémoire de l'hôte, le Linux responsable du lancement d'AFL et de QEMU. Elle n'est pas visible dans la mémoire de la VM démarrée par QEMU exécutant le noyau cible.

Initialement, nous avions implémenté une zone de mémoire d'entrée et sortie (*mmio*) au sein de la VM, à une adresse arbitraire à laquelle les *shim* allaient écrire. Tout accès à cette zone *mmio* déclenchait une interception de la part de QEMU qu'il traduisait par un accès à la zone de mémoire partagée dans l'hôte Linux. Cela engendre des performances catastrophiques (environ `0.5` cas de test/s), puisque chaque saut dans un *basic block* entraînait :

— un *VM exit* (entrée dans QEMU) ;
— la simulation d'un accès *mmio* ;
— un accès mémoire final vers la zone partagée avec AFL.

À la place, nous avons décidé de tirer profit de la flexibilité de la gestion de la mémoire offerte par QEMU. Il est possible de choisir avec

précision quelle page de mémoire physique de la VM peut être *mappée* dans la mémoire virtuelle de l'hôte Linux (cf. figure 3). Ainsi, nous avons pu associer directement la zone de mémoire de la VM accédée par les *shim* à la zone de mémoire partagée créée par AFL. La mise à jour de la *trace bitmap* par le noyau n'accuse désormais plus aucun surcoût.

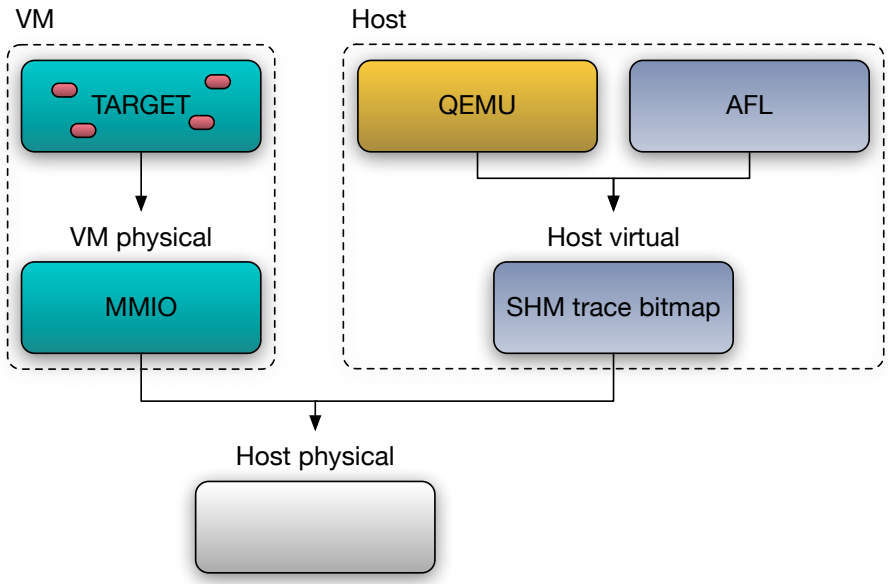

Fig. 3. AFL/QEMU *trace bitmap*.

3.6 *Fork server* de la *board* QEMU

Dans le contexte du fuzzing d'un système d'exploitation, nous ne pouvons pas laisser AFL injecter son *fork server* dans la cible. *Forker* un processus n'a pas vraiment de sens en dehors d'un système UNIX.

Ceci dit, l'approche est intéressante du point de vue d'AFL qui, lui, est un processus Linux, d'autant plus que nous ne souhaitions surtout pas modifier le design d'AFL. Nous avons donc décidé d'implémenter cette stratégie de *fork server* au sein de la *board* QEMU et de simuler le comportement attendu par AFL. À l'aide de *breakpoints* et de *snapshots* de la VM nous pouvons considérablement améliorer les performances de fuzzing sans modifier AFL.

La *board* QEMU installe un *breakpoint* à un endroit spécifique du noyau cible où nous considérons qu'il est intéressant de démarrer le fuzzing, généralement après l'initialisation de périphériques et durant le chargement des programmes/partitions utilisateur. Lorsque ce point d'arrêt est atteint, nous sauvegardons l'état complet de la machine virtuelle (*snapshot*) : processeur, mémoire et périphériques. Il permet d'effectuer une restauration du système à chaque nouveau cas de test généré par AFL.

QEMU contacte AFL pour l'informer qu'il est prêt à traiter une entrée. Notre *board* la traduit en une séquence d'appels système, l'injecte dans la mémoire de la VM et installe un nouveau point d'arrêt au niveau de la dernière instruction injectée. Le but est ici de pouvoir simuler une fin d'exécution *normale* du cas de test, l'équivalent d'un `exit()` d'un programme UNIX standard. Finalement, la VM reprend son exécution.

Notre *board* QEMU peut intercepter trois natures d'événements :

— une fin normale d'exécution (appelée *end-exec*) ;
— un time-out (appelé *end-timeout*) ;
— une faute (appelée *end-abort*).

Le *time-out* est déclenché à l'aide de timers internes à QEMU configurés dans la *board*, et démarrés avec la VM. Le déclenchement des fautes est traité dans la section 3.8.

Chaque *VM exit* (faute ou non) est accompagné de :

— la préparation d'un faux `waitpid(&status)` reflétant la terminaison d'un processus fils ;
— la restauration de l'état de la VM (*snapshot*) ;
— la lecture d'une nouvelle entrée fuzzée par AFL ;
— la traduction en séquence d'appels système ;
— la mise à jour du point d'arrêt de *end-exec* ;
— l'armement des *timers* pour *end-timeout* ;
— la reprise de la VM et attente d'un événement.

La figure 4 détaille notre implémentation de *fork server*.

3.7 Classification des comportements de la cible

Le système d'exploitation peut se comporter de manières différentes suivant les appels système testés et il revient à GUSTAVE de classer les cas pour signaler à AFL qu'il s'agit d'une exécution mettant en évidence une vulnérabilité ou non.

Premier cas : le test s'est mal terminé car le noyau a détecté qu'il y avait eu un contournement mémoire et l'a traité. Ce cas ne nous intéresse guère puisque de notre point de vue, le noyau a correctement géré le respect

des contraintes mémoire ; GUSTAVE remonte ce cas à AFL comme une exécution réussie.

Deuxième cas : le test s'est bien terminé du point de vue du noyau. Cela peut provenir de deux types de tests :

— l'appel système effectué n'engendre aucun contournement de la ségrégation mémoire ;

— l'appel système effectué a réussi à contourner la politique de ségrégation mémoire mais le noyau ne l'a pas détecté.

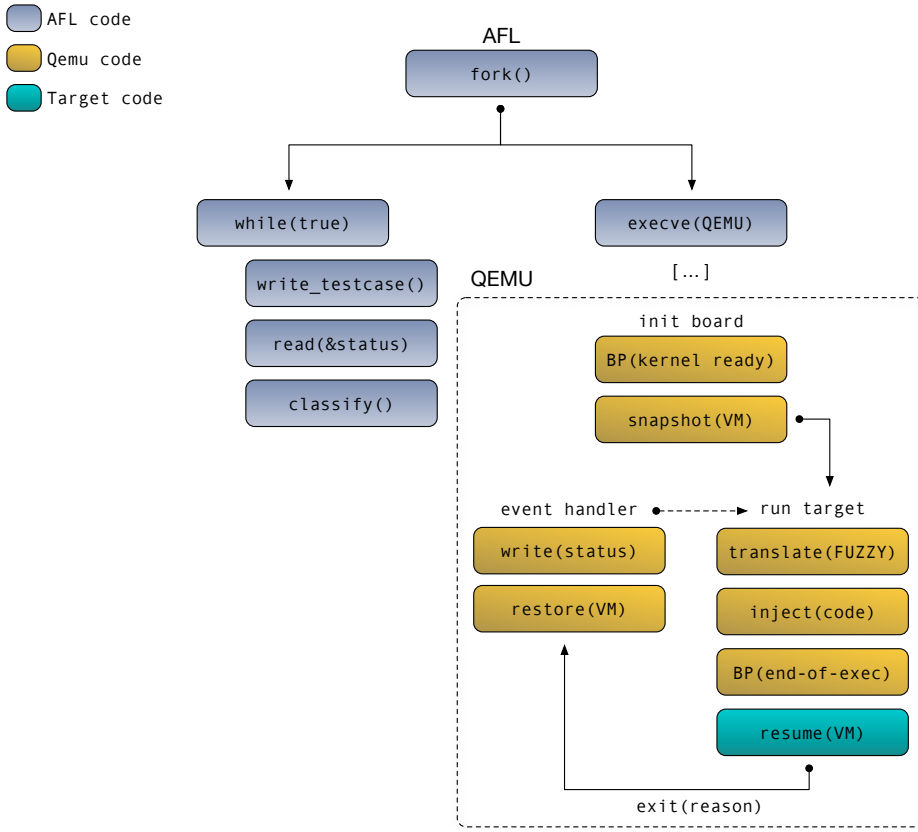

Fig. 4. QEMU *fork server*.

Le premier type est à classifier par GUSTAVE comme une exécution réussie également. Pour le deuxième type, il s'agit d'un test qui a mis à mal la propriété de ségrégation mémoire sans que cela ait levé d'erreur au niveau du noyau : il s'agit donc d'un cas de test ayant mis en évidence une vulnérabilité du noyau, à remonter à AFL en tant que *crash* (terminologie AFL pour les exécutions mal terminées).

Afin de distinguer entre les deux cas évoqués précédemment, il est obligatoire pour GUSTAVE de définir ses propres oracles mémoire et ses propres mécanismes de détection, différents de ceux implémentés par le noyau *fuzzé*. L'implémentation finale des critères de classification dans GUSTAVE est spécifique à chaque système d'exploitation et nécessite une étape d'analyse de la cible pour comprendre son modèle mémoire.

3.8 Interception des fautes et oracles mémoire

À l'inverse de simples applications utilisateur, détecter des fautes mémoire produites par un noyau de système d'exploitation n'est pas si évident. En effet, dans le cas d'un programme utilisateur, la *MMU* programmée par le noyau sert de *garde-fou* mémoire, en détectant lorsque une propriété de ségrégation mémoire n'est pas respectée, en envoyant des signaux à l'application, voire en terminant son exécution si nécessaire. Mais du point de vue d'un noyau, il n'existe généralement pas ces paradigmes car le noyau a souvent besoin de pouvoir accéder en toute légitimité à la mémoire des différentes tâches/processus/partitions qu'il gère.

Généralement, les noyaux disposent d'un service du type *kernel panic* rappelé lorsqu'un comportement inattendu a lieu en leur sein. Il s'agit d'un premier candidat pour QEMU pour l'interception de fautes. Cependant, il ne permettra d'intervenir que sur des fautes *identifiées* par le noyau.

Pour les cas non détectés par la *MMU* déjà en place, il est nécessaire de configurer QEMU et d'implémenter dans notre *board* QEMU une sorte d'extension de cette *MMU*. En essence, le but est de mimer les fautes de type *Segmentation Violation* levées par un CPU lors d'accès mémoire invalides effectués par des applications en dehors de leur espace d'adressage, mais cette fois du point de vue du noyau.

Cette extension nous permet de baliser des zones mémoire que nous considérons légitimes, d'autres que nous considérons illégitimes, et de détecter les accès illégitimes réalisés par le noyau cible. Remarquons que cela nécessite de connaître, au préalable, l'agencement mémoire spécifique à la cible (noyau et applications) pour proposer un mécanisme différent de celui déjà présent dans le noyau.

On peut imaginer que le non-respect de la ségrégation mémoire mise en place dans notre *board* QEMU peut par exemple déclencher une interruption lorsque les zones mémoire considérées comme illégitimes sont utilisées. En cas d'interruption déclenchée, la *board* QEMU est en mesure de notifier AFL en lui renvoyant un statut *end-abort*, afin de lui faire classifier le cas de test comme fautif.

3.9 Diverses optimisations à prendre en considération

Fuzzer un noyau ne va pas aussi vite, en termes d'exécution, qu'un simple programme utilisateur. Par exemple, il se peut que le noyau ordonnance un grand nombre de tâches annexes à celles auxquelles on s'intéresse. Cela peut avoir un impact non négligeable sur les performances de rapidité de *fuzzing*, et la métrique *test rate* d'AFL, indiquant le nombre de tests par seconde, peut être considérablement réduite.

Par ailleurs, les exécutions en parallèle, telles que le *multithreading* ou l'arrivée d'interruptions matérielles, peuvent influer sur la métrique *stability* d'AFL, indiquant le taux de déterminisme de comportement de la cible face à un test donné. Il est souhaitable que ce taux soit proche de 100%.

4 Mise en œuvre sur un premier exemple

Afin d'illustrer les concepts évoqués dans la section précédente, nous avons tenté de les mettre en œuvre pour *fuzzer* un petit système d'exploitation embarqué *open-source*, `POK`, dans sa version la plus avancée, celle fonctionnant pour l'architecture `x86`.

4.1 `POK`, une cible intéressante pour GUSTAVE

`POK` est un système d'exploitation pertinent à étudier pour plusieurs raisons :

— son code est *open-source* ;
— il met en place une stratégie intéressante de ségrégation mémoire ;
— l'implémentation de cette stratégie contient quelques défaillances que nous détaillons plus loin et que nous voulions tenter de détecter avec GUSTAVE.

Code source et AFL Puisque `POK` est un système d'exploitation open-source écrit en *C*, la phase d'instrumentation de la cible est simple : en effet, il est relativement aisé de recompiler `POK` (*gcc*) pour qu'AFL y injecte les *shims* dont il a besoin.

Un OS embarqué typique `POK` est un système d'exploitation embarqué temps réel mettant en œuvre des propriétés de ségrégation mémoire intéressantes. En effet, avec `POK`, chaque programme, ou « partition » dans le jargon consacré, s'exécutant au-dessus de ce système d'exploitation possède une portion mémoire qui lui est dédiée.

Comme dans beaucoup de systèmes d'exploitation embarqués, la définition de ces portions mémoire est déterministe ; elle s'effectue à la compilation. Une fois le système d'exploitation ainsi que ses partitions compilées, il n'est alors plus possible de rajouter dynamiquement de nouvelles partitions.

Design mémoire Le modèle mémoire de `POK` repose sur la segmentation `x86` [8], qui consiste à découper la mémoire en portions appelées segments. À la compilation de `POK`, un lot de deux segments (de code, de données) est défini pour chaque partition. Lorsque le flot d'exécution bascule d'une partition à une autre, le noyau met à jour les registres *cs* (*code segment*) et *ds* (*data segment*) avec le bon lot de segments. Notons également que le noyau possède lui aussi un jeu de segments qui lui est propre.

L'utilisation de la segmentation `x86` comme principe de ségrégation mémoire est en soi une bonne chose : si une partition tente d'utiliser une adresse hors du segment qui lui est alloué, une exception matérielle est levée ; mais encore faut-il définir les segments correctement. Il se trouve que `POK` définit le jeu de segments relatif au noyau en *flat*, autrement dit lorsque le noyau s'exécute, il a accès à *la totalité* de la mémoire.

Ce choix de conception peut se comprendre d'un point de vue fonctionnel puisqu'il facilite par exemple les accès mémoire lorsque le noyau doit gérer un échange de données entre deux partitions ; le noyau n'a qu'à copier les données d'une adresse à une autre sans se soucier de changer de segments puisqu'il a accès à la mémoire entière.

Cependant, ce choix de conception est tout de même risqué car cela signifie aussi que lorsque le noyau s'exécute, les accès mémoire ne sont plus limités par le matériel à une plage d'adresses restreinte. Cette mise à plat doit donc être compensée par des vérifications mémoire logicielles supplémentaires, en particulier au niveau des adresses utilisées par les arguments d'appels système, vecteur d'attaques potentiel depuis une partition vers le reste du système (noyau, autres partitions, etc.).

Faiblesses d'implémentation Si ces barrières logicielles mémoire sont implémentées et disponibles dans `POK` (fonction `POK_CHECK_PTR_IN_PARTITION`), tous les appels système ne s'en servent pas et ne proposent pas d'autres vérifications alternatives. C'est par exemple le cas de l'appel système `pok_current_partition_get_id` où l'argument `id` est déréférencé sans vérification (voir listing 1).

8. *Intel 64 and IA-32 Architectures Software Developer's Manual*, section `3.2.4`.

```
extern uint8_t pok_current_partition;
#define POK_SCHED_CURRENT_PARTITION pok_current_partition

pok_ret_t pok_current_partition_get_id (uint8_t *id)
{
  *id = POK_SCHED_CURRENT_PARTITION; // dereferencement
  return POK_ERRNO_OK;
}
```

Listing 1. Appel système `POK` vulnérable.

Cette absence de vérification donne par exemple la possibilité à une partition de passer en argument de l'appel système une adresse arbitraire, potentiellement hors de sa plage autorisée, qui sera écrite lorsque le noyau exécute l'appel système, sans qu'aucune alerte de violation de ségrégation mémoire ne soit levée.

Notre but a été tout d'abord de valider que cette vulnérabilité identifiée peut tout de même être détectée avec GUSTAVE et, le cas échéant, tenter de découvrir par *fuzzing* d'autres vulnérabilités de ce type.

4.2 GUSTAVE pour `POK`

Comme évoqué dans la section 3, si les concepts sont génériques, l'implémentation finale du générateur d'appels système, de l'environnement de simulation ainsi que des mécanismes de détection mémoire sont spécifiques au système d'exploitation que l'on souhaite *fuzzer*.

Configuration et instrumentation de la cible Le nombre de partitions ainsi que leurs caractéristiques (taille mémoire, durée de leur fenêtre temporelle, etc.) se configurent à la compilation de `POK`. Afin d'avoir un système d'exploitation proche de la réalité, nous avons choisi de configurer deux partitions que `POK` va ordonnancer tour à tour.

La partition 1 est simplissime. Elle se contente d'afficher quelques messages à l'écran. Sa présence n'est requise que pour rendre le système représentatif d'un environnement embarqué ordonnançant plusieurs partitions pouvant communiquer entre elles. Sa fenêtre temporelle est très courte pour ne pas perdre de temps à exécuter des parties de code peu utiles.

La partition 2, quant à elle, va être le support de notre *fuzzing*. Une portion mémoire est réservée pour son code, initialement vide à la compilation, qui va évoluer lors du *fuzzing* et contenir un ou plusieurs appels système `POK` dont les arguments viendront des entrées d'AFL. Sa fenêtre temporelle est très grande.

Afin qu'AFL puisse collecter les informations relatives à la couverture de code, il faut également compiler `POK` avec des *shims* d'AFL. Pour cela, nous nous sommes appuyés sur les outils d'instrumentation fournis par AFL (`afl-as`, `afl-gcc`), en adaptant le contenu du *shim* à l'architecture de GUSTAVE (accès à la zone *mmio*).

Notons qu'il n'est utile d'instrumenter que le code du noyau. À la compilation, le code des partitions a donc été volontairement exclu de l'instrumentation.

Générateur d'appels système `POK` expose aux partitions environ une cinquantaine d'appels système. L'accès à ces appels système depuis une partition s'effectue via l'instruction `int 42`, avec deux paramètres : l'identifiant d'appel système (`ID`) dans le registre `eax`, et l'adresse d'un tableau de cinq arguments dans le registre `ebx`. Le programme à générer doit :

— récupérer l'ID d'appel système donné par AFL dans le registre `eax` ;
— calculer la taille de données brutes attendue pour remplir tous les arguments de cet appel système ;
— organiser les données brutes provenant d'AFL pour construire les arguments ;
— invoquer l'appel système.

Suivant les appels système, le nombre d'arguments varie (de 1 à 5), le type d'arguments également : entier (par exemple un numéro de port), chaîne de caractères, pointeur vers une structure, etc. Divers points peuvent être *fuzzés* :

— la valeur de l'adresse passée à `ebx` ;
— la valeur des adresses d'arguments de type pointeur ;
— le contenu pointé par les arguments de type pointeur ;
— la valeur des arguments de type entier, caractère, etc.

En fonction de ce que l'on souhaite *fuzzer*, chaque cas donne lieu à une version de générateur différent. Par exemple dans le cas où l'on souhaite explorer les vulnérabilités atteignables en *fuzzant* le contenu pointé par un pointeur de structure, il faut en plus générer les *opcodes* qui attribuent les données brutes d'AFL au contenu pointé par cette adresse et donnent au pointeur une adresse valide dans la partition.

Ainsi, en partant de données brutes telles que celles du listing 2, GUSTAVE génère le programme illustré par le listing 3. Ce dernier peut être représenté par le pseudo-code *C* décrit par le listing 4. La stratégie appliquée ici est le traitement de chaque argument sans distinction de

type, donc en particulier en *fuzzant* simplement la valeur des adresses d'arguments de type pointeur.

```
00000000: 0011 1111 1122 2222 2203 1111 1111 2222
00000010: 2222 3333 3333 4444 4444 5555 5555
```

Listing 2. Exemple de données brutes d'AFL.

```
push    $0x0
push    $0x0
push    $0x0
push    $0x22222222  # parametre 2
push    $0x11111111  # parametre 1
push    $0x2         # 2 parametres requis
lea     (%esp),%ebx
mov     $0x196,%eax  # loc_ID: 0, PokID: 0x196
                     #(POK_SYSCALL_PARTITION_GET_PERIOD)
int     $0x2a

push    $0x55555555  # parametre 5
push    $0x44444444  # parametre 4
push    $0x33333333  # parametre 3
push    $0x22222222  # parametre 2
push    $0x11111111  # parametre 1
push    $0x5         # 5 parametres requis
lea     (%esp),%ebx
mov     $0x6e,%eax   # loc_ID: 3, PokID: 0x6e
                     #(POK_SYSCALL_MIDDLEWARE_QUEUEING_CREATE)
int     $0x2a
```

Listing 3. Code `x86` généré à partir du listing 2.

```
POK_SYSCALL_PARTITION_GET_PERIOD(
          0x11111111/* *period */,
          0x22222222 /* unused */)

POK_SYSCALL_MIDDLEWARE_QUEUEING_CREATE(
          0x11111111 /* *name */,
          0x22222222 /* size */,
          0x33333333 /* direction */,
          0x44444444 /* discipline */,
          0x55555555 /* *id */)
```

Listing 4. Pseudo-code `C` correspondant au listing 3.

Définition et implémentation d'oracles mémoire Dans le cas précis de `POK`, il se trouve que le mécanisme de pagination disponible en `x86` n'est pas exploité. Cela nous a rendu l'implémentation d'une ségrégation mémoire facile puisque nous nous en sommes servi, au niveau de QEMU,

pour mettre en place notre propre logique de ségrégation mémoire : ne sont *mappées* que les pages de la plage d'adresses relatives au noyau et aux partitions. Cela induit que, lorsqu'une adresse en dehors de cette plage est utilisée, une faute de page (*#PF*) est levée dans QEMU (mais jamais transmise à `POK`) ; QEMU signale alors cette faute à AFL.

4.3 Premiers résultats

Résultats pour `pok_current_partition_get_id` La première campagne de *fuzzing* a eu pour but de détecter la vulnérabilité décrite dans le listing 1. N'est donc testé que l'appel à `pok_current_partition_get_id`, l'appel système identifié comme vulnérable. Le listing 5 contient une partie des logs obtenus avec des adresses de pointeurs d'`id` quelconques.

```
ptr validation failure for 0x7f234261 ba 0x234000 sz 0x10c8e0
ptr validation failure for 0x13abf61 ba 0x234000 sz 0x10c8e0

pok_current_partition_get_id: 0x85440483
ES: 10, DS: 10
CS: 8, SS: 112fd0
EDI: 352fa4, ESI: 85440483
EBP: 352f74, ESP: 1131a8
EAX: 0, ECX: 11de1c
EDX: 7, EBX: 0
EIP: 101cc0, ErrorCode: 2
EFLAGS: 46

ptr validation failure for 0x54353d68 ba 0x234000 sz 0x10c8e0

pok_current_partition_get_id: 0x90b3d090
ES: 10, DS: 10
CS: 8, SS: 112fd0
EDI: 352fa4, ESI: 90b3d090
EBP: 352f74, ESP: 1131a8
EAX: 0, ECX: 11de1c
EDX: 7, EBX: 0
EIP: 101cc0, ErrorCode: 2
EFLAGS: 46

ptr validation failure for 0xfc438d6c ba 0x234000 sz 0x10c8e0
```

Listing 5. Logs lors du *fuzzing* de `pok_current_partition_get_id`.

On observe deux cas distincts. Les `ptr validation failure` sont des messages d'erreur que le noyau `POK` lève lors de la vérification de l'adresse que contient `ebx` (`0x54353d68, 0x13abf61, 0x7f234261`, etc.). Il s'agit de toutes les fois où `ebx` contient une adresse invalide. Ces cas sont *maîtrisés* par le système d'exploitation et ne nous intéressent guère.

L'autre type de message (en bleu), listant des valeurs de registres, nous intéresse plus. Il s'agit des logs qu'affiche `POK` lorsqu'une exception de type

#PF est levée par la *board*. À titre expérimental, nous avons transmis l'exception à `POK` afin d'illustrer la génération de faute mémoire liée à notre oracle. Avoir obtenu ce type de logs signifie la chose suivante : AFL ayant réussi à converger vers une adresse valide pour `ebx`, le programme a pu quelque peu continuer son exécution et atteindre le déréférencement mentionné dans le listing 1. Les informations données dans les logs montrent qu'il y a en effet eu une écriture en `0x85440483` et `0x90b3d090`, adresses classifiées par GUSTAVE comme étant hors de la plage autorisée à la partition.

En d'autres termes, GUSTAVE a su détecter un cas de contournement de la ségrégation mémoire non géré par le système d'exploitation.

Recherche de vulnérabilités similaires Avec la même stratégie de cartographie mémoire, nous avons étendu le *fuzzing* précédent à l'ensemble des appels système de `POK`. Des exceptions de type *#PF* similaires ont été levées. Les diverses valeurs d'`eip` contenues dans les logs nous ont permis de remonter à l'ensemble des points vulnérables au même problème, dont le listing 6 en illustre un extrait.

```
pok_current_partition_get_start_condition+86:    mov  %edx,(%ecx)
pok_current_partition_get_duration+85:           mov  %ecx,(%eax)
pok_current_partition_get_lock_level+86:         mov  %edx,(%ecx)
pok_thread_get_status+226:                       mov  %edx,0x10(%ecx)
```

Listing 6. Autres fonctions vulnérables trouvées par GUSTAVE.

En vérifiant à la main le code de ces fonctions, il s'est avéré qu'elles étaient effectivement bien vulnérables à une écriture mémoire arbitraire.

Détection d'autres types de vulnérabilités Le but du travail réalisé sur POK était surtout de valider que les concepts génériques de GUSTAVE fonctionnaient, ce qui est le cas. Cependant, on pourrait imaginer vouloir poursuivre plus la recherche de vulnérabilités au sein de POK.

En effet, les nouvelles vulnérabilités décrites précédemment sont relativement proches du problème que nous avions identifié dans l'appel système `pok_current_partition_get_id` (absence de vérification). Cependant, dans POK, il existe au moins une autre vulnérabilité (de type *off-by-one*) concernant tous les appels systèmes utilisant la fonction de vérification `POK_CHECK_PTR_IN_PARTITION`. Cette fonction est par exemple appelée en début de l'appel système `POK_SYSCALL_MIDDLEWARE_QUEUEING_SEND` qui envoie un buffer d'une partition vers une autre. Pour détecter ce type de vulnérabilités, il serait alors nécessaire de définir un oracle mémoire plus fin, considérant des zones précises des partitions concernées.

5 Bonus

5.1 *Quid* d'autres logiques de mutations ?

En choisissant le *fuzzer* d'AFL, la logique de mutations pour générer de nouveaux cas de tests est celle implémentée dans *afl-fuzz*. Mais on pourrait très bien imaginer vouloir profiter d'autres logiques de mutations, afin de tester d'autres méthodes ayant des approches de *coverage-guided fuzzing* différentes. Des projets dérivés d'AFL proposent justement des méthodes de mutations alternatives (comme AFLGo [14], FairFuzz [3] ou AFLFast [13]).

Un des intérêts de l'approche de GUSTAVE est d'être indépendant du moteur de *fuzzing* implémenté par l'*afl-fuzz* original. Autrement dit, il suffit de remplacer *afl-fuzz* par l'un des projets cités ci-dessus pour profiter de leur logique de mutations. Une mise en œuvre pour chacun des trois exemples a été testée avec la version d'implémentation de GUSTAVE pour `POK`. Sans aller jusqu'à l'analyse des crashs obtenus avec ces mutations alternatives, cela a permis de valider le fait qu'il est trivial et transparent du point de vue de GUSTAVE d'utiliser l'une ou l'autre des méthodes de mutations.

5.2 *Quid* d'autres architectures matérielles ?

GUSTAVE est un projet en développement qui ne se limite ni à son implémentation pour `POK`, ni à l'architecture *x86*. Des travaux ont été notamment initiés pour l'appliquer à l'architecture *PowerPC* pour `POK` et d'autres OS propriétaires.

L'architecture de l'outil ne change pas. QEMU offre le support du matériel virtualisé (microprocesseur et périphériques). La définition d'une nouvelle *board* implique l'assemblage de ces composants à l'implémentation du *fork server*, du traducteur et des oracles.

Comme précisé précédemment, les oracles sont à la fois spécifiques à l'OS cible et à l'architecture matérielle.

6 Discussion et conclusion

Afin de profiter des capacités d'AFL pour *fuzzer* la surface d'attaque exposée par les systèmes d'exploitation aux programmes de moindre privilège, GUSTAVE fait office de traducteur entre le monde d'AFL et celui du noyau cible, tout en gardant les autres acteurs (AFL, QEMU, noyau) quasi-intacts. Si l'approche générique est valable pour l'ensemble

des systèmes d'exploitation embarqués, l'implémentation finale n'en reste pas moins spécifique à la cible, afin d'adapter au mieux le *fuzzing* à la logique mémoire et à la gestion des appels système du noyau étudié.

La version de GUSTAVE pour `POK` sur l'architecture `x86` a permis de valider sur un exemple simple les concepts génériques exposés : GUSTAVE a permis de détecter une vulnérabilité connue d'écriture mémoire arbitraire malgré la ségrégation mémoire mise en place et de trouver, par *fuzzing*, d'autres morceaux de code vulnérables au même problème.

On peut toutefois remarquer que le succès de détection de GUSTAVE est conditionné par la manière dont sont définis ses oracles mémoire. Ces définitions primordiales sont laissées au choix et à l'objectif de l'utilisateur de l'outil et à adapter suivant la cible considérée.

Références

1. QEMU supported platforms and accelerators. `https://wiki.qemu.org/Documentation/Platforms`.
2. Andrew Griffiths. AFL qemu mode. `https://github.com/mirrorer/afl/tree/master/qemu_mode`.
3. Caroline Lemieux. FairFuzz. `https://github.com/carolemieux/afl-rb`.
4. Chamith, Svensson, Dalessandro, Newton. Instruction Punning : Lightweight Instrumentation for x86-64. `https://www.researchgate.net/project/Liteinst`.
5. Dave Jones. Trinity : A Linux System call fuzz tester. `https://codemonkey.org.uk/projects/trinity`.
6. Denis Selianin. Researching Marvel Avastar Wi-Fi. `https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf`, 2018.
7. Dmitry Vyukov. Syzkaller, unsupervised coverage-guided kernel fuzzer. `https://github.com/google/syzkaller`, 2016.
8. Fabrice Bellard. QEMU : the FAST ! processor emulator. `https://www.qemu.org`.
9. Fabrice Bellard. QEMU user space emulation. `https://qemu.weilnetz.de/doc/qemu-doc.html#QEMU-User-space-emulator`.
10. Intel. Pin, A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.
11. Intel. Intel Processor Trace (Intel PT). `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2013.
12. Jack Tang, Moony Li. When virtualization encounter AFL. `https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf`, 2016.
13. Marcel Böhme. AFL Fast. `https://github.com/mboehme/aflfast`.
14. Marcel Böhme. AFL Go. `https://github.com/aflgo/aflgo`.
15. Michał Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl`.

16. MIT, Hewlett-Packard. DynamoRIO, Dynamic Instrumentation Tool Platform. http://www.dynamorio.org, 2001.
17. Nathan Voss. afl-unicorn : Fuzzing Arbitrary Binary Code. https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf, 2017.
18. Nathan Voss. afl-unicorn : Fuzzing the 'Unfuzzable'. https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5, 2017.
19. Nathan Voss. afl-Unicorn : github. https://github.com/Battelle/afl-unicorn, 2017.
20. NCC Group. Project Triforce : Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/, 2016.
21. Nguyen Anh Quynh & Hoang-Vu Dang. Unicorn : The ultimate CPU emulator. https://www.unicorn-engine.org, 2015.
22. S. Schumilo, et al. kAFL : Hardware-Assisted Feedback Fuzzing for OS Kernels. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-schumilo.pdf, 2017.
23. University of Maryland. Dyninst. https://www.dyninst.org/dyninst.
24. Valerie Zhao. Evaluation of Dynamic Binary Instrumentation Approaches. https://repository.wellesley.edu/cgi/viewcontent.cgi?article=1739&context=thesiscollection.
25. Yong Chuan Koh. Fuzzing the Windows Kernel. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-fuzzing-the-windows-kernel.pdf, 2016.
26. Yong Chuan Koh. Platform Agnostic Kernel Fuzzing. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-Platform-Agnostic-Kernel-Fuzzing-FINAL.pdf, 2016.