Recherche de vulnérabilités dans les piles USB : approches et outils

Jordan Bouyat et Fernand Lone-Sang
{jbouyat,flonesang}@quarkslab.com

QuarksLAB

Résumé La norme USB est aujourd'hui adoptée à grande échelle et son interface tend à se généraliser rapidement. En effet, il n'est pas rare de trouver des ports USB sur les objets que nous utilisons quotidiennement (ordinateurs, *smartphones*, etc.). L'USB est entrée dans les habitudes des utilisateurs de l'informatique et constitue souvent un moyen d'attaque privilégié. Les clés USB, par exemple, sont parfois plébiscitées par les attaquants comme un bon moyen pour s'introduire avec succès dans les systèmes informatiques d'organisations, même lorsqu'ils ont un niveau de sécurité élevé. Par ailleurs, l'USB constitue également un excellent vecteur d'attaque. En effet, étant simplement un protocole de transport pour d'autres protocoles spécifiques (SCSI, SATA, etc.), l'USB permet de couvrir une surface d'attaque large dans les couches basses des systèmes d'exploitation. Pour ces raisons, il est intéressant de rechercher des vulnérabilités dans les piles USB dans différents systèmes. Étant donné que la majorité des systèmes sont fermés et que leurs sources ne sont pas disponibles, la recherche de vulnérabilités par *fuzzing* est, sans doute, la plus appropriée. Cet article constitue un retour d'expérience quant à la mise en œuvre d'un *fuzzer* USB pour les plates-formes Windows.

Mots-clés: Recherche de vulnérabilités, *Universal Serial Bus* (USB), *Fuzzing*, Qemu, Facedancer.

1 Contexte et problématique

Avant les années 1990, l'ajout d'un nouveau périphérique dans un ordinateur personnel était une opération souvent délicate et source de problèmes importants, si bien que cela nécessitait un minimum de connaissances en informatique. Il était nécessaire, entre autres, de résoudre les problèmes liés à la connectique et à la configuration [1] du nouveau périphérique, mais surtout de redémarrer complètement le système afin que le nouveau périphérique soit reconnu. En réponse à ces limitations, la norme USB a émergé au milieu des années 1990 pour faciliter l'interconnexion de

1. Typiquement, il était nécessaire d'affecter manuellement les ressources nécessaires à son bon fonctionnement (par exemple, les plages d'adresses, les lignes d'interruption, etc.) et de résoudre les conflits qui pouvaient en résulter.

périphériques dans un ordinateur personnel, pour simplifier — du point de vue de l'utilisateur final — leur configuration qui est opérée à chaud. Aujourd'hui, Madame Michu (mais Monsieur Michu également, pour ne pas faire de jaloux) sait utiliser un périphérique USB.

La norme USB a évolué, a mûri, s'est généralisée et ne se limite plus aux ordinateurs personnels. En effet, il n'est pas rare de trouver des ports ou des connectiques USB sur les objets que nous utilisons quotidiennement, tels que les voitures, les *smartphones*, les téléviseurs et, de plus en plus, sur des systèmes inhabituels tels que l'électroménager. Force est de constater que l'USB est entrée dans les habitudes des utilisateurs de l'informatique et, *de facto*, constitue souvent un vecteur d'attaque privilégié sur des organisations à haute valeur ajoutée. Par exemple, on peut l'utiliser pour recopier discrètement des données confidentielles, voire pour s'introduire avec succès dans les systèmes informatiques, en particulier lorsqu'ils sont situés sur des réseaux informatiques isolés avec un niveau de sécurité élevé.

Dans un contexte d'intrusion, les périphériques USB, et plus spécifiquement les clés USB, sont pratiques car elles sont discrètes et furtives, se glissent aisément dans un portefeuille, dans un sac ou un cartable, sont négligemment laissées sur un coin de bureau ou même s'oublient sur une place de parking d'entreprise [2]. C'est d'ailleurs par une clé USB qu'a été introduit, dans l'usine de Natanz, le ver informatique Stuxnet [11] supposément conçu pour saboter le programme nucléaire iranien.

Le support de l'USB dans la majorité des systèmes d'exploitation sur étagère est relativement mûr. Malgré cela, des vulnérabilités liées aux piles USB continuent à être régulièrement révélées. Une des raisons de cet échec vient du fait que le standard USB se veut générique et on y greffe des modules complémentaires implémentant des protocoles plus spécifiques (par exemple, Ethernet, SATA, etc.) et dépendant du type de périphérique. Cette complexité les fragilise vis-à-vis de la sécurité. À notre connaissance, à l'exception du *jailbreak* de la PlayStation 3 [18], de la vulnérabilité dans iBoot [14] d'iOS et, plus récemment, de quelques CVEs [10,15], peu de vulnérabilités ont été révélées sur les couches basses de l'USB. La majorité des vulnérabilités portent sur ces greffons qui sont particulièrement nombreux et complexes.

À la vue des exemples mentionnés précédemment, nous concevons alors que rechercher des vulnérabilités dans les piles USB est particulièrement intéressant du point de vue de l'attaquant car corrompre la pile USB

2. Dans ce scénario d'attaque, il y a fort à parier qu'une personne curieuse, dans un moment d'inattention, tentera de connecter le périphérique trouvé à son poste de travail pour en voir son contenu, permettant alors à une éventuelle attaque de se dérouler.

signifie potentiellement corrompre le noyau d'un système d'exploitation (dans le cas d'un noyau monolithique), puis corrompre potentiellement tous les programmes qui s'exécutent au-dessus de ce noyau pour, enfin, prendre complètement le contrôle du système. Aussi, le fait que la pile USB ne soit qu'un socle pour d'autres protocoles plus spécifiques la rend d'autant plus intéressante car elle permet de couvrir une surface d'attaque large dans les couches basses des systèmes d'exploitation. Bien souvent, seule une approche d'analyse en boîte noire est possible et la recherche de vulnérabilités liées à l'USB par *fuzzing* reste la plus appropriée. Le concept de *fuzzing* sur les piles USB n'est pas nouveau en soi. Nous présentons notre retour d'expérience sur l'implémentation d'un *fuzzer* pour rechercher des vulnérabilités dans les piles USB de Windows.

2 Rappels sur la norme USB

L'*Universal Serial Bus* (USB) est une norme de bus série conçue pour simplifier les échanges des données entre un ordinateur, appelé « hôte », et des périphériques. Actuellement, la norme en est à sa troisième version et définit, pour chaque version, les câbles, les connecteurs supportés ainsi que le protocole de communication.

Au fil des évolutions de la norme USB, le protocole de communication entre le contrôleur USB et les périphériques est resté quasi-inchangé d'un point de vue logique. En effet, de nouvelles requêtes sont rajoutées d'une version à une autre de la norme, permettant ainsi la réutilisation et la mutualisation du code implémentant le protocole de communication.

Cette section rappelle quelques éléments sur la norme USB. En nous basant sur la topologie de bus USB, nous commençons par introduire la terminologie employée dans la norme. Nous présentons ensuite les fondements du protocole de communication que nous illustrons finalement par quelques requêtes standards. Le lecteur est invité à consulter les spécifications de la norme USB [19,20,22] et de l'interface du contrôleur de bus [6,5,7,8] pour les éléments spécifiques à chaque version.

2.1 Topologie de l'USB

La figure 1 présente la topologie typique d'un bus USB. Elle est organisée selon une topologie hiérarchique dans laquelle l'ordinateur communique avec les périphériques qui lui sont connectés selon un modèle maître-esclave. En effet, seul l'ordinateur, via le contrôleur hôte, initie les transferts (cf. section 2.2) vers les périphériques (qui ne sont pas autorisés à initier par

eux-mêmes des transferts). Ainsi, il ne peut y avoir de communication directe entre deux périphériques, à l'exception des périphériques implémentant l'USB « On-The-Go » [3] [21] connectés par un lien point-à-point. Typiquement, le contrôleur hôte, à la demande du système d'exploitation, scrute (pour le terme anglais *polling*) périodiquement un clavier USB, auquel a été attribuée une adresse de bus unique lors de la phase d'énumération, pour déterminer si des données sont disponibles (résultant d'une touche pressée par exemple).

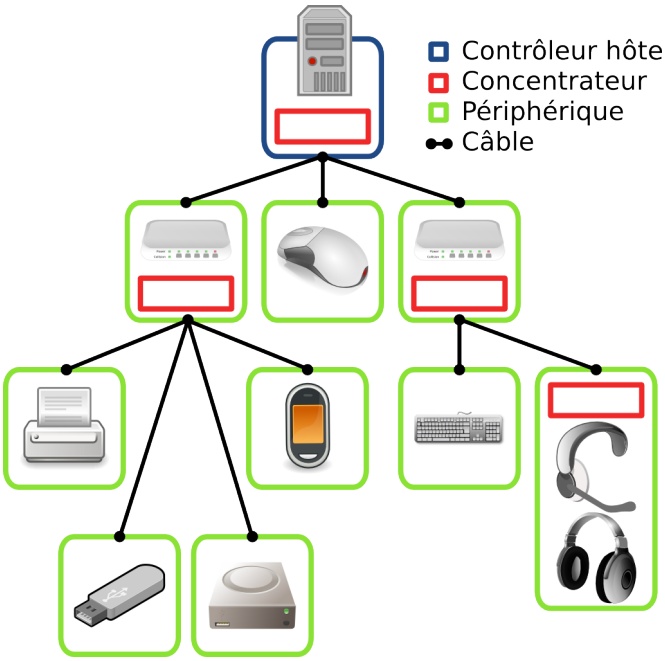

Figure 1. Exemple de topologie d'un bus USB

Contrôleur hôte. Le contrôleur hôte est chargé de générer, sur son bus, les transactions mises en place par le système d'exploitation (pour être plus précis, par la pile USB et par les pilotes de périphériques USB). Concrètement, pour programmer des transactions, le système d'exploitation construit en mémoire une liste chaînée de descripteurs de transferts, qui contient toutes les informations permettant au contrôleur hôte de générer les dites transactions. Les registres et les structures des descripteurs de transferts permettant au système d'exploitation de dialoguer avec le con-

3. Nous rappelons que cette extension rajoute à la norme USB un mécanisme d'auto-distribution des rôles maître-esclave, leur permettant, à tour de rôle, d'émettre des requêtes au périphérique pair.

trôleur hôte sont normalisés pour chaque version de la norme USB : les spécifications UHCI (*Universal Host Controller Interface*) [6] et OHCI (*Open Host Controller Interface*) [5] ont été définies pour l'USB 1.x, l'EHCI (*Enhanced Host Controller Interface*) [7] pour l'USB 2.x et, finalement, l'xHCI (*eXtended Host Controller Interface*) [8] pour l'USB 3.x. Comme représenté sur la figure 1, il convient de noter que plusieurs périphériques peuvent être directement connectés au contrôleur hôte par l'intermédiaire de son concentrateur racine.

Périphériques. Il existe deux types de périphérique USB. Typiquement, les périphériques *simples* (par exemple, une souris) ne fournissent qu'un seul service alors que les périphériques *composites* en fournissent plusieurs. Par exemple, une imprimante multifonction délivre des services d'impression, de numérisation, etc. L'accès à ces services se fait par la mise en œuvre de transferts avec des *Endpoints*, que l'on peut conceptualiser comme un ensemble de registres auxquels on accède exclusivement en lecture ou en écriture avec des transactions.

D'un point de vue logique, un périphérique USB contient un ensemble de structures de données, appelées descripteurs, qui détaillent au système d'exploitation les caractéristiques du périphérique, les configurations supportées, les services fournis et les registres pour mettre en place ces services. Les descripteurs sont lus par des requêtes au *Default Control Pipe* associé à l'*Endpoint 0*. La figure 2 représente ces différents descripteurs. Il existe principalement 4 types de descripteur, organisés de façon hiérarchique, avec chacun un rôle bien défini. Notons que nous excluons délibérément les *String Descriptors* des types de descripteur principaux car ils sont optionnels et ne sont rajoutés par les fabricants que pour représenter de manière textuelle (encodée en Unicode) des informations destinées à l'utilisateur telles que le nom du périphérique, le nom du fabricant, le numéro de série, etc. Le lecteur est invité à se référer aux spécifications USB [19,20,22] pour le format détaillé des descripteurs.

Le *Device Descriptor* contient des informations sur le périphérique, telles que ses identifiants [4], la version de la norme USB implémentée, la taille maximale des paquets ainsi que le nombre de configurations.

Les *Configuration Descriptors* détaillent les configurations supportées par le périphérique, notamment en termes de consommation électrique, de débits supportés, etc. La configuration choisie s'effectuera par une requête

4. Il s'agit du `vendorID` et `productID` d'un périphérique attribué par l'USB Implementers Forum (USB-IF) et qui l'identifient de manière unique. Les couples `vendorID` et `productID` sont répertoriés à l'adresse `http://www.linux-usb.org/usb.ids`.

`SET_CONFIGURATION`, provenant de l'hôte, spécifiant la configuration désirée par son index.

Les *Interface Descriptors* précisent les services disponibles dans la configuration choisie. Typiquement, on retrouve dans chaque *Interface Descriptor* la classe (par exemple, *Human Interface Device* ou HID pour un clavier ou une souris) auquel appartient le périphérique, des informations spécifiques à cette classe de périphériques (par exemple, la sous-classe, le protocole implémenté, etc.) et le nombre d'*Endpoint Descriptors*.

Les *Endpoint Descriptors* décrivent les registres, les types de transfert supportés (*Control, Interrupt, Bulk* et *Isochronous*), la direction (`IN` ou `OUT`[5]) des transferts disponibles pour chaque registre ainsi que d'autres informations (bande passante requise, durée nécessaire entre deux scrutations d'un même registre, la taille maximale des transferts) pour interroger les services fournis. Précisons que chaque *Endpoint* ne supporte qu'un type de transfert et chacun d'entre eux est unidirectionnel, à l'exception de l'*Endpoint 0* qui supporte à la fois les transferts `IN` et `OUT`.

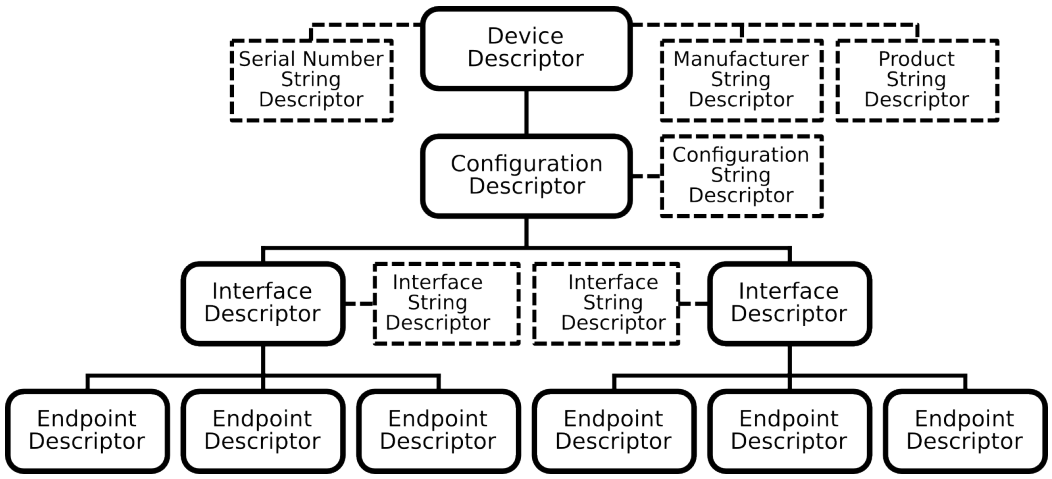

Figure 2. Hiérarchie de descripteurs dans un périphérique USB

Dans la terminologie utilisée par la norme USB, un concentrateur est considéré comme un périphérique particulier auquel d'autres périphériques se connectent. En effet, il possède également des descripteurs permettant au système d'exploitation de l'identifier comme un concentrateur et de détecter les événements survenant sur son segment de bus.

5. La direction des transferts est exprimée par rapport au contrôleur hôte.

2.2 Protocole de communication

Le protocole de communication décrit dans la norme USB est un protocole pouvant être représenté sur trois niveaux. En effet, les transferts sur les bus USB se manifestent par l'émission de nombreux paquets USB. Lorsqu'ils sont regroupés, ces paquets représentent des transactions individuelles qui constituent, à leur tour, les transferts. Les trois niveaux du protocole de communication sont représentés sur la figure 3.

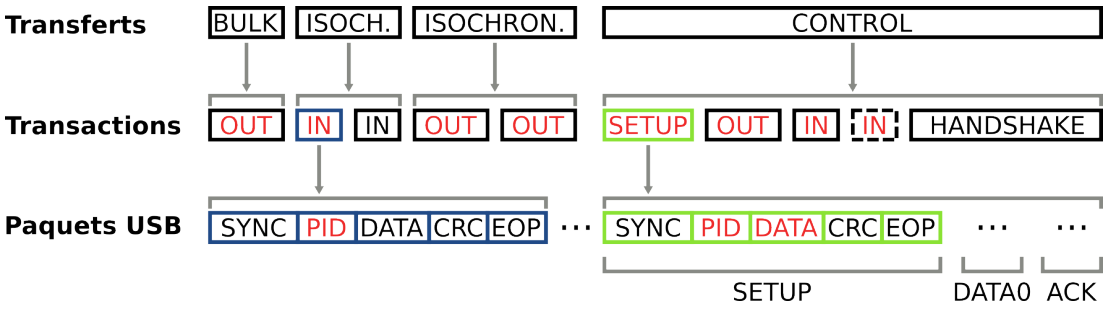

Figure 3. Représentation multi-niveaux des transferts USB

Transferts USB. Nous distinguons quatre types de transfert qui peuvent véhiculer des commandes (également appelées requêtes) standards ou spécifiques, des interruptions ou simplement des données aux périphériques. Nous les décrivons brièvement dans la suite.

Les transferts de type *Isochronous* correspondent à des transferts qui requièrent un débit minimum, voire constant, telles que les trames audio ou vidéo. Par exemple, un microphone et un haut-parleur USB utilisent les transferts isochrones pour veiller à ce qu'aucune distorsion ne s'effectue sur les signaux lors de leurs transferts sur les bus. Il s'agit, sans doute, du mode de transfert le plus efficace (en terme de débit, de bande passante et de délai) mais il est également le plus complexe à mettre en œuvre.

Les transferts de type *Bulk* sont utilisés pour véhiculer une grande quantité de données sans garantie en terme de qualité de service. Ce type de transfert est mis en place, par exemple, avec les imprimantes USB. En effet, bien que le débit reste un critère de performance important, un travail d'impression peut être transmis lentement sans que cela n'empêche le service d'être délivré correctement.

Les transferts de type *Interrupt* sont destinés aux *Endpoints* qui envoient ou reçoivent ponctuellement de faibles quantités de données et qui requièrent des garanties en terme de latence. Les *Endpoints* accessibles par

des transferts de type *Interrupt* doivent être interrogés périodiquement afin de déterminer s'ils ont des données à transférer. Au vu de ce mode de communication, il est évident que ces transferts diffèrent des interruptions mises en œuvre sur d'autres bus tels que PCI. Néanmoins, ils émulent cette fonctionnalité. Pour reprendre l'exemple du clavier ou de la souris USB, ceux-ci possèdent généralement un *Endpoint* nécessitant des transferts de type *Interrupt* qui est périodiquement interrogé pour déterminer si les données (par exemple, résultant d'une touche pressée) sont prêtes à être transférées. Le système d'exploitation peut programmer le contrôleur hôte pour scruter périodiquement ces *endpoints* puis générer une interruption matérielle uniquement lorsqu'une donnée est disponible.

Les transferts de type *Control* sont utilisés pour émettre des commandes (également appelées requêtes) pour les périphériques USB. Ils interviennent généralement lors de la phase de configuration d'un périphérique USB, en particulier, pour l'énumération des bus, pour lire les descripteurs standards et attribuer une adresse unique à un périphérique.

Chaque périphérique USB dispose d'*Endpoints* exposant des descripteurs qui spécifient la manière dont les données qu'ils contiennent peuvent être atteintes. Par exemple, le contenu d'un disque audio inséré dans un lecteur de CD-ROM USB, lu depuis un explorateur de fichiers, repose sur l'*Endpoint* de données utilisant des transferts de type *Bulk* alors que les accès effectués par un lecteur multimédia passent par un autre *Endpoint* qui exige des transferts de type *Isochronous*.

Transactions USB. Comme illustré sur la figure 3, les transferts USB sont constitués d'une ou plusieurs transactions USB définies par le standard, elles-mêmes constituées de paquets réellement transmis sur les bus USB. De façon générale, à un transfert correspond une transaction USB à l'exception des transferts de type *Control*. En effet, ceux-ci requièrent plusieurs transactions : ils commencent par une phase de configuration symbolisée par une transaction `SETUP`, suivie par une phase de transfert de données (contenant la commande) pouvant se matérialiser par plusieurs transactions `IN` ou `OUT` et se terminent par une phase d'interrogation de statut afin de déterminer si l'exécution de la requête par le périphérique a réussi ou a échoué.

Paquets USB Les paquets USB correspondent aux trames émises physiquement sur les bus. Leur format est rappelé sur la figure 3. Chaque paquet USB est précédé par une séquence de bits de synchronisation (`SYNC`) et est terminé par une séquence de bits particulière pour indiquer la fin de

paquet (`EOP`). Ces séquences sont automatiquement générées par le *Serial Interface Engine* dans le contrôleur hôte. Le type de paquet est défini par le *Packet ID* (`PID`) qui est suivi par des informations (par exemple, une adresse ou des données) qui varient en fonction du type de paquet. Finalement, un CRC est inséré avant `EOP` pour détecter et corriger, dans la mesure du possible, les erreurs de transmissions.

La complexité de la norme USB est masquée aux programmeurs. En effet, les descripteurs de transferts mis en place par le système d'exploitation pour générer une transaction sont particulièrement bien pensés pour cacher les niveaux précédemment évoqués. Au final, le programmeur ne fournit au contrôleur hôte que les informations qui lui sont nécessaires pour générer automatiquement les paquets correspondant aux transferts sur les bus. Les informations minimales nécessaires pour la mise en place d'un transfert ont été représentées (colorées en rouge) sur la figure 3.

2.3 Connexion d'un périphérique : exemple de transactions

Afin d'obtenir les caractéristiques et de configurer les périphériques nouvellement connectés, le système d'exploitation, via le contrôleur hôte, met en place des transferts de type *Control*. Comme illustré sur la figure 3, ce type de transfert se manifeste par plusieurs transactions. Une première transaction `SETUP`, suivie par des transactions `OUT` contiennent la requête standard à destination du *Default Control Pipe* (*Endpoint 0*) que doit exécuter le périphérique. Lorsqu'une réponse est attendue, le descripteur correspondant à la réponse du périphérique à la requête envoyée est ensuite lue par une transaction `IN`. Enfin, le contrôleur hôte vérifie que le transfert des données s'est bien déroulé par une autre transaction.

Le format d'une requête standard (correspondant à une transaction `SETUP`) est représenté sur la figure 5. Il s'étend toujours sur 8 octets. Le champ `bmRequestType` définit par exemple la direction du transfert (hôte vers périphérique ou inversement), le type de transfert (*Standard*, *Class*, etc.) et le descripteur destinataire (*Device*, *Interface*, etc.). Le champ `bRequest` contient la commande envoyée. Il s'agit ici de la commande `GET_DESCRIPTOR` dont le format de la réponse dépend du descripteur interrogé. Le lecteur est invité à se référer aux spécifications USB [19,20,22] pour plus de détails sur la signification des champs et le format des différentes transactions.

Offset	Champ	Taille (octet)	Valeur	Description
0	`bmRequestType`	1	bitmap	**D7** Data Phase Transfer Direction 0 = Host to Device 1 = Device to Host **D6..5 Type** 0 = Standard 1 = Class 2 = Vendor 3 = Reserved **D4..0 Recipient** 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	`bRequest`	1	value	request
2	`wValue`	2	value	value
4	`wIndex`	2	index/offset	index
6	`wLength`	2	count	number of bytes to transfer

Figure 4. Format d'une requête standard

2.4 L'énumération : exemple

Le tableau 1 résume les différents paquets échangés lors du processus d'énumération qui survient à la connexion d'un périphérique USB. Dans cet exemple, il s'agit d'un clavier connecté à Windows 8.1.

Les champs des requêtes et des réponses sont volontairement omis. L'hôte s'adresse au périphérique avec l'adresse 0 avant de lui attribuer une adresse à la transaction 3. S'ensuivent des échanges de descripteurs de périphérique, de configuration et de chaînes de caractères. Une fois ces échanges terminés, l'hôte choisit et définit la configuration du périphérique à la transaction 22. Enfin, le descripteur HID du clavier est échangé en 23 et en 24. Entre autres, ce descripteur spécifie à l'hôte le format des données à recevoir.

Les figures 5 et 6 présentent un transfert de type *Control* pour lire le *Device Descriptor* d'un périphérique.

3 État de l'art des recherches de vulnérabilités sur USB

Cette section présente un rapide état de l'art des recherches de vulnérabilités dans les piles USB. Puisque l'approche de *fuzzing* reste globalement la même, à savoir injecter des fautes au niveau des entrées du système

Numéro	Source	Destination	Requête/Descripteur
1	host	**0.0**	GET_DESCRIPTOR Request Device
2	0.0	host	GET_DESCRIPTOR Response Device
3	host	0.0	**SET_ADDRESS**
4	host	**11.0**	GET_DESCRIPTOR Request Device
5	11.0	host	GET_DESCRIPTOR Response Device
6	host	11.0	GET_DESCRIPTOR Request Configuration
7	11.0	host	GET_DESCRIPTOR Response Configuration
8	host	11.0	GET_DESCRIPTOR Request String
9	11.0	host	GET_DESCRIPTOR Response String
10	host	11.0	GET_DESCRIPTOR Request String
11	11.0	host	GET_DESCRIPTOR Response String
12	host	11.0	GET_DESCRIPTOR Request String
13	11.0	host	GET_DESCRIPTOR Response String
14	host	11.0	GET_DESCRIPTOR Request Device Qualifier
15	11.0	host	GET_DESCRIPTOR Response Device Qualifier
16	host	11.0	GET_DESCRIPTOR Request Device
17	11.0	host	GET_DESCRIPTOR Response Device
18	host	11.0	GET_DESCRIPTOR Request Configuration
19	11.0	host	GET_DESCRIPTOR Response Configuration
20	host	11.0	GET_DESCRIPTOR Request Configuration
21	11.0	host	GET_DESCRIPTOR Response Configuration
22	host	11.0	**SET_CONFIGURATION Request**
23	host	11.0	GET_DESCRIPTOR Request RPIPE
24	11.0	host	GET_DESCRIPTOR Response RPIPE

Table 1. Enumération d'un périphérique HID sous Windows 8.1

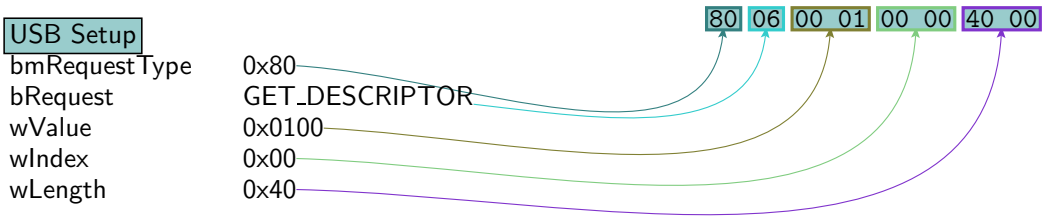

Figure 5. Requête *Device Descriptor*

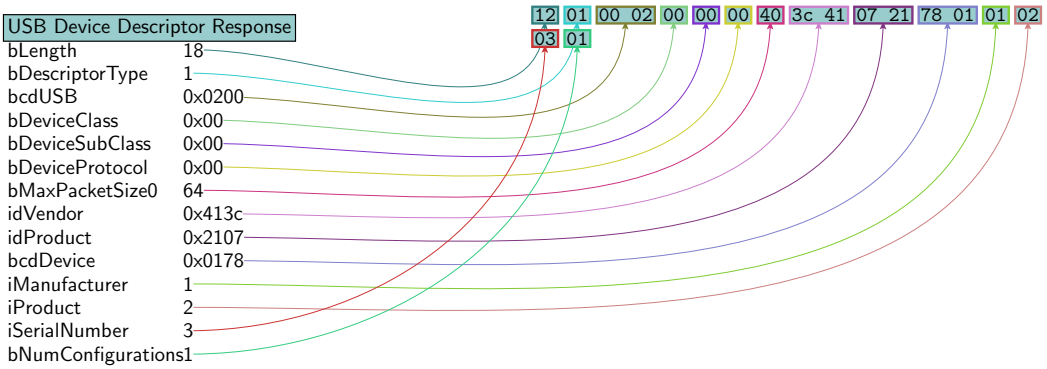

Figure 6. Réponse *Device Descriptor*

et à observer ses sorties dans le but de détecter des éventuelles erreurs, nous décrivons dans la suite les approches utilisées dans la littérature pour injecter des fautes dans les piles USB.

3.1 Utilisation d'environnements virtualisés

L'utilisation d'environnements virtualisés pour effectuer du *fuzzing* sur les piles USB est couramment rencontrée dans les différentes publications. En effet, ceux-ci sont propices à cette utilisation car les gestionnaires de machines virtuelles permettent aussi bien de rediriger le trafic d'un périphérique USB réel vers une machine virtuelle, que de connecter à la machine virtuelle un périphérique USB virtuel que l'on contrôle complètement. Le premier cas (cf. figure 7) permet de mettre en place un *fuzzing* par mutation car il repose sur des trames « réelles », tandis que le second permet aussi bien du *fuzzing* en génération qu'en mutation.

Parmi les différentes solutions de virtualisation existantes, seule Qemu (avec éventuellement Bochs) est utilisée en pratique. En effet, étant donné que les sources sont libres, il est possible de se placer dans les fonctions clés de gestion de l'USB pour modifier le comportement. Dans le cas de la redirection du trafic, pour une machine virtuelle Linux, il faut modifier la fonction `async_complete()` du fichier `usb_linux.c` [6] pour intercepter et modifier le trafic. La figure 7 montre une plate-forme de test dans laquelle le trafic du périphérique USB physique traverse l'hôte avant d'être modifié puis redirigé dans la machine virtuelle cible.

L'implémentation d'un périphérique USB virtuel se fait, quant à elle, en rajoutant du code dans les sources de Qemu (en créant, par exemple, un nouveau fichier `my_device.c` et en modifiant les fichiers *Makefile* adéquats), dans lequel le comportement du périphérique, les routines de connexion et de déconnexion sont définies. C'est l'approche décrite par MWR Labs dans [23] et qui est représentée sur la figure 8. Cette approche présente l'avantage de pouvoir définir soi-même les fonctions qui seront appelées pour gérer les flux de contrôle et de données. Il suffit de renseigner les champs `handle_control` et `handle_data` de la structure `USBDeviceClass` de façon à pointer vers des fonctions dans notre périphérique USB virtuel.

Par ailleurs, comme nous programmons entièrement le comportement du périphérique USB virtuel dans Qemu, il est possible de mettre en place, par exemple, des canaux de communication pour transférer les données reçues (ou récupérer les données à transmettre, grâce à nos fonctions de gestion de flux) depuis un programme externe. Cette approche est

6. Comme expliqué dans [17].

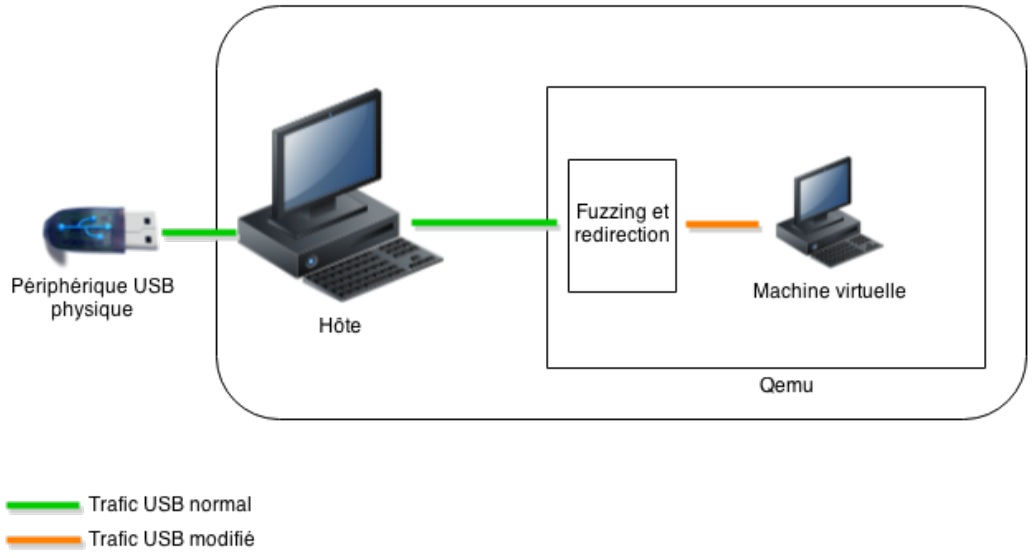

Figure 7. Fuzzing du trafic USB redirigé

celle qu'a utilisée Tobias Mueller [16] dans ses travaux. Il a mis en place deux tubes (en anglais, *pipes*) nommés. Le premier a servi à rediriger les requêtes USB du système d'exploitation virtualisé vers un programme du système d'exploitation hôte et le second a permis quant à lui d'y écrire les réponses (c'est-à-dire, les descripteurs) qui étaient renvoyées au système d'exploitation virtualisé. Notre *sniffer* de trafic USB, présenté en section 4, s'inspire de cette architecture logicielle.

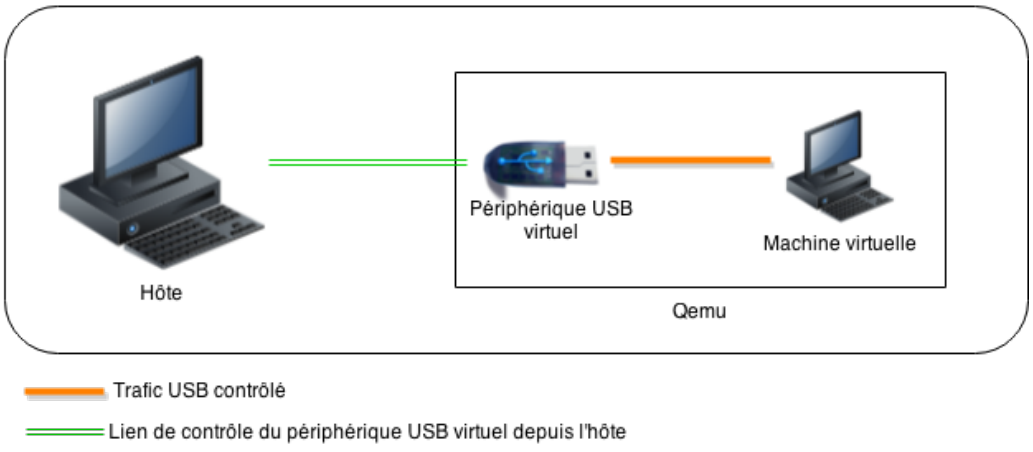

Figure 8. Fuzzing en utilisant un périphérique USB virtuel contrôlé depuis l'hôte

Enfin, une dernière approche consiste à se placer directement au niveau du contrôleur USB. Nous l'avons vérifié expérimentalement pour capturer du trafic. Cette approche est inédite et aucun travail, à notre connaissance, n'a exploité cette technique. En se plaçant dans la fonction `uhci_handle_td()` du fichier `hcd-uhci.c`, on voit passer la totalité du trafic circulant sur le contrôleur UHCI, c'est-à-dire des périphériques physiques redirigés et des périphériques virtuels.

En termes d'approche de *fuzzing*, les environnements virtualisés présentent un grand nombre d'avantages. Pour commencer, ils ne nécessitent pas de matériel spécifique. Plusieurs instances d'environnements virtualisés peuvent être lancées et cette approche se prête particulièrement bien à la parallélisation de l'exécution des cas de tests. Par ailleurs, la mise en œuvre de *snapshots* permet de restaurer une machine rapidement dans un état sain pour l'exécution d'un autre test, et il est possible de profiter des débogueurs intégrés aux solutions de virtualisation ou bien de rajouter du code dans le gestionnaire de machines virtuelles pour analyser comment un événement (par exemple, un plantage) s'est produit.

3.2 Détournement de fonctions dans la pile USB

Bien que pratique, l'approche de *fuzzing* au travers du gestionnaire de machines virtuelles n'est pas forcément la plus efficace. Cela dépend fortement de la cible des tests. Si un pilote de périphérique spécifique est testé, l'approche la plus adaptée consiste à modifier à la volée les données en entrée des fonctions clés dans le pilote de périphérique. Cette approche est implémentée, par exemple, dans Uhooker [7] et elle est, par nature, spécifique à un pilote donné. Il convient de noter que, puisque nous instrumentons le système analysé, nous ne sommes pas à l'abri d'effets de bord lors de l'analyse et il est possible qu'une partie des vulnérabilités découvertes par ce moyen ne soient pas exploitables en pratique.

3.3 Utilisation de matériel physique

En fonction de la cible, l'utilisation d'environnements virtualisés n'est pas toujours possible. Par ailleurs, l'environnement virtualisé jouant le rôle de mandataire (en anglais, *proxy*), nous ne sommes pas protégés d'un bogue d'implémentation de notre module d'injection, voire un bogue d'implémentation dans le gestionnaire de machine virtuelle (ce que nous

7. `http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Uhooker`

avons constaté à nos dépends au cours de l'implémentation de nos outils). Ainsi, il est parfois préférable d'utiliser du matériel physique. La suite en présente quelques-uns qui sont souvent usités.

Facedancer Le Facedancer est une carte électronique permettant de générer ses propres paquets USB. Cette carte a été développée par Travis Goodspeed. Elle est composée d'un microcontrôleur MSP430 [12] avec lequel on communique par une liaison série, et d'un contrôleur USB MAX3421E [13] contrôlé par le MSP430 via un bus SPI. Ce dernier peut être configuré, en positionnant des registres dédiés, pour agir comme un hôte ou comme un périphérique. Ainsi, cet outil permet d'émuler des périphériques USB, donc de se comporter en « client » communiquant avec un hôte, mais aussi de se comporter en tant qu'hôte. Il est également possible de s'attaquer à la pile USB d'un périphérique que celle d'un hôte, ce qui en fait un outil très intéressant.

L'utilisation la plus courante du Facedancer consiste à émuler un périphérique USB. La carte est reliée d'un côté à la machine qui génèrera les réponses aux requêtes (port USB gauche sur la figure 9) et de l'autre, à l'hôte dont on teste la pile USB (port USB droit). Les requêtes USB de la machine cible sont transmises par une liaison série [8] à la machine générant les réponses, en commandant au Facedancer de transmettre à la cible ces réponses. L'avantage principal du Facedancer est que la complexité de la pile USB du périphérique émulé est déportée sur une machine, en permettant l'écriture de périphérique USB en Python. Il agit alors comme un *proxy* série-USB. Cela permet de mettre en place une architecture de *fuzzing* et de débogage simple à mettre en œuvre.

Utilisé de cette manière, le Facedancer nous offre les possibilités de générer les réponses USB désirées aux requêtes, de capturer et de rejouer le trafic USB. L'implémentation actuelle de notre *fuzzer* pour Windows décrite en section 4 peut utiliser le Facedancer, tant pour la capture [9] que pour l'injection. Au cours de ces travaux, nous avons identifié quelques limitations qui empêchent cependant cet outil d'émuler tout type de périphérique USB, ce qui limite grandement le spectre des recherche de vulnérabilités qu'il est possible de faire.

Teensy Le Teensy est une carte de développement complète comprenant un microcontrôleur AVR et supportant l'USB. Utilisé conjointement avec

8. Un FTDI est connecté au MSP430.
9. Uniquement dans le cas où le Facedancer capture son propre trafic.

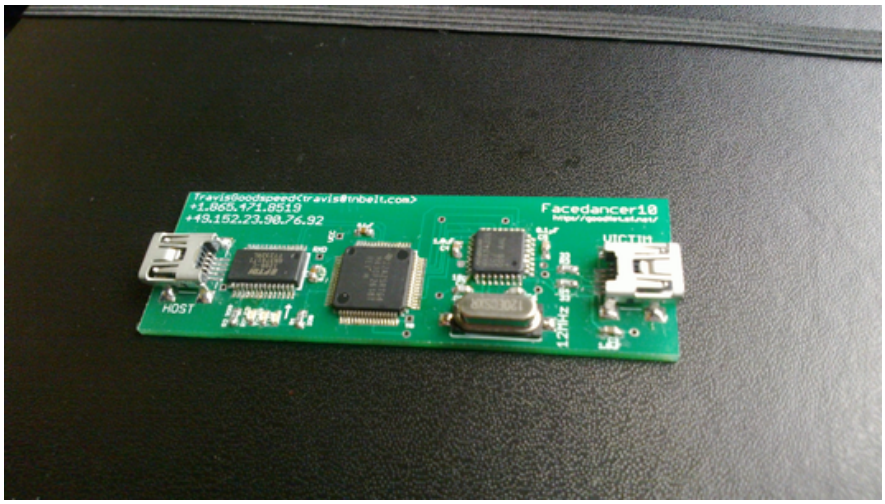

Figure 9. Facedancer 10

la bibliothèque LUFA [3] (*Lightweight USB Framework for AVRs*), qui permet, entre autres, de jouer le rôle d'hôte ou de périphérique. Cette bibliothèque est riche et supporte de nombreuses classes de périphériques standards.

Cette bibliothèque, utilisée avec un microcontrôleur AVR, est intéressante car elle permet de modifier directement les descripteurs USB du périphérique que l'on veut créer. Par conséquent, si une vulnérabilité est découverte sur un pilote d'une classe de périphérique USB d'un hôte, il suffit de modifier le descripteur en question pour déclencher la vulnérabilité et dans le meilleur des cas, l'exploiter. On obtiendrait ainsi un périphérique « clé en main » pour déclencher cette vulnérabilité. Cette approche a été utilisée par Fabien Perigaud pour son article dans MISC [17]. Cette piste n'a pas été explorée, car les efforts ont été portés sur le développement du *fuzzer* plus que sur l'aspect exploitation.

USB over IP USB over IP est une technique permettant de transporter des paquets USB sur un réseau IP. De cette manière, un périphérique USB peut se retrouver éloigné géographiquement, mais surtout être partagé entre plusieurs machines. Il est alors possible de modifier les paquets réseaux, et donc les paquets USB. Mais cette méthode requiert du matériel supplémentaire et l'installation d'une passerelle logicielle USB vers IP sur la machine.

4 Retours d'expérience sur la mise en œuvre d'un *fuzzer*

Cette section présente un retour d'expérience sur la mise en place d'un *fuzzer* pour une analyse de vulnérabilités dans les piles USB 1.x et 2.x. Les outils développés jusqu'à présent permettent de mettre en œuvre une chaîne complète de *dumb-fuzzing* et quelques expérimentations sont en cours.

4.1 Description générale du *fuzzer*

Étant donné la complexité du protocole USB, l'approche de *fuzzing* vers laquelle nous nous sommes dirigés immédiatement est le *fuzzing* par mutation. Elle consiste à collecter un ensemble de paquets USB valides sur lesquels on effectue ponctuellement des mutations. Les paquets mutés sont ensuite injectés sur le système cible dont nous analysons le comportement par différents agents. La figure 10 reprend l'idée générale du processus de *fuzzing* que nous avons envisagé. Elle se découpe grossièrement en trois unités fonctionnelles : une unité de capture de transactions USB, une unité de mutation des données capturées et une unité d'injection de paquets USB sur le système analysé.

Capture de trames Il existe différentes façons de capturer le trafic USB entre un hôte et son périphérique en fonction de la configuration dans laquelle nous nous plaçons. Il est possible de :

- capturer le trafic entre un périphérique physique et le système d'exploitation en se basant sur les outils de capture disponibles au sein de systèmes d'exploitation sur étagère, par exemple USBMon sous Linux et USBPcap sous Windows ;
- capturer le trafic transitant par un hôte ou un périphérique que nous avons préalablement programmé (par exemple, un FaceDancer grâce aux périphériques qu'il émule, et les réponses qu'il reçoit, un BeagleBoard-XM configuré comme un proxy USB) ;
- capturer le trafic transitant par une machine virtuelle dans laquelle fonctionnerait l'hôte cible en utilisant les options de débogage de la machine virtuelle (ex, *Virtual USB debug* dans VMWare) ;
- capturer le trafic transitant par un dispositif dédié à la capture de paquets USB, tels des analyseurs protocolaires. Par exemple, un Beagle USB 480 permet de capturer du trafic USB 1.1 et USB 2.0, un Beagle USB 5000 v2 supporte en plus l'USB 3. Ces dispositifs restent tout de même assez onéreux (2950 et 6000 dollars respectivement).

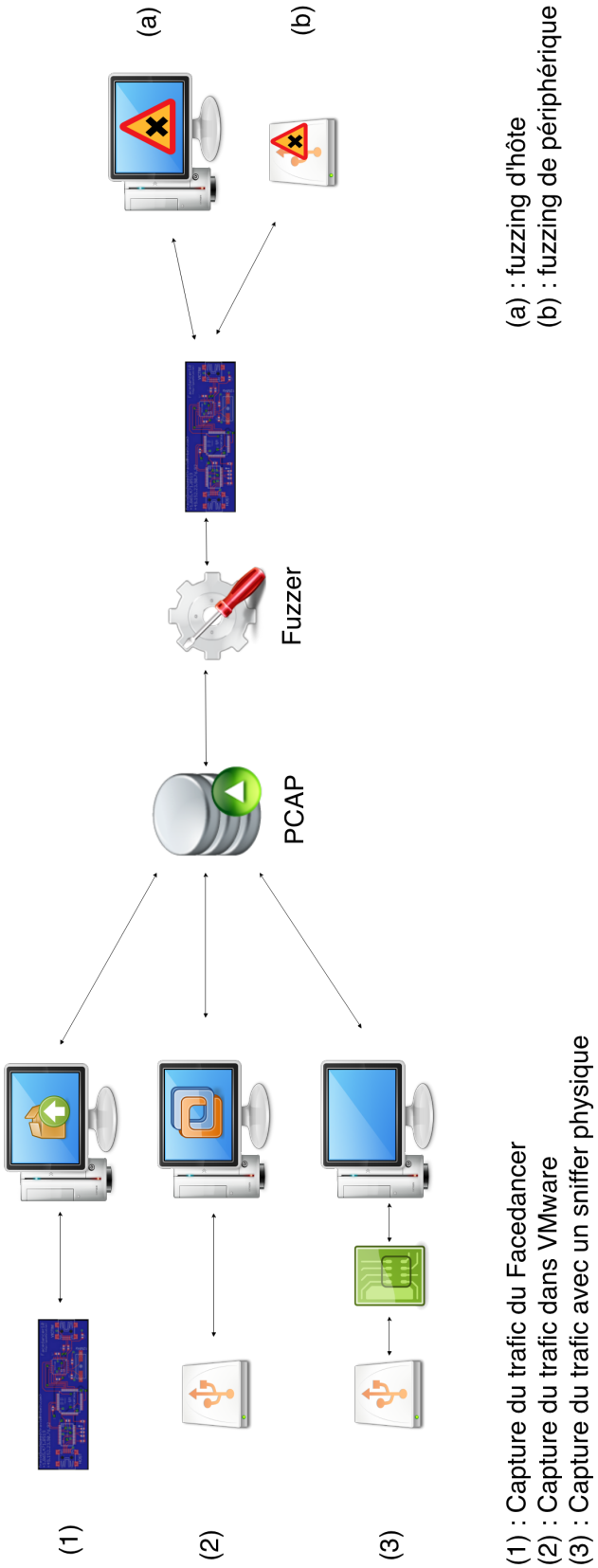

FIGURE 10. Architecture du *fuzzer* implémenté

Mutation de trames La difficulté reste dans le choix des trames à muter pour atteindre de manière optimale la cible de l'analyse et passer au travers des éventuelles vérifications opérées par les couches intermédiaires avant d'y arriver. Il y a une grande surface d'attaque, du fait qu'il existe beaucoup de descripteurs qui contiennent des champs qu'on pourrait qualifier de sensibles, car ils se réfèrent les uns les autres avec des index, des tailles, ce qui fait beaucoup de choses à contrôler au niveau du driver. Cependant, c'est ce qui rend difficile le choix des données à muter. Nous utilisons actuellement Radamsa [4] en *dumb-fuzzing* mais il serait sans doute judicieux d'utiliser un modèle.

Injection de trames Nous avons présenté en section 3 plusieurs manières d'injecter des transactions USB en fonction de la cible. Nos efforts sont concentrés sur le FaceDancer, mais nous développons, en parallèle, un module sur Qemu. Le principe est identique dans tous les cas et nous faisons un rejeu naïf des trames. Tout d'abord, nous chargeons les données capturées. Dans de le cas du *fuzzing* d'hôte, à chaque requête de l'hôte, nous jouons la prochaine réponse disponible. C'est le même principe pour le *fuzzing* de périphériques, sauf que nous rejouons les requêtes plutôt que les réponses. La raison pour laquelle nous nommons ce rejeu comme naïf vient du fait qu'il s'agit d'un rejeu strict qui ne maintient pas une machine à états du protocole. Lorsqu'on doit rejouer la requête ou la réponse que l'on désire modifier, on la mute avant de l'émettre.

On peut dire que le fuzzer travaille en mode déconnecté ou hors-ligne dans le sens où les données ne sont pas mutées à la volée, mais sont mutées à partir d'une précédente capture.

4.2 Démarches adoptées et outils résultants

Notre première approche s'est concentrée sur Qemu, car en faisant l'état de l'art, c'est ce qui nous a semblé le plus adapté. En réalité, nous avons eu au départ des difficultés pour injecter nos réponses. Nous nous sommes ensuite tournés vers le FaceDancer. Certaines de ces difficultés résolues, nous nous sommes intéressés de nouveau à Qemu.

Concrètement, nous avons modifié les sources de Qemu pour capturer le trafic USB d'une part et, d'autre part, commencé à implémenter un périphérique virtuel. Pour ce dernier, l'approche que nous avons suivie est identique à celle de Tobias Mueller [16] et utilise des tubes nommés.

Travaux sur Qemu Les fichiers relatifs à la gestion de l'USB de Qemu sont situés dans `hw/usb/` et `include/hw`. Pour faire notre périphérique,

nous nous sommes inspirés de `dev-hid.c` et nous avons implémenté le nôtre. La création d'un périphérique USB n'est pas très complexe. Il suffit de remplir les bonnes structures de données. Les extraits de code suivants permettent de voir comment s'y prendre. La structure suivante représente le cœur du périphérique virtuel :

```
static void usb_mon_device_class_initfn(ObjectClass *klass, void *
    data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    USBDeviceClass *uc = USB_DEVICE_CLASS(klass);
    uc->init           = usb_mon_device_initfn;
    uc->product_desc   = "Mon device";
    uc->usb_desc       = &desc_usbdevice;
    uc->handle_reset   = usb_mon_device_handle_reset;
    uc->handle_control = usb_mon_device_handle_control;
    uc->handle_data    = usb_mon_device_handle_data;
    uc->handle_destroy = usb_mon_device_handle_destroy;
    dc->desc = "Mon device perso";
    dc->vmsd = &vmstate_usb_device;
}
```

On spécifie dans la structure les pointeurs de fonctions. On a, en particulier :

```
    uc->handle_control = usb_device_handle_control;
```

Ainsi, lorsque le périphérique recevra des requêtes standards, spécifiques (HID par exemple) ou autres sur l'*Endpoint* 0, la fonction `usb_device_handle_control()` sera appelée. Le pointeur de fonction assigné au champ `handle_control` respecte le prototype suivant :

```
device_handle_control(USBDevice *dev, USBPacket *p, int request, int
     value, int index, int length, int8_t *data)
```

On peut donc savoir quelle requête à été émise (*request*, *value*, *index*), et pour répondre, il suffit d'écrire les données dans le *buffer* pointé par `data` et de renseigner la taille des données écrites dans `p->actual_length`. Comme nous nous focalisons sur le *fuzzing* des données à l'énumération, il suffit de lire la requête, de choisir la réponse adaptée et de la muter avant de la renvoyer. On regarde ensuite si l'hôte ou sa pile USB sont toujours vivants, et si oui, on déconnecte le périphérique et on recommence.

Travaux sur le FaceDancer En raison des limitations que nous avons identifiées en implémentant la solution de capture et d'injection basée sur Qemu, nous nous sommes ensuite intéressés aux dispositifs d'injection physique et, en particulier, au FaceDancer. L'avantage principal d'avoir

un module physique est de pouvoir *fuzzer* des systèmes qui peuvent difficilement être virtualisés, ce qui peut être le cas de systèmes embarqués.

Grâce au code fourni par Travis Goodspeed [1], nous avons pu facilement prendre en main le FaceDancer et commencer à développer nos périphériques USB en Python, puis à modifier simplement les informations retournées dans les descripteurs. Par la suite, Andy Davis a rendu publiques les sources de son outil d'émulation, de *fuzzing* et *fingerprinting* USB appelé Umap [2]. Notre *fuzzer* actuel se base essentiellement sur Umap. Celui-ci permet de *fuzzer* différentes classes de périphériques, mais il faut ajouter les cas de tests à la main, ce qui ne nous convient pas pour automatiser les tests.

Nous avons donc étendu Umap pour l'adapter à nos besoins avec les modules suivants :

- lecture/écriture de PCAP ;
- module de rejeu ;
- module de *fuzzing* ;
- module de *logs* pour le moniteur.

Le mécanisme de *fuzzing* est assez simple. Notre *fuzzer* commence par *parser* le PCAP (notre source de trafic valide) et remplir une liste de paquets qu'on utilisera pour rejouer les réponses.

Le moniteur analyse les requêtes de l'hôte pour détecter une anomalie. Par exemple, si l'hôte n'effectue plus de requêtes depuis un temps que nous fixons expérimentalement, l'hôte et/ou sa pile USB ne répondent plus et ont probablement planté.

Le module de *log* note quelle réponse a fait planter la cible et l'heure. L'avantage de ces modules et qu'ils sont assez indépendants les uns des autres, et on peut facilement les adapter, par exemple, à un périphérique virtuel Qemu au lieu de les utiliser avec le FaceDancer.

Limitations Bien qu'étant un outil extrêmement intéressant, le FaceDancer possède tout de même quelques limitations qui empêchent d'utiliser cet outil comme outil de capture, mais surtout d'injection de transactions USB générique, limitant ainsi le spectre de la recherche de vulnérabilités. Voici ces limites :

- Il n'est pas possible de faire des transferts à haut débit. En effet, l'hôte communique avec le FaceDancer via une liaison série. Ainsi, toutes les trames à envoyer et les trames reçues passent par celle-ci, limitant fortement le débit maximum de capture ou d'injection de transaction USB.

– Le contrôleur USB du FaceDancer ne possède que trois *Endpoints* en plus de l'*Endpoint* par défaut. Cela limite fortement le type de périphérique qu'il est possible d'émuler. Par ailleurs, le FaceDancer ne supporte pas les transferts de type *Isochronous*.

Parallèlement au FaceDancer, nous étudions d'autres possibilités d'injections de trames USB outrepassant les limitations mentionnées ci-dessus. Notamment, nous étudions des cartes de développement à base de processeurs ARM disposant de ports USB 3.x pour *fuzzer* les systèmes USB 1.x, USB 2.x et USB 3.x. Celles-ci sont discutées brièvement à la section 5.

4.3 Avancement du *fuzzer*

L'outil actuel est surtout développé pour une utilisation conjointe avec le FaceDancer. Son utilisation avec Qemu n'en est qu'au stade de preuve de concept. Le *fuzzer* pour FaceDancer, quant à lui, commence à être fonctionnel. Les sources de données peuvent provenir d'une capture des périphériques émulés par Umap ou bien d'une capture provenant de VMware. Il fonctionne pour tout trafic qui ne requiert pas de flux *Isochronous* et pas plus de trois *Endpoints*, et les transferts *Control*, *Interrupt* et *Bulk* sont implémentés. Il peut muter une ou plusieurs trames, et il est possible soit de muter des octets choisis aléatoirement, soit de ne muter que certains octets. Les mutations et les *patterns* peuvent être spécifiés (*bit-flip*, *byte-swap*) au *fuzzer*.

Résultats En lisant les travaux existants, nous n'avons pas trouvé de métriques concernant les performances de *fuzzing*. Nous avons donc décidé de publier les nôtres. En rejouant les 24 premières trames de l'échange entre un hôte et un clavier HID émulé, en *fuzzant* la 25$^{\text{ième}}$ trame (le descripteur HID), nous avons un débit entrant de 70 octets/s et envion 200 octets/s en sortie. Ces mesures correspondent au débit moyens pour un cas de test. Elles comprennent (1) le temps occupé par le *fuzzer* pour créer puis injecter le paquet via le Facedancer, (2) le temps de réponse de l'hôte et (3) la durée du *watchdog* qui determine si l'hôte est vivant ou non.

Ces performances ne semblent pas exceptionnelles. Ce phénomène ne provient probablement pas du fait qu'on utilise une liaison série. Cela est plutôt dû au moniteur (il utilise un *watchdog* pour savoir si l'hôte est toujours vivant) et aux temps de connexion/détection/déconnexion. Nous travaillons actuellement à son amélioration. Les performances de *fuzzing* sous Qemu ne sont pas exposées car elles ne sont pas pertinentes étant

donné que notre fuzzer n'est qu'au stade de preuve de concept et que tout n'est pas encore implémenté.

Le *fuzzing* s'est concentré sur les descripteurs échangés lors de l'énumération du périphérique par l'OS. Nous avons reproduit le *crash* d'Andy Davis sur Windows 8.1 déclenché par un descripteur HID invalide. Nous en avons découvert d'autres qui sont en cours d'investigation.

5 Travaux récemment entrepris et envisagés sur l'USB

Cette section présente quelques travaux que nous avons récemment entrepris, parallèlement aux expérimentations, pour pallier les limitations que nous avons identifiées sur le module d'injection de transaction USB et étendre le *fuzzer* que nous avons déjà implémenté à d'autres plates-formes. Nous avons actuellement deux cibles supplémentaires en vue : la nouvelle pile USB 3.x intégrée au système d'exploitation Windows 8 et la pile USB des systèmes embarqués tels que les smartphones Android au-dessus desquels les constructeurs rajoutent souvent leur propre surcouche.

5.1 Émulation d'un périphérique depuis un système Linux

Le *framework* « USB gadget » dans le noyau Linux a été créée pour faciliter le développement de périphériques USB dans des systèmes GNU/Linux. Cette API, couplée avec un programme en espace utilisateur définissant les actions associées à chaque événement sur le bus USB, permet de mettre en œuvre rapidement des périphériques USB. Il convient de noter que de nombreux systèmes sur étagère ne sont pas en mesure de l'utiliser. En effet, l'utilisation du *framework* nécessite un contrôleur USB en mode esclave, chose que n'ont pas les ordinateurs personnels, les stations de travail ou les serveurs. Cependant, aujourd'hui de plus en plus de plates-formes ARM disposent de contrôleurs USB *On-The-Go* qui peuvent être configurés pour agir en tant que périphériques. Il est, par conséquent, possible d'émuler un périphérique USB sur ce type de système via l'API « USB gadget ».

Ce *framework* permet d'émuler les périphériques simples ou composites, disposant de multiples configurations, classes, etc. Il peut soutenir des vitesses allant au moins jusqu'à l'USB 2.x, et il est possible de définir un nombre arbitraire d'*Endpoints*. Pour ces raisons, nous creusons actuellement cette piste afin de supplanter, lorsque cela est nécessaire, le FaceDancer comme unité d'injection de transactions USB.

5.2 Capture et injection sur les bus USB 3.x SuperSpeed

Les conditions les plus favorables pour effectuer de la capture ou de l'injection de transaction USB consistent à se placer en coupure sur le bus USB. Nous étudions actuellement la possibilité de placer une carte ARM disposant de ports USB 3.x en coupure sur un bus USB SuperSpeed. La motivation principale est de pouvoir évaluer les piles USB 3.x, notamment la nouvelle pile qui a été développée dans Windows 8. Il existe des projets similaires en cours, par exemple le projet Daisho, mais aucun outil ni aucun résultat n'ont été publiés à notre connaissance.

5.3 Travaux futurs

Le *fuzzing* des piles USB n'est pas nouveau en soi, de nombreuses approches sont discutées dans la littérature et de nombreux outils existent. Partant du constat que ces outils sont souvent développés pour une analyse spécifique pour un système précis et que ceux-ci peuvent être adaptés pour répondre à un besoin plus général, notre volonté est de reprendre ces outils, d'en développer de nouveaux lorsque cela est nécessaire, et de les mutualiser au sein d'une boîte à outils pour une infrastructure de *fuzzing*, de façon à être opérationnel rapidement pour une nouvelle analyse de vulnérabilités. La figure 11 présente la boîte à outils pour du *fuzzing* USB à laquelle nous souhaitons arriver à terme. L'idée est de pouvoir faire face à n'importe quelle analyse sur les bus USB. Constituer cette boîte à outils étant un travail de longue haleine, nous l'alimenterons au fur et à mesure des analyses que nous ferons sur des systèmes différents.

Dans cette boîte à outils, nous retrouvons les éléments d'un *fuzzer* de pile USB qui ont été discutés jusqu'à présent. À ceux-ci s'ajoute un *fingerprinting engine* qui servira, d'une part, à identifier le système ciblé et, d'autre part, à délimiter la surface d'analyse. Pour cette dernière, nous pensons émuler des périphériques USB différents (par exemple, en rejouant des trames capturées auparavant) afin d'identifier quelles surcouches au protocole de communication USB (par exemple, périphérique de masse, *Human Interface Device*, etc.) sont supportées et ainsi identifier les vecteurs d'attaques possibles. À notre connaissance, aucun outil n'est actuellement disponible pour cela. Des idées intéressantes ont été présentées par Andy Davis [9].

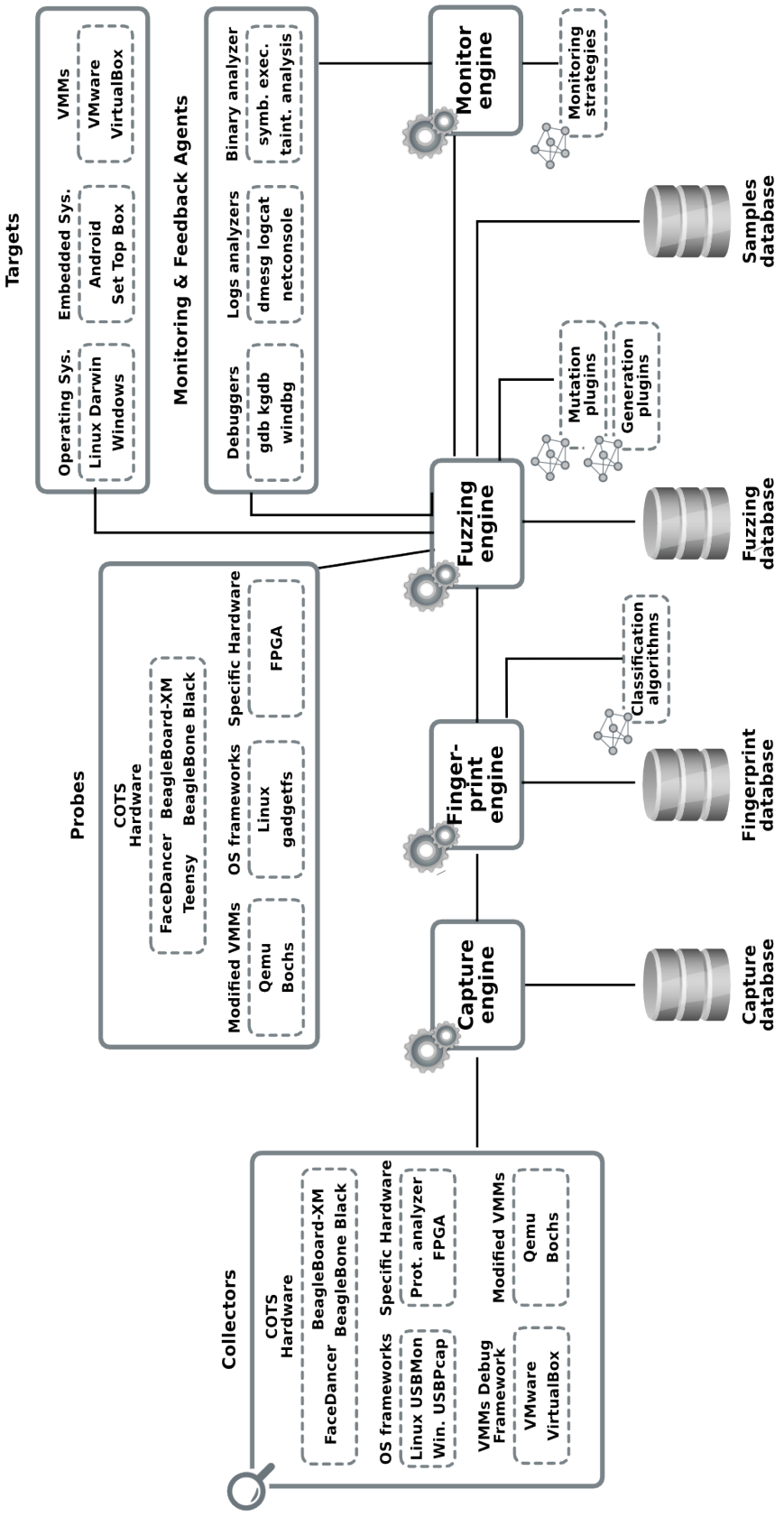

FIGURE 11. Fuzzing du trafic USB redirigé

6 Conclusion

Nous avons présenté, dans cet article, un retour d'expérience sur l'implémentation d'un *fuzzer* pour les piles USB 1.x et USB 2.x de Windows. Nous avons décrit les approches que nous avons suivies, celles qui ont abouties et celles qui sont encore en suspens. Nous faisons actuellement quelques expérimentations sur Windows XP, Windows 7 et principalement sur Windows 8. Les outils employés souffrent de quelques limitations qui réduisent le spectre des recherches de vulnérabilités qu'il est possible de faire. Nous creusons quelques pistes pour rendre plus génériques les éléments limitants de notre architecture, notamment les unités de capture et d'injection utilisées pour étendre ce fuzzer à d'autres parties de Windows (notamment la pile USB 3.x de Windows 8) ainsi qu'à d'autres systèmes, aussi bien dans leurs rôles d'hôte ou de périphérique. Une des pistes prometteuses consiste à capturer et injecter des trames depuis un système embarqué Linux sur une carte de développement ARM placée en coupure entre le système analysé et un périphérique.

7 Remerciements

Nous souhaitons remercier les différents relecteurs, Damien Aumaître ainsi que toute l'équipe de QuarksLab pour leurs conseils et leurs remarques constructives.

Références

1. Dépôt des sources du Facedancer. `https://goodfet.svn.sourceforge.net/svnroot/goodfet`.
2. Dépôt des sources d'Umap. `https://github.com/nccgroup/umap`.
3. Lightweight USB Framework for AVRs. `http://www.fourwalledcubicle.com/files/LUFA/Doc/120219/html/index.html`.
4. Radamsa. `https://code.google.com/p/ouspg/wiki/Radamsa`.
5. Compaq, Microsoft, National Semiconductor. *OpenHCI – Open Host Controller Interface Specification for USB, Release 1.0a*, 14 septembre 1999.
6. Intel Corporation. *Universal Host Controller Interface (UHCI) Design Guide, Revision 1.1*, mars 1996. `http://www.intel.com/technology/usb/spec.htm`.
7. Intel Corporation. *Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0*, 12 mars 2002.
8. Intel Corporation. *eXtensible Host Controller Interface for Universal Serial Bus (xHCI), Revision 1.1*, 20 décembre 2013.
9. Andy Davis. Revealing embedded fingerprints : Deriving intelligence from usb stack interactions. *HITB Magazine, keeping knowledge free*, 4(10).

10. Andy Davis and Lucas Bouillot. Microsoft Windows USB Descriptor CVE-2013-3200 Local Privilege Escalation Vulnerability. http://www.symantec.com/security_response/vulnerability.jsp?bid=62823.
11. Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.stuxnet dossier, version 1.4. Technical report, Symantec Security Response, Cupertino (CA, USA), February 2011. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.
12. Texas Instruments. MSP430 Overview. http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/overview.page.
13. Maxim Integrated. MAX3421E Overview. http://www.maximintegrated.com/datasheet/index.mvp/id/3639.
14. The iPhone Wiki. usb_control_msg(0x21, 2) Exploit. http://theiphonewiki.com/wiki/Usb_control_msg(0x21,_2)_Exploit.
15. Petr Matousek. CVE-2013-1860 kernel : usb : cdc-wdm buffer overflow triggered by device. https://bugzilla.redhat.com/show_bug.cgi?id=921970.
16. Tobias Mueller. Virtualised usb fuzzing using QEMU and Scapy - breaking usb for Fun and Profit. *Ekoparty, 9th edition*, September 2011. School of Computing, Dublin City University.
17. Fabien Perigaud. Découverte et exploitation d'une vulnérabilité dans la pile USB de Windows XP. *MISC 71*.
18. PSGroove. PSGroove. https://github.com/psgroove/psgroove.
19. USB Implementers Forum, Inc. *Universal Serial Bus Specification, Revision 1.1*, 23 septembre 1998.
20. USB Implementers Forum, Inc. *Universal Serial Bus Specification, Revision 2.0*, 27 avril 2000.
21. USB Implementers Forum, Inc. *On-The-Go Supplement to the USB 2.0 Specification, Revision 1.0*, 18 décembre 2001.
22. USB Implementers Forum, Inc. *Universal Serial Bus 3.0 Specification, Revision 1.1*, 26 juillet 2013.
23. Rafael Dominguez Vega. USB Fuzzing for the Masses. MWR LABS, July 2011. https ://labs.mwrinfosecurity.com/blog/2011/07/14/usb-fuzzing-for-the-masses/.