SASUSB : protocole sanitaire pour l'USB

Fabrice Desclaux et Louis Syoën
fabrice.desclaux@cea.fr
louis.syoen@cea.fr

Commissariat à l'énergie atomique et aux énergies alternatives

Résumé. Le SASUSB est un outil permettant la lecture de supports de stockage USB suivant les principes de défense en profondeur et de moindre privilège, le but étant de réduire la surface d'attaque et les conséquences d'une exploitation de faille. Les données peuvent être transférées sur un autre support de confiance ou importées dans le système d'information après analyse antivirale. Il est écrit en Rust et s'inspire du fonctionnement des micro noyaux.

1 Introduction

Les menaces liées à la connexion de périphériques USB potentiellement malveillants sur un système d'information sont multiples. La surface d'attaque relative à ce protocole est d'autant plus grande qu'il supporte une quantité faramineuse de périphériques différents (clef USB, disque dur, caméra, carte réseau, etc.). On pense notamment à l'introduction de virus / ransomware, exploitation de la pile USB, des parseurs de systèmes de fichiers, ainsi qu'aux périphériques de type BadUSB. Les ports USB sont présents sur quasiment tous les composants communément utilisés en entreprise : ordinateurs (fixes ou portables), serveurs, téléphones,... Lorsqu'un utilisateur branche une clef USB non maîtrisée sur son poste de travail, n'importe quel *driver* USB peut être chargé par le noyau et servir de porte d'entrée. Nous en avons encore eu la démonstration récemment ici [12] ou là [8].

Nous allons dans un premier temps tenter de dessiner la carte de la surface d'attaque liée au protocole USB ainsi qu'aux différentes couches logicielles relatives au transfert de données via ce protocole. Dans un second temps nous illustrerons certains écueils dans lesquels tombent les stations blanches / SAS d'import de données actuellement proposés dans le commerce ou open source. Enfin nous décrirons l'architecture de l'outil présenté ici : le SASUSB, qui tente de répondre aux menaces précédemment évoquées.

1.1 Définitions

Nous reprenons ici les définitions du guide de l'ANSSI : *Profil de fonctionnalités et de sécurité – Sas et station blanche (réseaux non classifiés)* [11]. Nous noterons cependant que les travaux présentés ici ont commencé avant la publication de ce guide et que, même si nombre de nos choix concordent a posteriori, notre solution n'est pas calquée sur ce guide.

Station blanche : Poste de travail ou serveur isolé du réseau opérationnel, dédié à l'analyse antivirale des médias amovibles et des données qui y sont stockées. Ce dispositif donne des garanties raisonnables quant à l'innocuité du média amovible et des données transférées vers le réseau opérationnel.

Sas d'import de données : Association d'une station blanche et d'un point d'insertion de données. La station blanche et le point d'insertion de données sont physiquement cloisonnés. Ce dispositif, interconnecté au réseau opérationnel, garantit l'innocuité du média amovible et des données transférées à destination de ce réseau.

1.2 Surface d'attaque

Le protocole USB permet de connecter un certain nombre de périphériques appelés *devices* à une machine maître, appelée *host*. Si on le compare au réseau, il assure la couche physique et un mélange de couches Ethernet / IP / TCP. Lorsqu'un périphérique est branché et détecté par l'hôte, ce dernier lui attribue une adresse. Chaque périphérique se voit attribuer une adresse différente. La machine hôte possède au moins un contrôleur USB, relié à un *hub* USB permettant l'attachement de plusieurs périphériques. Il est possible de brancher un second *hub* sur le premier, augmentant ainsi le nombre de ports USB d'une machine, on parle de topologie USB en étoile.

Physique C'est le contrôleur USB de l'hôte qui reçoit et traite les paquets reçus des périphériques. Ces paquets sont découpés en plusieurs champs, contenant l'adresse source du périphérique, le type de paquet, des numéros de séquence, la longueur des données embarquées, des CRC,... Cette liste n'est pas exhaustive et bien qu'elle attire l'œil pour un attaquant, nous n'avons pas connaissance de vulnérabilités exploitées à ce niveau.

Néanmoins, Intel a pris la décision il y a quelques années d'utiliser un des ports physiques USB comme interface JTAG sur le CPU. Cette interface permet de debugger au niveau *hardware* le CPU de la machine

en question. Certaines protections sont tout de même mises en place et sont exposées dans la conférence *Tapping into the core* [7, 13]. Un debugger *hardware* est un cas de figure assez délicat car il n'offre pas de contre-mesures permettant de limiter son impact.

Communication des hub USB Les contrôleurs USB sont en général directement liés à un hub USB permettant à la machine hôte de gérer plusieurs ports USB physiques. Un hub dispose d'un port *upstream* connecté directement au contrôleur USB hôte ainsi que plusieurs ports *downstream* pour connecter des périphériques. Les données reçues sur le port *upstream* sont *broadcastées* à tous les périphériques connectés aux ports *downstream*. Les données reçus sur un port *downstream* ne sont transmises qu'au port *upstream*. Ce mode de fonctionnement pose problème car un périphérique peut recevoir des données qui ne lui sont pas directement adressées et un périphérique malveillant pourrait effectuer une attaque de type *eavesdropping* [15].

Notons que la norme USB 3.0 ajoute un mode point à point aux transmissions *downstream* pour pallier ce problème.

Protocoles USB Le protocole USB permet de lire la configuration des périphériques. Celle-ci décrit entre autres le type de périphérique qui est branché, appelé *classe* qui peut être mass storage, HID (Human Interface Device), printer,... En analysant cette réponse, le système d'exploitation reconnaît la nature du périphérique et, s'il en est doté, va charger le *driver* associé. Il y a ici deux grands cas de figure, il s'agit soit :

— d'un *driver* générique, qui fonctionne pour toute une classe donnée ;
— d'un *driver* spécifique pour un périphérique bien défini (souvent lié au vendorID/productID du périphérique USB).

Il est important de noter qu'une clef USB n'est pas une clef USB de par sa forme, mais par la définition qu'elle envoie à l'hôte. Un périphérique tel qu'une clef USB ou une souris embarque un composant électronique qui s'occupe de communiquer en USB avec l'hôte, et de traduire les requêtes de l'hôte en requêtes pour la flash embarquée ou son système de positionnement de souris. Il a d'ailleurs déjà été montré que certains fabricants utilisent des micro-contrôleurs reprogrammables pour assurer cette fonctionnalité. La démonstration en a été faite en 2014 [16]. La clef USB connectée se décrit comme un clavier, une fois reconnue et ajoutée par l'OS, elle simule l'appui de touches pour écrire un script Powershell et déclenche ainsi une exécution de code depuis le compte de l'utilisateur.

Des manipulations du même genre peuvent être réalisées sur des cartes de développement USB, coûtant autour de 10$ (AT90USBKEY, Raspberry Pi,...). Ces prix dérisoires rendent l'attaque à portée de toutes les mains.

On notera qu'un même périphérique USB peut proposer plusieurs interfaces en même temps. Il peut par exemple présenter une carte réseau, un périphérique *mass storage* ainsi qu'un clavier/souris. Il peut alors copier des fichiers de la machine et les coller dans le disque fraîchement monté par le système (ou les partager sur le réseau).

Ce genre d'attaque s'effectuait jusqu'ici "à l'aveugle", le périphérique malveillant n'avait pas de retour sur ses actions. Cependant, il est maintenant possible avec les nouveaux connecteurs USB type C de connecter un moniteur, l'attaque peut se baser sur les retours de l'interface graphique pour plus de fiabilité [18].

Mass Storage Lorsqu'un périphérique de type *mass storage* (une clef USB, un disque dur) est connecté sur la machine hôte, le *driver* responsable de ces matériels est chargé. Pour communiquer, deux *endpoints* sont utilisés : un pour transférer les données montantes, l'autre pour les données descendantes. Un *endpoint* est une unité virtuelle à laquelle est associée une direction (*up*/*down*) qui se rapproche un peu d'un port TCP. Un périphérique *mass storage* est accédé en utilisant le protocole SCSI (*Small Computer System Interface*) au dessus des données transportées par USB. Un périphérique *mass storage* implémente des fonctions de base SCSI :

— lire N secteur(s) à tel offset ;
— écrire N secteurs(s) à tel offset ;
— énumération des LUNs (*Logical Unit Number*) ;
— ...

Les codes responsables de la gestion du protocole SCSI font partie intégrante de la surface d'attaque, d'autant que ce sont en général des *drivers* et qu'ils disposent donc d'une exécution privilégiée.

Systèmes de fichiers Le protocole SCSI permet d'exposer un *block device* au système d'exploitation. Lorsqu'un périphérique de stockage est branché, l'OS lit dans un premier temps la table des partitions (qui fait donc elle aussi partie de la surface d'attaque), puis monte le système de fichiers.

Il existe de nombreux systèmes de fichiers (*FAT*, *NTFS*, *EXT4*,...), leurs relatives complexités peut mener à des failles de sécurité (par exemple pour NTFS : CVE-2020-17096 [3] RCE, CVE-2021-28312 [5] DoS, CVE-2021-31956 [6] PrivEsc). La menace est d'autant plus grande si le système

d'exploitation supporte un nombre important de systèmes de fichiers différents, un système de fichiers *exotique* et/ou non maintenu peut être une porte d'entrée intéressante pour un attaquant. Les codes responsables de la gestion des systèmes de fichiers sont en général des *drivers* et s'exécutent donc en mode privilégié.

Fichiers spéciaux Certains fichiers peuvent subir un traitement particulier par le système d'exploitation, cela va du fichier *.lnk* sous Windows, aux fichiers cachés permettant de sauver la configuration d'un répertoire, en passant par les aperçus d'images ou les icônes ainsi que le fichier *autorun.inf* à la racine d'un périphérique de stockage qui peut déclencher l'exécution automatique d'un programme à la connexion de ce périphérique.

Ces fichiers sont sensibles car le simple fait d'accéder à un répertoire avec l'explorateur de fichiers peut déclencher leur traitement et une éventuelle vulnérabilité (on pense notamment à [1]). Leur filtrage est relativement difficile car intimement lié au mécanisme du système d'exploitation ou de l'explorateur de fichiers.

Mélange des entrées Il y a quelques années, les postes étaient équipés de ports PS2 pour connecter les claviers/souris. Ils ont aujourd'hui laissé place aux ports USB, ce qui signifie que du point de vue du système d'exploitation, le même protocole est utilisé pour les entrées de l'utilisateur et le traitement d'une clef USB potentiellement malveillante.

La classe USB utilisée pour ces périphériques est la classe HID (*Human Interface Device*), elle sert aussi pour les joysticks, les pointeurs de tablettes, les écrans tactiles,... Pour gérer ce grand nombre de périphériques, elle se base sur l'utilisation de structures permettant à un périphérique de lister tous ses moyens de mesure (boutons, molette, axes, ...), leur précision (8 bits signés, 3 bits non signés, ...) ainsi que leurs limites mécaniques. En bref, un nid de problèmes potentiels [4]. Le code responsable du HID est en général exécuté en mode privilégié. Une clef USB malveillante présentant une classe HID peut mener à une attaque de type BadUSB précédemment citée.

Stations blanches Il existe plusieurs manières d'implémenter une station blanche, certaines se contentent d'analyser les fichiers stockés sur les périphériques USB, le scénario est le suivant :

1. L'utilisateur insère une clef USB sur la station.

2. La station *nettoie* la clef USB.
3. L'utilisateur récupère la clef USB.
4. L'utilisateur branche la clef sur son poste et récupère ses données.

Ce mode de fonctionnement pose problème car, comme nous l'avons vu, le périphérique USB décide ce qu'il expose en terme de classe de périphérique, mais également en terme de données. Une *race condition* existe entre l'étape 2 et l'étape 4, un périphérique malveillant pourrait exposer un système de fichiers ne contenant que des fichiers sains lors de l'analyse antivirale et présenter un autre système de fichiers contenant des fichiers infectés sur un poste de travail. Une pile USB, à la manière d'une pile IP, peut fonctionner différemment d'une machine ou d'un OS à l'autre, il est ainsi possible pour un périphérique de détecter qu'il est connecté à une station blanche, une machine Linux ou Windows etc. et ainsi adapter son comportement.

Ce type d'attaque *TOCTOU* (*time-of-check-time-of-use*) / *double read* peut aussi se retrouver lors de la vérification de signatures numériques. Imaginons que l'import de données (ou la mise à jour de la station blanche par clef USB) soit soumis à la vérification d'une signature numérique et que cette vérification est effectuée par la station, directement depuis la clef USB. Un périphérique malveillant pourrait présenter lors de la première lecture un fichier validant la signature mais un fichier totalement différent lors de la deuxième lecture pour l'import de données (ou l'application de la mise à jour, voir aussi [2]).

Enfin, pour des raisons de commodité, la station blanche peut utiliser un navigateur en guise d'interface graphique. Un soin particulier doit donc être apporté au traitement des données affichées (noms des fichiers, métadonnées,...), ces dernières sont contrôlées par l'attaquant et doivent être maîtrisées afin d'éviter une injection de code javascript.

1.3 État de l'art

Nous verrons dans cette partie deux solutions open source apportant une réponse aux menaces évoquées précédemment : *CIRCLean* et *Le Guichet*. Il existe dans le commerce plusieurs autres solutions, il est cependant difficile de connaitre leur fonctionnement interne et leurs spécificités sans documentation approfondie. Nous avons pu étudier deux de ces solutions commerciales dont nous détaillerons les points négatifs sans trop en dire ni les nommer, on les appellera *Station A* et *Station B*.

Circlean *CIRCLean* [9] est une station blanche (selon la définition en 1.1) open source fonctionnant sur Raspberry Pi, développée par *The Computer Incident Response Center Luxembourg* (CIRCL) permettant le transfert de fichiers entre deux périphériques de stockage USB. Les fichiers sont analysés par l'antivirus ClamAV et filtrés par leurs type MIME. Certains types de fichiers (PDF, Microsoft Word, ...) sont convertis en HTML.

`Avantages` :

— Utilisation de deux clefs USB différentes (une en entrée potentiellement infectée, une en sortie de confiance)
— Analyse antivirale
— Vérification / conversion des fichiers jugés dangereux

`Inconvénients` :

— Tous les fichiers sont copiés (pas de sélection)
— Utilisation des modules noyaux (privilégiés) pour l'USB, SCSI et systèmes de fichiers

Le Guichet *Le Guichet* [14] est une station de décontamination (ou SAS d'import de données selon 1.1) open source permettant de transférer des fichiers de manière sécurisée. Il offre deux modes de fonctionnement :

— *Gateway* : transfert depuis un réseau non sûr vers un réseau sécurisé
— *USB mode* : transfert d'un périphérique USB non sûr vers un périphérique USB de confiance

Il est écrit en Rust et un soin particulier a été porté au durcissement de la machine : utilisation de *Seccomp*, sandboxing *systemd*, *AppArmor*, testé avec les patchs noyaux *grsecurity* ...

`Avantages` :

— Analyse antivirus (*ClamAV*)
— Filtrage des fichiers par règles *YARA* personnalisables
— Filtrage des fichiers selon leurs *magic number*
— Filtrage des périphériques USB par leurs numéros de série
— Signature des périphériques de sortie

`Inconvénients` :

— Utilisation des modules noyaux (privilégiés) pour l'USB, SCSI et systèmes de fichiers

Notons que nous n'avons eu connaissance de ce projet que récemment (notre solution était déjà bien avancée) et qu'il est encore en développement.

Station A Cette première solution commerciale est une borne de décontamination (station blanche selon 1.1) sur laquelle les utilisateurs branchent

leurs clefs USB pour analyse antivirale avant de la brancher sur leurs postes. L'analyse de la station a révélé les points suivants :

- La station se contente d'analyser les fichiers sur la clef USB, nous avons vu précédemment en quoi ce mode de fonctionnement (à une clef) est problématique (présentation d'un système de fichiers sain si détection de la station) voir 1.2
- La station est basée sur la distribution Linux Ubuntu, la chaîne de démarrage n'est pas vérifiée ni authentifiée (pas de secure boot), le disque dur n'est pas chiffré, le branchement des périphériques HID (dont les claviers) n'est pas filtré et le menu du chargeur d'amorçage (*GRUB*) n'est pas verrouillé (il n'apparait pas par défaut mais il est possible de le déclencher en appuyant sur les touches *Ctrl+Alt+Del* pendant le démarrage). La combinaison de ces éléments fait qu'il est trivial de prendre le contrôle de la station en mode administrateur (par exemple en modifiant la *cmdline* du noyau) avec un simple clavier ou un périphérique de type BadUSB.
- Aucun *driver* USB n'est filtré, laissant une large surface d'attaque. Par exemple : la station monte une nouvelle interface réseau au branchement d'une carte réseau USB. Ceci est d'autant plus problématique que la station a la possibilité de fonctionner en mode connecté au réseau de l'entreprise (pour le monitoring et l'administration).
- Il n'y a pas d'analyse antivirus sur les fichiers trop volumineux.
- Les clefs USB sont montées automatiquement par le noyau et les modules (privilégiés) pour l'USB, SCSI et les systèmes de fichiers sont utilisés.
- La station peut être mise à jour par clef USB. Le fichier de mise à jour doit être signé avec PGP. Cependant, la vérification de la signature s'effectue directement sur la clef, ce qui rend la procédure vulnérable à l'attaque de type *TOCTU* évoquée précédemment.

Cette station de décontamination peut donc facilement devenir une station contaminante ainsi qu'un point d'entrée dans le réseau de l'entreprise.

Station B Cette seconde solution commerciale est similaire à la première, elle nécessite cependant deux périphériques USB (un en entrée, un en sortie). Lors du transfert, les fichiers sont analysés par plusieurs antivirus. Une diode *logicielle* est implémentée entre un guichet bas (pour la lecture du périphérique d'entrée) et un guichet haut pour l'analyse et l'écriture sur le périphérique de destination. La communication se fait via un protocole

maison sur UDP et les deux guichets sont exécutés dans des machines virtuelles différentes. Un serveur et un client web font office d'interface graphique. L'analyse de la station a révélé les points suivants :

— L'interface graphique est réalisée en web qui est sujet à des XSS. Ces XSS permettent de manipuler des vues offertes par le serveur. L'attaquant peut ainsi lister, télécharger ou envoyer n'importe quel fichier arbitraire. Ceci mène à l'exécution d'un shell root.

— Le serveur web exécute des binaires natifs lors du listing des fichiers de la clef d'entrée (pour récupérer la taille,...). Avec un nom de fichier particulier, l'attaquant peut profiter d'une injection de code shell et avoir directement la main dans le guichet bas.

— Les informations [1] remontées du guichet bas vers le guichet haut sont sérialisées. Le programme du guichet haut recevant ces données est sujet à un buffer overflow dans la partie décodant la taille de nom de fichier. Malheureusement, ce binaire n'est pas compilé avec les options classiques de protections de débordement de tampon, la prise de contrôle du guichet haut est alors possible en utilisant une attaque de type ROP (la position du binaire n'est pas randomisée).

Bien que l'exploitation de cette seconde station ne soit pas aussi triviale que la précédente, la compromission reste totale. La station décontaminante peut, au simple branchement d'une clef USB préparée spécialement, devenir une station contaminante.

2 Solution proposée

Ce chapitre est consacré à la description de l'architecture et aux choix qui ont été faits.

Pour répondre aux problématiques précédemment évoquées et en prenant en compte les écueils des solutions étudiées, nous avons développé le *SASUSB* : un logiciel fonctionnant sous Linux permettant de transférer de manière sécurisée les fichiers d'un périphérique de stockage potentiellement malveillant vers un périphérique de confiance ou vers une machine distante, après analyse antivirale. Cette solution permet (entre autres) le développement d'un SAS d'import de données (1.1).

Le *SASUSB* s'inspire de l'architecture des micro noyaux. Plusieurs processus s'exécutent en espace utilisateur et chacun s'occupe d'une tâche simple : chaque couche (USB, SCSI et système de fichiers) est traitée par un de ces processus. Ces processus communiquent par un IPC utilisant

[1] noms de fichiers, métadonnées ainsi que leurs contenu

un mécanisme qui sera décrit dans la suite. Ceci donne l'architecture en figure 1 :

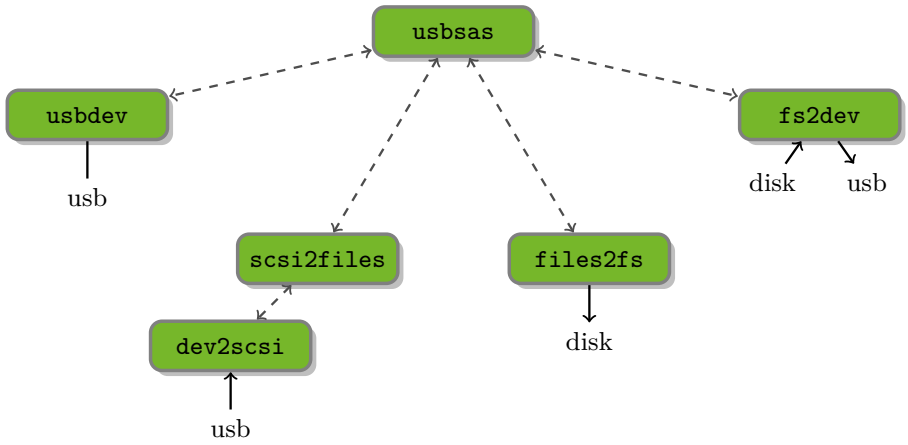

Fig. 1. Schéma simplifié des processus composant le *SASUSB*

2.1 Système

Drivers Afin de réduire la surface d'attaque, *tous* les *drivers* USB du noyau Linux sont désactivés/supprimés, y compris le module USB HID responsable des claviers/souris. Nous ne gardons que le composant "core USB", responsable de la gestion du contrôleur USB physique. Ceci permet de limiter la surface d'attaque liée au support de multiples *drivers* décrite en 1.2. Le *SASUSB* n'est pas lié à un matériel particulier, ce qui signifie qu'il peut s'exécuter en utilisant différents contrôleurs USB. Cette partie noyau est donc conservée et permet d'envoyer et de recevoir des paquets USB aux périphériques, ainsi que de leur assigner des adresses. Notons une autre conséquence : le noyau enverra un premier paquet de demande de description de device permettant de récupérer le strict minimum d'informations (vendorID/productID/USBClass). La demande émise par le noyau est dans le cas général en deux parties :

— une première requête est opérée, forçant une taille minimale pour la réponse permettant seulement de connaître le strict minimum sur le device ainsi que de récupérer la taille de la structure complète
— une deuxième requête, identique, à l'exception de la taille qui est fixée à celle précédemment demandée par le client

Dans notre cas, seule la première requête est envoyée par le noyau. La suite sera gérée en espace utilisateur par le code du *SASUSB*.

Userland Pour limiter le risque lié à une vulnérabilité, les composants qui traiteront les données seront exécutés en *userland* (espace utilisateur), à la façon des micro noyaux, par un utilisateur non privilégié. Pour ce faire, une règle *udev* permet de donner les droits nécessaires à un *user* particulier permettant d'accéder en brut à tout périphérique USB branché. On communiquera alors en utilisant la *libUSB* avec ces derniers.

HID Comme nous le disions précédemment, le système est dépourvu de *driver* HID. Pour palier ce problème, nous avons développé un gestionnaire minimaliste de périphérique HID en *userland*. Ce dernier n'implémente que le minimum de la norme pour supporter les souris et les écrans tactiles prévus. La machine ne supporte donc pas les claviers. Ce dernier *poll* les périphériques comme le ferait le *driver* original, récupère les informations de positionnement et les applique au serveur X en simulant les mouvements associés. Il est écrit en Rust.

La configuration du *SASUSB* permet également de définir un numéro de bus USB ainsi qu'un numéro de port physique sur lequel seront autorisés les périphériques de type HID. Ainsi, seul ce port USB physique sera pris en compte par le gestionnaire HID décrit ci-dessus. L'administrateur pourra par exemple dédier un port physique de l'arrière de la machine à la souris, et utiliser les ports physiques de façade pour l'insertion de clef USB. Dans ce cas, même si la clef USB se fait passer pour une souris ou autre périphérique HID ou si elle expose plusieurs interfaces USB, elle ne sera pas prise en compte par la machine. Ceci permet de répondre aux problématiques de faux périphériques USB tentant d'interférer par le biais de ce protocole (voir 1.2)

Notons que même si un périphérique USB tente d'écouter le flux descendant (problématique d'*eavesdropping*, voir 1.2) des autres périphériques, il n'apprendra rien d'intéressant car le seul flux descendant correspond à l'écriture des fichiers qui proviennent de lui-même (périphérique d'entrée).

Nous le verrons plus tard, l'interface finale est accessible via une API web. Il est tout à fait envisageable d'utiliser le *SASUSB* dans une configuration dépourvue de HID. Le *SASUSB* est alors simplement une machine recevant les clefs USB et l'interaction avec l'utilisateur se fait alors à travers cette API web, par le réseau depuis le poste client. Ici le *SASUSB* n'a ni écran, ni clavier ni souris.

2.2 Séparation des privilèges, moindre privilège

Afin de réduire au maximum les privilèges, nous l'avons dit, le code relatif à la communication avec les périphériques USB et le traitement des données qu'ils contiennent s'exécute en espace utilisateur.

Afin de cloisonner ces tâches, elles sont séparées en plusieurs processus (chacun correspondant peu ou prou à une couche de la pile USB).

USB / SCSI Un premier processus a accès au périphérique USB brut en utilisant la *libUSB* à travers un fichier spécial (par exemple */dev/bus/USB/001/002*, qui correspond au périphérique USB branché sur le bus numéro 1, et qui répond à l'adresse de device USB 2). Ce processus parle le protocole *SCSI* qu'il encapsule dans les paquets USB. Il expose une interface permettant d'effectuer les opérations suivantes :

— demander la taille totale du volume ;
— demander la taille des secteurs ;
— lire N secteurs à l'offset O.

Il est écrit en Rust.

SCSI / Systèmes de fichiers Un autre processus s'occupe de l'interprétation du système de fichiers. Il communique avec le processus précédent, ce qui lui permet de lire n'importe quel secteur du périphérique de stockage d'entrée. Il est capable de lire plusieurs systèmes de fichiers :

— *FAT*
— *exFAT*
— *NTFS*
— *ext4*
— *ISO9660*

Il expose une nouvelle interface, permettant les opérations suivantes :

— lister un répertoire ;
— récupérer les attributs d'un fichier / répertoire ;
— récupérer le contenu d'un fichier.

Ce programme est écrit en Rust, ainsi que les parseurs de systèmes de fichiers à l'exception de *FAT*/*exFAT* ou nous utilisons la bibliothèque *ff* écrite en C. Si une vulnérabilité est exploitée, l'attaquant sera à la place de ce processus. Il pourra commencer un pivot à partir de là. Ce problème doit être pris en compte dans notre architecture.

Communication Chaque processus expose une interface de communication. Celle-ci se traduit concrètement par deux descripteurs de fichiers

communs entre deux processus. Le premier permet de recevoir des données du processus A vers le processus B, le deuxième de réaliser l'opération inverse. Les communications se font sous forme de requêtes/réponses. Pour sérialiser les données échangées, Protobuf est utilisé. Ce dernier a l'élégance de générer le code responsable de la sérialisation/désérialisation des données à partir d'un fichier de spécification.

Les fonctions de communication sont écrites en Rust.

Cloisonnement L'une des raisons pour lesquelles nous avons séparé les différentes couches de la pile USB en processus distincts était de pouvoir leur appliquer un filtre *Secure Computing Mode* (*seccomp*) respectif. *seccomp* est un mécanisme du noyau Linux permettant à un processus de se verrouiller de manière non réversible dans un état dans lequel il ne pourra effectuer qu'un certain nombre d'appels systèmes prédéfinis. Il sera tué par le noyau s'il tente d'effectuer un appel système non autorisé.

Quelques exemples de règles appliquées :

- processus USB/scsi
  - opérations sur le *filedescriptor* associé au périphérique USB ;
  - mmap filtré pour autoriser les allocations (sans rwx) ;
  - lecture sur le *filedescriptor* de réception sur l'interface publiée ;
  - écriture sur le *filedescriptor* d'envoi sur l'interface publiée ;
  - ...
- processus scsi/système de fichiers
  - lecture sur le *filedescriptor* de réception sur l'interface avec le processus USB/scsi ;
  - écriture sur le *filedescriptor* d'envoi sur l'interface avec le processus USB/scsi ;
  - mmap filtré pour autoriser les allocations (sans rwx) ;
  - lecture sur le *filedescriptor* de réception sur l'interface publiée ;
  - écriture sur le *filedescriptor* d'envoi sur l'interface publiée ;
  - ...

Ainsi, si un des processus est corrompu, l'attaquant est limité dans ses mouvements. Par exemple s'il corrompt le processus de gestion du système de fichiers, il peut envoyer des données contrôlées sur l'interface publiée d'accès au système de fichiers. Interface qui doit être sérialisée en Protobuf. Du point de vue du processus client de cette interface, cela ne change rien : l'attaquant pourra choisir des noms de fichiers exotiques, des droits ou des tailles de fichiers farfelus. On peut noter que ceci aurait déjà pu être le cas en modifiant directement le système de fichiers sur la

clef USB. Le gain est nul pour l'attaquant de ce point de vue. Notons que tous les processus sont tués et relancés entre chaque transfert.

L'attaque est tout de même utile dans d'autres scénarios : depuis l'intérieur de la sandbox, l'attaquant pourra tenter une évasion de *seccomp* en utilisant la vulnérabilité *rowhammer* si le matériel y est sensible. L'attaquant pourra également tenter d'accéder à la mémoire hors de la *sandbox* via l'exploitation de vulnérabilité sur les caches de la machine. L'administrateur du *SASUSB* devra là aussi faire une configuration permettant de répondre à ce genre d'exploitations.

2.3 Gestion des processus

Les différents processus composant les *SASUSB* sont lancés et orchestrés par un processus père, il est lui aussi sous *seccomp* et communique avec ses enfants via *protobuf*. Ce processus expose une interface de communication à destination de l'application finale. N'importe quel programme parlant *protobuf* peut ainsi utiliser les mécanismes offert par le *SASUSB*. Notons ici l'avantage de *protobuf* par rapport à *serde* qui est la référence Rust en terme de sérialisation/désérialisation, et qu'il existe de nombreuses bibliothèques dans différents langages capables de générer du code *protobuf*, ne limitant pas l'application finale au langage Rust.

2.4 Traitement des données

Nous l'avons vu, le *SASUSB* fait office de point d'insertion de données au sens du guide de l'ANSSI [11].

Le choix a été fait de ne pas mettre d'antivirus directement sur la machine. Le grand nombre de *SASUSB* dispersés dans l'entreprise tend à garder le prix unitaire des *SASUSB* relativement bas et donc avec des capacités de calcul limitées. Ces capacités seraient trop modestes pour permettre de faire tourner des machines virtuelles contenant plusieurs antivirus différents sur un *SASUSB*.

Le *SASUSB* offre cependant la possibilité d'uploader les fichiers sélectionnés (stockés dans une archive intermédiaire du transfert) sur un serveur distant pour analyse antivirale. Le résultat est attendu sous forme de JSON décrivant l'état de chaque fichier (sain/infecté).

Les administrateurs peuvent faire le choix d'implémenter un serveur d'analyse maison basée par exemple sur une installation de Irma [17], ou de faire une petite adaptation pour utiliser des services externes clef en main comme VirusTotal ou équivalents.

Cette fonctionnalité peut également être utilisée pour séquestrer les fichiers dans le but d'offrir un historique lors d'analyses post mortem.

Après réception des résultats antiviraux, deux cas de figure se présentent :

— La destination du transfert est une autre clef USB : dans ce cas, un autre processus lit cette archive et génère un système de fichiers complet en y incluant seulement les fichiers sains. La clef USB de sortie est donc effacée et possède une nouvelle table de partition et un système de fichiers contenant uniquement les fichiers sélectionnés par l'utilisateur et ayant passé l'antivirus.

— La destination du transfert est un serveur distant : dans ce cas une archive contentant les fichiers sains est uploadée vers ce serveur (qui pourra, par exemple, envoyer les fichiers par mail à l'utilisateur).

Notons que dans les deux cas, l'attaque du *double read* décrite en 1.2 est évitée puisque les données sont d'abord récupérées depuis la clef USB, puis sont traitées dans un second temps, la clef USB n'est donc lue qu'une fois. En outre, les métadonnées peuvent être modifiées par une clef USB malveillante : les tailles de fichiers sont affichées à l'utilisateur mais le contenu des fichiers n'est copié qu'une fois la demande effectuée.

2.5 Interfaces clientes

Pour utiliser les fonctionnalités du *SASUSB*, une application cliente doit donc communiquer (via *protobuf*) avec le processus père évoqué précédemment. Nous verrons ici l'application principale (un serveur WEB) ainsi que d'autres exemples.

API web L'utilisation principale du *SASUSB* se fait par l'intermédiaire d'un client et d'un serveur WEB. Ce dernier, écrit en Rust, expose une API WEB qui offre les fonctionnalités suivantes :

— Pour effectuer un transfert :
  — Récupération des périphériques USB de type *mass storage* connectés au *SASUSB*
  — Sélection d'un périphérique d'entrée
  — Sélection d'une destination (USB ou upload vers un serveur)
  — Lecture des partitions du périphérique d'entrée
  — Sélection d'une partition
  — Parcours de l'arborescence du système de fichiers
  — Copie d'une liste de fichiers vers la destination choisie
— Formatage d'un périphérique USB (avec ou sans effacement préalable)

— Faire une image d'un périphérique USB (pour analyse ultérieure par un administrateur)
— Réinitialisation du *SASUSB* (nécessaire entre chaque transfert)

La partie cliente utilisant l'API du serveur a été développée pour *NW.js*, une plateforme basée sur *chromium* permettant l'implémentation d'applications de bureau à partir de technologies WEB. Il a l'avantage de fournir un mode *kiosque* (l'utilisateur ne peut sortir de la fenêtre affichée).

La figure 2 décrit l'architecture complète des processus du *SASUSB*, l'annexe A représente l'interface graphique du client WEB.

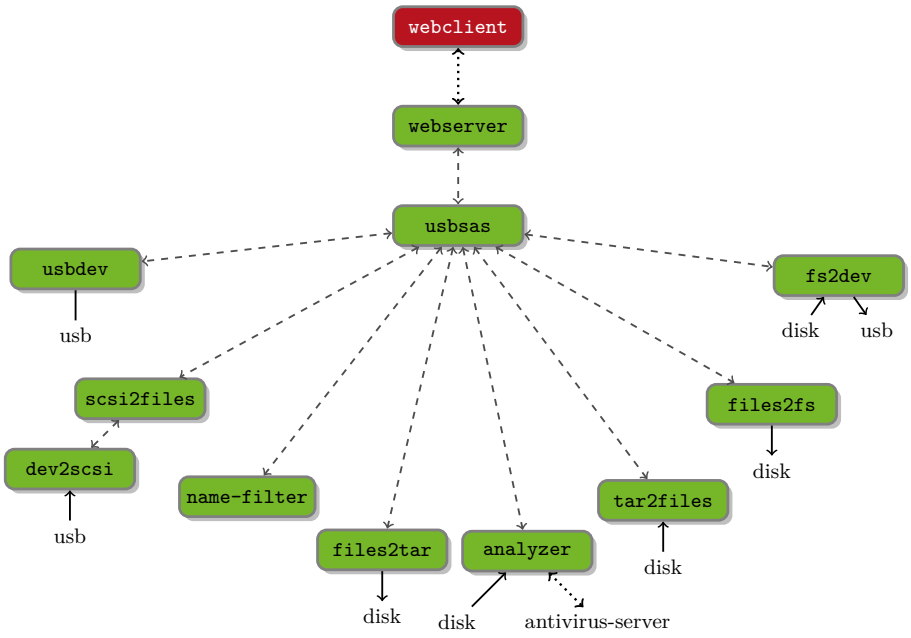

Fig. 2. Schéma complet des processus composant le *SASUSB*

Fuse La séparation des processus du *SASUSB* permet d'implémenter différentes applications utilisant ce dernier à moindre coût. Pour illustrer cette affirmation nous avons implémenté un système de fichiers virtuel utilisant la *libFUSE* (*Filesystem in UserSpacE*) et les briques du *SASUSB*. Il permet, sous Linux, de monter un périphérique de stockage USB en lecture seule (les modules noyaux évoqués précédemment sont toujours désactivés/supprimés).

2.6 Hardening système

Le présent article se focalise sur la description du logiciel en lui-même, cependant son déploiement sur une machine servant de station d'import de données doit immanquablement s'accompagner de pratiques de durcissement adaptées, par exemple :

- utilisation d'une machine permettant de fermer et sceller l'accès aux connecteurs Ethernet, alimentation, écran (tactile), lecteur de cartes à puce,... ;
- utilisation de *Secure Boot*, chiffrement du disque dur par un secret du *TPM* pour un démarrage sécurisé et vérifié ;
- *UEFI* protégé, pas de démarrage sur un autre média ;
- partitions systèmes en lecture seule ;
- assignation de ports spécifiques pour le périphérique d'entrée et le périphérique de sortie (via le *SASUSB*) afin de n'autoriser le HID que sur les ports où l'accès physique est restreint ;
- authentification *TLS* et/ou *Kerberos* (implémenté dans le *SASUSB*) des serveurs d'analyse et d'upload ;
- suppression/désactivation des modules USB du noyau Linux (*uas*, *usb_storage*, *usbnet*...).

Enfin, nous renvoyons le lecteur aux recommandations de l'ANSSI concernant la sécurité des systèmes GNU/Linux [10].

3 Conclusion

Le *SASUSB* permet de répondre aux nombreuses problématiques posées par la manipulation de périphériques USB malveillants. L'utilisation de deux périphériques distincts, l'exécution du code en espace utilisateur non privilégié et contraint (*seccomp*) ainsi que l'analyse antivirale augmentent fortement la sécurité des transferts de données par le protocole USB. La modularité du *SASUSB* et son interface *protobuf* offrent, outre le développement de SAS d'import de données, de nombreuses possibilités.

Les travaux sur ce projet sont toujours en cours et des améliorations sont envisagées dans un futur plus ou moins lointain :

- signature des clefs USB de confiance afin de n'autoriser que celles-ci sur les postes utilisateurs (développement d'un *driver* Windows et Linux nécessaire pour vérifier cette signature) ;
- remplacement des bibliothèques C restantes (pour le support *exFAT* en lecture/écriture et *NTFS* en écriture) par des *crates* Rust ;
- fuzzing ;
- lecture de conteneurs chiffrés (*LUKS*, *ZED !*...) ;

— exportation de données du SI par le *SASUSB* (afin d'éliminer totalement la connexion de périphériques de stockage sur les postes utilisateurs).

Références

1. CVE-2010-2568 - Windows Shell LNK Vulnerability, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568.
2. Read it twice ! a Mass-Storage-Based TOCTTOU attack. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Bellevue, WA, August 2012. USENIX Association.
3. CVE-2020-17096 - Windows NTFS Remote Code Execution Vulnerability, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-17096.
4. CVE-2020-7456 - FreeBSD USB HID Vulnerability, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7456.
5. CVE-2021-28312 - Windows NTFS Denial of Service Vulnerability, 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28312.
6. CVE-2021-31956 - Windows NTFS Elevation of Privilege Vulnerability, 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31956.
7. Catalin Cimpanu. Intel CPUs Can Be Pwned via USB Port and Debugging Interface. https://www.bleepingcomputer.com/news/hardware/intel-cpus-can-be-pwned-via-usb-port-and-debugging-interface/, 2017.
8. Catalin Cimpanu. Rare BadUSB attack detected in the wild against US hospitality provider, 2020. https://www.zdnet.com/article/rare-badusb-attack-detected-in-the-wild-against-us-hospitality-provider/.
9. The Computer Incident Response Center Luxembourg (CIRCL). CIRCLean USB Sanitazer. https://circl.lu/projects/CIRCLean/, 2013.
10. Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI). Recommandations de sécurité relatives à un système GNU/Linux. https://www.ssi.gouv.fr/guide/recommandations-de-securite-relatives-a-un-systeme-gnulinux/, 2019.
11. Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI). Profil de fonctionnalités et de sécurité – Sas et station blanche (réseaux non classifiés). https://www.ssi.gouv.fr/guide/profil-de-fonctionnalites-et-de-securite-sas-et-station-blanche-reseaux-non-classifies/, 2020.
12. Sergiu Gatlan. Hackers use BadUSB to target defense firms with ransomware, 2022. https://www.bleepingcomputer.com/news/security/fbi-hackers-use-badusb-to-target-defense-firms-with-ransomware/.
13. Maxim Goryachy and Mark Ermolov. Tapping into the core. Chaos Computer Congress, 2016. https://media.ccc.de/v/33c3-8069-tapping_into_the_core.
14. Stephane N. Le Guichet. https://le-guichet.com, 2020.
15. Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. A transparent defense against usb eavesdropping attacks. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, New York, NY, USA, 2016. Association for Computing Machinery.

16. Karsten Nohl and Jakob Lell. BadUSB - On Accessories that Turn Evil. Black Hat, 2014. https://www.youtube.com/watch?v=nuruzFqMgIw.
17. Quarkslab. Irma. https://irma-oss.quarkslab.com/, 2018.
18. Fengwei Zhang. Badusb-c : Revisiting badusb with type-c. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*, ASSS '21, page 7–9, New York, NY, USA, 2021. Association for Computing Machinery.

A Interface WEB

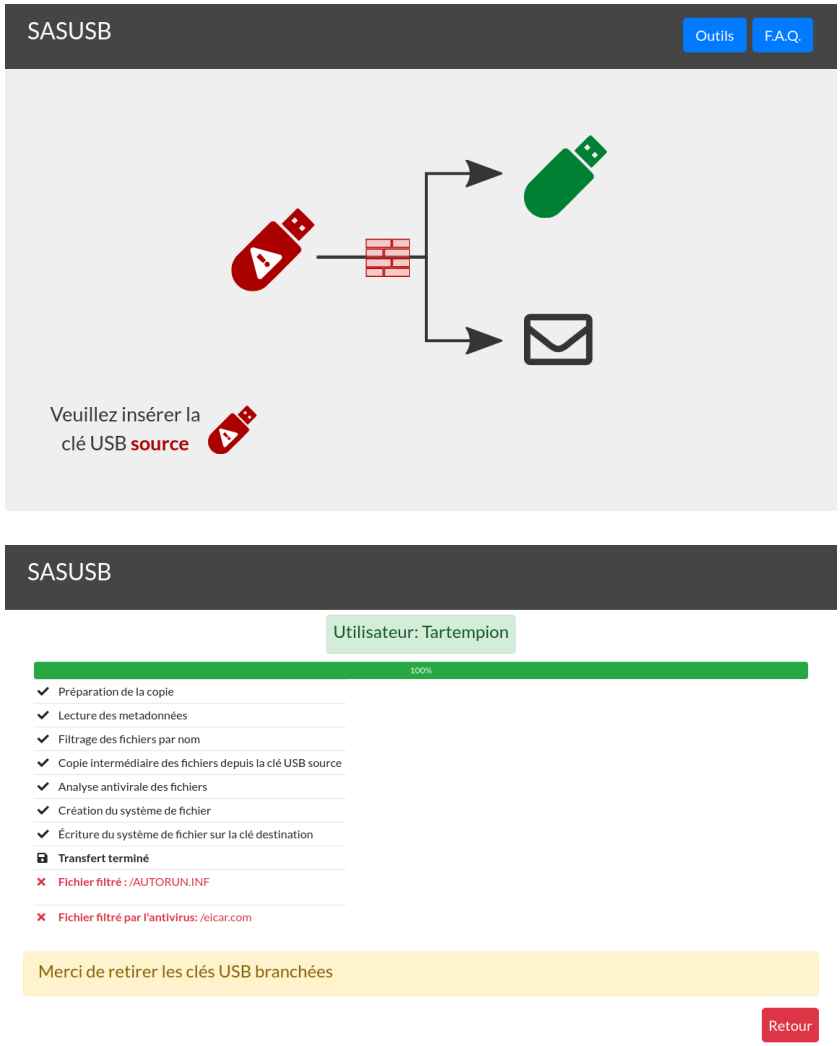

Fig. 3. Interface d'accueil du *SASUSB* / Interface après un transfert