# Google Chrome sous Linux

**Yoann Guillot**

***Chrome est le navigateur internet publié par Google. Il utilise des fonctions de sécurité avancées pour tenter de protéger l'utilisateur contre les pirates de tous poils. Nous allons ici détailler les mesures novatrices mises en place spécifiquement pour l'environnement GNU/Linux.***

Google Chrome est un navigateur Internet. Sa première version stable est publiée fin 2008 pour Windows ; mi-2010 pour les versions GNU/Linux et MacOS.

Son code source est disponible intégralement, via le projet Chromium[chromium]. Google Chrome en est une version précompilée, packagée et distribuée par Google.

Quelques éléments distribués avec Chrome ne sont pas disponible dans Chromium, notamment une version modifiée du plugin Flash afin de pouvoir s'exécuter dans un environnement restreint (nommé « Pepper Flash »).

Par la suite, nous parlerons indifféremment de Chrome ou Chromium, sauf mention particulière.

## Chrome et la sécurité

Chrome est très récent par rapport aux autres navigateurs existants : en particulier Mozilla Firefox et Microsoft Internet Explorer, qui ont chacun un long historique. Cela lui donne un avantage certain : pas besoin de rétro-compatibilité, et la possibilité d'utiliser directement une architecture « moderne », notamment l'exploitation de machines multi-coeurs, la rapidité d'exécution, et la prise en compte de la sécurité dès le départ.

Sur ce dernier point, Chrome est apparu après une période qui a vu la multiplication d'attaques en tout genres visant les sites Internet, et les navigateurs eux-mêmes. De plus, étant développé par une société dont le business dépend quasi-exclusivement d'Internet, la composante sécurité est primordiale. Cela aura deux effets principaux : le premier, évident, est que Chrome sera développé avec la sécurité comme objectif prioritaire ; mais un deuxième effet se manifeste également : le fait d'inciter la concurrence, principalement Firefox et Internet Explorer, à eux aussi se pencher plus sérieusement sur cette problématique.

Nous nous intéresserons ici à la protection de la machine de l'utilisateur par rapport à un site Web malveillant en détaillant les mécanismes du noyau Linux utilisés ; nous laisserons de coté les techniques déployées pour se prémunir contre les XSS, man in the middle TLS et autres joyeusetés peuplant la toile.

L'approche choisie par Google consiste à mettre en place le principe du moindre privilège, c'est-à-dire faire en sorte que le système ne nous laisse comme capacités que celles dont nous aurons besoin et nous interdise l'accès à tout le reste. Cette politique est mise en œuvre au sein de Chrome par l'utilisation d'un bac à sable (sandbox), que nous allons étudier ici. Plusieurs moutures sont disponibles, dont les deux principales sont :

- setuid
- seccomp

Chacune repose sur un ensemble de fonctionnalités distinctes du noyau Linux[sandbox].

Le modèle de sécurité envisagé est le suivant : on suppose qu'un attaquant prend le contrôle d'une partie du navigateur. Il peut s'agir d'une extension malveillante, ou d'une vulnérabilité logicielle exploitée, par exemple dans le code responsable de l'interprétation et du rendu d'une page web. Ce genre de scénario est tout à fait réaliste : le navigateur ayant un très grand nombre de fonctionnalités complexes, il héberge une quantité considérable de code qu'il serait naïf de considérer exempt de bugs, comme le prouve la longue liste de rectificatifs publiée régulièrement par tous les acteurs de ce marché. Ainsi, armé au mieux d'une exécution de code arbitraire dans le contexte du navigateur, l'attaquant a la capacité d'effectuer n'importe quelle opération, n'importe quel appel système. On souhaite, dans ce cadre, empêcher cet attaquant de pouvoir accéder aux fichiers de l'utilisateur, élever ses privilèges, ou encore pouvoir s'installer de manière pérenne

sur la machine.

On entend souvent parler de mécanismes d'isolation sous Linux, mais il s'agit en général de systèmes de contrôle d'accès globaux, comme *Grsec*, *Tomoyo*, *Rsbac* ou encore *SELinux*. Ceux-ci sont très efficaces, mais ne correspondent pas à l'usage que l'on souhaite ici. Tous reposent sur l'intervention de l'administrateur de la machine qui doit définir une politique de sécurité, indiquant par exemple que le navigateur n'a pas le droit d'accéder à */opt/secret/*. Ici nous nous trouvons dans un paradigme différent, où un programme souhaite lui-même limiter ses prérogatives afin de contenir un attaquant exploitant une éventuelle faille de programmation.

# La sandbox *setuid*

Dans la première incarnation de la sandbox, « setuid », Chrome fait appel à un certain nombre de fonctionnalités classiques du noyau Linux. On y retrouve des idées anciennes, comme on pouvait en voir dans Qmail, ou OpenSSH avec le patch PrivSep. Dans le code source de Chromium, cette sandbox se trouve dans */src/sandbox/linux/suid/*.

## État de l'art

Pour OpenSSH, qui est un démon qui s'exécute avec les droits root (il doit pouvoir logguer n'importe quel utilisateur sur le système), le patch PrivSep[privsep] fut publié en 2002. Il permet de drastiquement réduire la quantité de code s'exécutant effectivement avec les privilèges root, en déléguant le maximum de traitements à un nouveau processus, créé pour l'occasion, qui s'exécute avec des droits restreints. Seules les opérations nécessitant effectivement l'uid 0 sont effectuées par le processus privilégié, après validation.

Qmail utilise l'architecture unix de façon encore plus poussée[qmail]. Ce logiciel de distribution de mail, publié en 1998 par D.J. Bernstein, segmente le travail en un jeu de tâches distinctes, et chacune de ces tâches est réalisée par un processus dédié, qui s'exécute avec un UID lui aussi dédié. Les processus communiquent entre eux via des *pipes*. Ainsi le processus chargé de recevoir les e-mails venant d'un serveur distant ne peut interagir avec le processus responsable du stockage de ce mail dans la boîte aux lettres de l'utilisateur local qu'au travers de l'interface spécifiée. En fait, il ne peut même pas directement impacter le processus responsable du stockage des logs qu'il génère. Chacun de ces programmes est concu pour s'exécuter dans un **`chroot()`**, c'est à dire sans accès au système de fichier. Enfin, leurs ressources sont restreintes via **`setrlimit()`** afin de minimiser le risque de déni de service envers la machine. Nous reviendrons plus en details sur certains de ces différents mécanismes par la suite.

## La sandbox de Chrome

Dans la même veine, Chrome utilise les fonctionnalités offertes par le noyau Linux pour restreindre ses privilèges.

Cette sandbox se présente sous la forme d'un binaire, « chrome-sandbox », *setuid* root pour pouvoir utiliser les différentes APIs. Il vient avec une bibliothèque qui permet au programme restreint de communiquer avec lui. Nous sommes dans le contexte d'un programme qui souhaite lui-même limiter ses capacités, on part donc du principe que le programme est initialement de confiance, mais qu'il pourrait ensuite être corrompu et détourné par un attaquant. Ainsi on laisse au programme le loisir de s'initialiser avant de mettre en place les restrictions, et finalement de traiter des données potentiellement corruptrices. Cela évite de devoir inclure dans les permissions de la sandbox toutes les actions qui ne sont nécessaires qu'au démarrage du programme, comme le chargement de bibliothèques dynamiques, la lecture de fichiers de configuration, etc.

La première action réalisée par *chrome-sandbox* consiste à migrer vers un nouvel espace d'identifiants de processus.

## CLONE_NEWPID

Il s'agit d'une fonctionnalité du noyau Linux ajoutée à la version 2.6.24 (en 2008), et initialement prévue pour aider au support des *containers*, une forme de virtualisation. Un processus root peut créer un nouvel espace

de PID en effectuant l'appel système **clone()** avec le flags **CLONE_NEWPID**. L'appel système **clone()** est similaire à **fork()** à ceci près qu'il supporte un argument numérique permettant de spécifier un certain nombre d'options au kernel ; il est notamment utilisé pour la création des *threads*. Tout comme **fork()**, cet appel système duplique le processus courant dans un nouveau processus (le fils), et reprend l'exécution. Du point de vue du programmeur, cette de fonction retourne deux fois, une fois dans le processus père, avec comme valeur de retour l'identifiant du fils fraîchement créé, et une fois dans le fils, avec comme valeur de retour **0**. Avec l'option **CLONE_NEWPID**, le fils se retrouve dans un nouvel espace de PIDs. Les effets de ce namespace font que le processus sera isolé du reste du système, et aura de son point de vue le pid 1. S'il **fork()** à nouveau, ses fils auront les PID 2, 3, etc. L'intérêt du point de vue sécurité est que tous les processus créés dans ce namespace ne pourront utiliser, pour les appels systèmes requérant un identifiant de processus, que les PIDs existant dans ce conteneur. Cela signifie notamment qu'un tel processus ne pourra pas envoyer de signal à un autre en-dehors de cet enceinte, ni tenter de le déboguer via **ptrace()**. Par contre, pour le père ainsi que pour tous les processus du namespace parent, ces processus seront visible ; mais sous un PID différent. Le premier processus dans un nouveau namespace possède un rôle particulier, similaire au *init* classique : tous les processus orphelins du namespace seront reparentés sous celui-ci ; et de plus, si ce processus venait à se terminer, tous les processus du namespace (et des éventuels namespace fils) seraient terminés.

*Note*

Le système de fichier */proc/* n'est pas impacté par le changement de namespace PID : tous les processus, dans tous les namespaces, voient le même contenu dans ce pseudo-répertoire. Ce contenu reflète la vision qu'avait le processus qui a monté le système de fichier : si un processus dans le namespace restreint monte un autre filesystem */proc/* quelquepart, celui-ci ne reflètera que les processus visibles à depuis le namespace, et ce quelque soit le namespace de celui qui y navigue.

*Fin note*

## CLONE_NEWNET

Cette fonctionnalité date de la même époque, et permet de virtualiser les interfaces réseau. On peut l'activer via les appels système **clone()** ou **unshare()**. Sans configuration particulière, il devient impossible pour les processus dans ce namespace de communiquer avec l'extérieur : **bind()** et **connect()** échoueront, faute d'interface réseau adéquate disponible.

La première action de la sandbox est donc la migration dans un nouvel espace de PIDs, dans la fonction *MoveToNewNamespaces()*. L'appel système **clone()** a la particularité d'ignorer les options invalides, ainsi on peut directement tenter un appel avec l'ensemble des options que l'on souhaite, et l'implémentation du noyau ne retiendra que celles qu'elle comprend. Par contre, cela nous prive de la possibilité de savoir quelles options ont effectivement été prises en compte. Dans le cas de **CLONE_NEWPID**, un test via **getpid()** dans le fils nous indiquera si l'on se trouve dans un nouvel espace (pid == 1) ou non. Il faut prendre garde pour ce test à utiliser directement l'appel système, car la fonction de la glibc pourrait reposer un cache, invalidé par l'appel à **clone()**. Le code de la sandbox adopte un mode de programmation défensive, pour le cas où l'API du noyau changerait, et renverrait une erreur pour les options non supportées : une première tentative est faite en combinant **CLONE_NEWNET** et **CLONE_NEWPID**, et en cas d'échec **CLONE_NEWPID** est spécifié seul. Si le kernel ne supporte pas NEWPID (retour d'erreur de l'appel ou ignorance silencieuse du paramètre), l'exécution continue dans le namespace courant. En cas de création d'un nouveau namespace, le premier processus effectue immédiatement un nouveau **fork()**, pour se consacrer à son rôle d'*init* dédié (fonction *SystemInitProcess()*). L'exécution continue en tant que fils de celui-ci. La sandbox crée finalement quelques variables d'environnement indiquant quelles options ont été utilisées pour le **clone()**, à titre informatif.

*Image : chrome-newpid.png*

Les différents changements de processus impliqués dans la création d'un nouvel espace de PIDs

*fin legende*

La fonction de cette partie du code est similaire à l'utilisation d'UIDs spécifiques dans *qmail* ou *openSSH* : interdire à un de ces processus, devenu malveillant, de pouvoir impacter les autres activités de l'utilisateur. Cependant cette approche présente plusieurs avantages : lesdits processus appartiennent toujours à l'utilisateur, ce qui permet à celui-ci de les tuer éventuellement, s'ils se mettaient par exemple à consommer des ressources de manière excessive, chose impossible s'ils changeaient d'UID. De plus, le changement d'utilisateur nécessite une configuration spécifique du système par l'administrateur, qui doit réserver une certaine plage d'UIDs pour cet usage. L'utilisation de **`CLONE_NEWPID`**, au contraire, fonctionne sans configuration particulière de la machine.

## Chroot

**`Chroot()`** est un appel système permettant de redéfinir la racine du système de fichier (« / ») pour un processus et ses fils à un sous-répertoire particulier. Il s'agit d'une fonctionnalité extrêmement intéressante pour la restriction de privilèges : si l'on exécute un programme dangereux dans un chroot défini sur un répertoire vide, ce programme ne pourra a priori accéder à aucun fichier de la machine. Cette méthode n'est toutefois accessible qu'au super-utilisateur root. On pourrait donc penser à la démarche suivante : on exécute un programme assistant, *setuid* root, qui va effectuer le **`chroot()`**, reprendre l'UID courant, et finalement exécuter notre programme cible dans l'environnement vide. Cependant une fois le chroot() effectué, il n'a plus accès au système de fichier, y compris pour y trouver le code du programme cible ou aucune des librairies dont il aura besoin... Une stratégie plus complexe s'avère nécessaire.

La seconde opération réalisée par la sandbox sert à préparer le changement de racine du système de fichier. La fonction *SpawnChrootHelper()* créée un nouveau processus, via l'appel système **`clone()`**, en lui passant cette fois l'argument **`CLONE_FS`**. Ce paramètre a pour effet que les deux processus partagent leur **`umask()`**, mais surtout leur répertoire courant et la racine du système de fichier. Ainsi, quand le fils effectuera un **`chroot()`** ou un **`chdir()`**, cela impactera également le parent. Comme nous l'avons dit précédemment, le sandboxing repose sur une coopération volontaire du processus cible. Dans cette optique, une paire de *sockets* est créée et partagée entre les deux processus. Le fils, qui s'exécute toujours avec les privilèges administrateur, se met en attente en lecture sur son *socket*. Dès qu'il y recevra un message particulier, il effectuera un **`chroot()`** et un **`chdir("/")`** dans un répertoire prédéfini, et terminera son exécution. De son côté, le parent peut immédiatement restreindre ses privilèges, notamment perdre l'uid 0, car il n'a dorénavant plus besoin, pour abandonner l'accès au système de fichier, que d'envoyer un message dans le *socket* dédié.

Il reste enfin à trouver un répertoire adéquat à utiliser comme nouvelle racine. Une première approche consiste à créer un répertoire vide dans */tmp/*, et à l'utiliser ; cependant */tmp/* est notoirement une source de *race conditions*. Une solution astucieuse est choisie par la sandbox Chrome : elle utilise le répertoire */proc/self/fdinfo/*. */proc/* est un pseudo-système de fichier (il ne correspond à rien sur le disque dur) qui contient un ensemble de meta-fichiers représentant entre autres l'état des différents processus du système. En particulier */proc/self/* contient les informations relatives au processus courant. */proc/self/fdinfo/* contient lui un meta-fichier pour chaque descripteur du processus. Or, si l'on utilise ce répertoire pour notre **`chroot()`**, il s'agira du répertoire correspondant au processus effectuant l'appel système, i.e. le processus privilégié. Cela signifie que le répertoire, en plus d'être naturellement en lecture seule, ne sera modifiable que par root. De plus, le processus se termine une fois le changement de racine effectué ; à ce moment le répertoire se retrouvera vide, car le noyau ferme tous ses descripteurs de fichiers à la mort d'un processus. Cerise sur le gâteau, ce répertoire existe toujours, et son chemin peut être stocké « en dur » dans le binaire, sans avoir besoin de recourir à des concaténations de chaînes ou de PIDs.

*Image : chrome-chroot.png*

Isolation du système de fichier dans Chrome

*fin legende*

Pour résumer, suite à cette étape, nous disposons d'un nouveau *socket* dans lequel nous pouvons écrire afin d'effectuer un **`chroot()`** dans un répertoire vide. Le code de la sandbox stocke le numéro du descripteur de

socket ainsi que PID du processus dans des variables d'environnement, afin d'être plus facilement utilisable. Notons que dans le cadre de la défense en profondeur, cette technique de chroot est également appliquée au processus *init* créé avec notre nouvel espace de PIDs en étape 1. Celui-ci prend soin de garder de côté un descripteur vers */proc/*, pour pouvoir énumérer ses processus fils. En l'occurence, un attaquant qui prendrait le contrôle de ce processus *init* pourrait s'échapper du chroot, en passant le descripteur de */proc* avec **`fchdir()`** puis en ré-invoquant **`chroot()`** sur la véritable racine du système de fichier ; toutefois dans ce cas l'utilisation du chroot vise à empêcher la prise de contrôle initiale du processus plus qu'à contenir un attaquant ; celui-ci étant root, et aurait toute lattitude pour ourdir ses plans machiavéliques. Cela nous rappelle malgré tout l'importance de ne garder aucun descripteur de répertoire ouvert pointant en-dehors du **`chroot()`** sous peine de réduire nos efforts à néant.

À ce stade, il ne nous reste plus qu'à abandonner les droits root, et à transférer l'exécution au programme cible. Ceci est fait au moyen des classiques **`setresgid()`** et **`setresuid()`** qui nous permettent de repasser sous l'UID du simple utilisateur nous ayant invoqué.

## MMF_DUMPABLE

Cette variable interne du noyau détermine principalement le comportement d'un processus en cas de crash, à savoir s'il génère un *coredump* ou non. Mais elle a également un effet secondaire : elle défini si un processus peut être débogué au moyen de **`ptrace()`**. Un processus non dumpable ne pourra pas être débogué, sauf si le requérant possède les droits **`CAP_SYS_PTRACE`** (i.e. le débogueur est *root*). Un autre effet de bord pour les processus non dumpable fait que leur entrée dans */proc/* sera accessible en lecture par root uniquement.

La sandbox recommande le positionnement de cette variable quand plusieurs processus partagent l'environnement restreint, afin de réduire leur capacité de nuisance les uns envers les autres. Ils pourront toujours s'envoyer différents signaux. Ce positionnement se fait au moyen de l'appel système **`prctl(PR_SET_DUMPABLE)`**.

Au final, nous disposons ainsi d'un binaire setuid, qui possède les propriétés suivantes :

- N'importe qui peut l'invoquer, avec n'importe quel argument.

- Il va transférer l'exécution au programme passé en argument, lequel n'aura aucun privilège supplémentaire.

- Au contraire, il se trouvera dans un nouvel espace de PIDs, et ne pourra interagir avec les processus existants.

- Il n'aura pas accès au réseau si le noyau supporte et honore CLONE_NEWNET.

- Il aura accès au système de fichier, jusqu'à ce qu'il écrive un message particulier sur descripteur dont le numéro se trouve dans une variable d'environnement. Après quoi il sera confiné à un répertoire vide, et ne pourra interagir qu'avec les fichiers ayant été ouverts avant cette étape.

Cette sandbox est disponible en version autonome[setuid] sous license libre, et réutilisable pour d'autres projets. Elle dispose dans ce cas d'options supplémentaires, permettant par exemple de passer sous un identifiant utilisateur et groupe particulier.

## Intégration dans Chrome

Chrome est organisé pour s'exécuter dans différents processus. Un processus principal (« navigateur ») est responsable de l'affichage de l'interface graphique et de l'interaction avec l'utilisateur, et pour la navigation un processus est créé pour chaque onglet. Ces processus, nommés « *renderer* », sont responsables d'interpréter le code html et le javascript et d'en fournir une représentation graphique, sous forme d'image, au processus principal, qui se chargera de la présenter à l'utilisateur. Sous GNU/Linux, les *renderers* sont tous des clones d'un même processus, nommé « zygote ». Au démarrage du navigateur, le zygote est créé, et à chaque ouverture d'onglet, celui-ci se **`fork()`** et le nouveau processus se charge du rendu de l'onglet. Les

*renderers* effectuent tous les traitements complexes, et s'exécutent dans une sandbox. Ils communiquent avec le navigateur via un ensemble de canaux de communication inter-processus (IPC). Les plugins du navigateur (comme Flash) s'exécutent également dans un processus différent, et peuvent ou non être confinés dans une sandbox.

Le zygote et le processus principal exécutent le même binaire. Au démarrage, le processus principal `fork()`, et re-exécute son image au travers de la sandbox setuid. Cela permet à la fois d'isoler le zygote dans la sandbox, mais également de réinitialiser sa disposition en mémoire (ASLR), ce qui évite qu'il ne connaisse l'adresse d'élements dans l'espace d'adressage du navigateur. Selon ce mode de fonctionnement, tous les *renderers* se trouvent dans la même sandbox : même espace de PIDs, et même racine du système de fichier. Dans ce cadre, le zygote passe en mode non-DUMPABLE, qui sera hérité par chaque renderer et leur interdira de se déboguer les uns les autres. De plus, le zygote active les restrictions sur le système de fichier avant tout contact avec des données non contrôlées, auxquelles seuls les *renderers* seront exposés.

*note*

Ce mode de fonctionnement, basé sur un zygote qui se clone pour chaque nouvelle requête, permet de gérer les mises à jour du système[zygote] : de cette façon, le navigateur est sûr de la version du renderer avec laquelle il communique, même si le programme sur le disque est mis à jour avec une nouvelle version dont le système IPC serait incompatible.

*Fin de note*

Il existe quelques cas où la présence de la sandbox s'avère gênante pour le navigateur.

Tout d'abord, une grande partie de l'interface du navigateur, comme les pages de status, s'exécutent dans l'espace restreint. Or certaines de ces pages présentent à l'utilisateur des informations bas niveau pour indiquer l'état des différents onglets par exemple (comme *about:memory*). Celles-ci ont besoin notamment d'afficher le PID réel des renderers, or ceux-ci ne sont pas accessibles depuis la sandbox. Il est également difficile d'y accéder depuis l'extérieur de la sandbox : il faudrait pour cela posséder un marqueur unique pour un renderer et pouvoir rechercher parmi tous les processus lequel possède ce marqueur. Or comme ils sont non-dumpable, leur entrée dans */proc* n'est lisible que par root. Chrome résoud ce problème en permettant au binaire chrome-sandbox de faire cette recherche pour l'utilisateur. En lui passant l'option *--find-inode*, il va parcourir l'ensemble des processus, et renvoyer le premier qui possède un descripteur de fichiers ouvert ayant un numéro d'inode particulier. À la création de chaque renderer, le zygote créé un socket spécifiquement pour servir de marqueur pour cette technique, et stocke le mapping entre les PIDs du namespace et les PIDs externes (PID réel). Notons que cela ne constitue pas une faille dans la sandbox, puisque la sécurité repose sur le refus par le kernel de laisser interagir les processus dans la sandbox avec les processus en dehors, et non sur le caractère secret des PIDs.

Ensuite, le code de Chrome est principalement développé par Google, mais il repose également sur certains composants tiers. Par exemple, la cryptographie est gérée par la bibliothèque NSS. Il serait sans doute possible, mais possiblement fastidieux, de modifier leur code afin de ne pas requérir de ressources inaccessibles depuis la sandbox ; ou de réserver celles-ci avant l'entrée en mode confiné. Une autre approche a été retenue : laisser ces bibliothèques tenter d'accéder aux différentes ressources qu'elles souhaitent, accès qui seront refusés, mais qui devraient être gérés gracieusement. Une ressource particulière est toutefois nécessaire pour NSS : l'accès à */dev/urandom*, comme source d'entropie. Plutôt que de modifier le code, les développeurs ont optés pour une approche disons plus générique. Ils mettent de côté un descripteur de fichier vers */dev/urandom* lors de l'initialisation du zygote, et modifient le binaire de Chrome afin qu'il exporte une fonction `fopen()`. Le chargement de programmes sous GNU/Linux est fait de telle sorte que les librairies utilisant *fopen* vont choisir d'utiliser la version exportée par Chrome plutôt que la version disponible dans la GLibc. La version de Chrome est très simple : soit on lui demande d'ouvrir ce chemin particulier, auquel cas elle réutilise le descripteur mis de côté, soit elle transfère le contrôle à la véritable fonction de la GLibc, qui va alors renvoyer une erreur.

Le navigateur peut souhaiter passer un certain nombre de variables d'environnements au zygote lors de sa création. Le fait d'utiliser un binaire setuid conduit la GLibc à filtrer une partie de ces variables (par exemple *LD_PRELOAD*) ; aussi un mécanisme de contournement est-il mis en place. Avant la création du processus, le navigateur va remplacer toutes les variables sensibles par une version préfixée de la chaîne « SANDBOX_ », et la sandbox, avant d'exécuter le zygote, va restaurer ces variables à leur nom initial afin qu'elles prennent tous leurs effets.

Enfin, afin de maximiser le confort des utilisateurs, Chrome met en place, comme beaucoup de logiciels conséquents, un gestionnaire de fautes, afin de pouvoir remonter aux développeurs les erreurs critiques rencontrées par le programme. Ce composant se nomme « Breakpad » pour ce navigateur. Son fonctionnement est simple : en cas d'erreur fatale (erreur de segmentation par exemple, causée par un déférencement de pointeur invalide), un processus spécial est chargé d'inspecter l'état du renderer afin de remonter le maximum d'informations à Google : la pile d'appels, l'état des registres du processeur, etc. Or ceci nécessite d'une part de connaître l'identifiant du renderer, problème résolu comme nous l'avons vu précedemment, mais également de pouvoir interroger le renderer, ce qui se fait au moyen de l'appel **`ptrace()`**. Celui-ci étant interdit par la propriété *non-dumpable*, l'appel échouera. La parade consiste à installer un gestionnaire de signaux dans chaque renderer, qui va simplement rendre le processus dumpable en cas d'erreur critique. Par mesure de sécurité, ce gestionnaire de signal vérifie auparavant, en inspectant la structure *siginfo*, que le signal résulte bien d'une faute réelle, et non pas simplement d'un renderer corrompu qui aurait envoyé un signal *SIGSEGV* manuellement, via **`kill()`**, afin de pouvoir nous déboguer lui-même.

Ainsi décrite, la sandbox setuid est efficace. Il reste à prendre garde à bien sécuriser tous les canaux de communication que l'on fournit aux processus restreints, en l'occurence toutes les fonctions du navigateur accessible via les IPCs ; mais cela dépasse du périmètre de cet article.

Pour l'anecdote, sachez que cette sandbox a été conçue par un compatriote et ami, Julien Tinnès, que l'on salue au passage.

# SeccompSandbox

La sandbox setuid est intéressante, cependant elle ne protège pas contre un vecteur d'attaque très important : les failles du noyau. Si un attaquant dispose d'un *0-day*, ou que le noyau n'est pas à jour, il pourra vraisemblablement prendre le contrôle du processus (et de la machine). Pour pallier ce problème, le seul moyen offert par le kernel pour restreindre l'utilisation d'appels systèmes, et donc réduire cette surface d'attaque, est *`seccomp`*. Il s'agit d'un paramètre à passer à l'appel système **`prctl(PR_SET_SECCOMP)`**, dont la fonction est très simple : après qu'il a retourné, le thread courant n'aura plus le droit que d'utiliser les quatre appels systèmes : **`read()`**, **`write()`**, **`sigrestart()`**, et **`exit()`**. Tout autre appel système terminera immédiatement la tâche. Difficile de faire plus sécurisé !

Un autre inconvénient est que la sandbox setuid nécessite un binaire setuid, et requiert donc l'intervention de l'administrateur de la machine pour pouvoir être effective. Par opposition, *seccomp* ne nécessite aucun privilège supplémentaire, à aucun moment : un utilisateur pourra donc installer son navigateur dans son coin, et tout de même bénéficier d'une sécurité renforcée.

Le code source de cette sandbox[seccompsrc] se trouve dans */src/seccompsandbox/*.

On imagine bien qu'il sera extrêmement difficile au renderer de faire son travail en utilisant uniquement *read* et *write*. En particulier, le renderer aura besoin d'allouer de la mémoire pour travailler. Or il est impossible de déléguer cette tâche à un autre processus : la seule façon d'allouer de la mémoire dans l'espace d'adressage d'un autre processus, sous linux, consiste à détourner le flot d'exécution de la cible, via **`ptrace()`**, pour lui faire exécuter un appel système type **`brk()`** ou **`mmap()`** ; mais cet appel échouera du fait de seccomp. Il nous reste une échappatoire : on peut envisager qu'un thread, non soumis à seccomp, partage l'espace d'adressage du renderer. Il aura alors la possibilité d'allouer de la mémoire, en fait il pourra effectuer n'importe quel appel système. En le couplant avec un *socketpair* partagé avec le thread restreint, ceux-ci pourront communiquer, et le renderer pourra demander au thread privilégié d'effectuer toutes les actions auxquelles il n'a pas droit, en son nom. C'est l'approche choisie par Google pour la sandbox seccomp[seccomp].

## La libc

Cette organisation pose de gros problèmes en pratique. En effet, le renderer ne peut désormais même plus

tenter de faire un appel système interdit, car il serait terminé immédiatement. Or Chrome est composé d'une quantité considérable de code ; il parait difficile d'inspecter et de modifier tous les sites où le programme pourrait effectuer un appel système pour les remplacer par un appel via le thread privilégié. La philosophie de Google est au contraire plutôt axée sur l'indépendance entre la partie développement et la partie sécurisation. De toute façon, même si le code de Chrome était modifié pour cela, énormément de code tiers resterait susceptible d'effectuer des appels systèmes, en particulier la GLibc. Voici la solution retenue par Google (attachez vos ceintures, nous entrons maintenant dans la quatrième dimension). Au moment de la mise en place du mode seccomp, le zygote parcourt la liste de ses mappings mémoire. Lorsqu'il trouve un mapping correspondant à une librairie connue pour contenir des appels systèmes (*libc*, *libdl*, etc.), alors un parseur ELF maison est utilisé pour retrouver en mémoire l'adresse de la section « .text », qui contient le code exécutable. Cette section est parcourue, octet par octet, et découpée en morceaux, qui sont identifiés en utilisant comme marqueur les séquences de *nops* dont le compilateur se sert pour maintenir le code aligné, une mesure d'optimisation. Ces morceaux sont ensuite *désassemblés,* à la recherche à la fois d'une instruction correspondant à un appel système (*int 80h*, *call gs:[10h]*, etc.) et de quelques instructions simples entourant celle-ci.  Une instruction est dite « simple » ici si elle peut être déplacée sans effets secondaires ; par exemple une addition, ou autre opération arithmétique, est acceptée. Par contre une instruction comme « *call* » utilise sa propre adresse et la pousse au sommet de la pile, elle ne peut donc pas être déplacée sans lui adjoindre un traitement spécifique visant à corriger l'addresse poussée. Le but est de trouver une plage de 6 octets (6 en x86, 5 suffisent en x64) contenant l'appel système, afin de la remplacer par un saut vers une séquence de code spécifique. Cette séquence reprend les instructions identifiées précédemment, que l'on vient d'écraser, en remplaçant l'appel système par une requête au thread sécurisé via le *socket* dédié. Autrement dit, Chrome contient un désassembleur rudimentaire, et modifie en mémoire et à la volée le code de la *glibc* pour remplacer les instructions responsables des appels système par un appel au thread sécurisé. Notons que le code en charge de cette phase prend garde à ne pas dépendre lui-même de la libc, et effectue ses appels système directement, afin d'éviter de modifier une instruction qu'il serait en train d'exécuter.

Cette démarche a pour unique objectif d'aider le renderer à fonctionner. Si jamais le code décrit ci-dessus oublie un appel système quelquepart dans un chemin, l'unique effet de son exécution sera la terminaison du processus. Notre algorithme sert simplement à permettre au programme de continuer de fonctionner, autrement dit à améliorer l'expérience utilisateur. Nous pouvons nous permettre d'avoir un code approximatif, tant que celui-ci fonctionne sufisamment bien en pratique : quoi qu'il arrive la sécurité ne repose pas notre capacité à identifier tous les appels système.

## Le thread privilégié

Si l'on reprend notre cible de sécurité, l'objectif est de se protéger d'un renderer malveillant. Ici, notre thread non restreint partage son espace d'adressage avec le renderer. Or celui-ci peut modifier, à tout instant, n'importe quelle page mémoire qui lui est accessible en écriture, et celles-ci sont communes aux deux threads. La conséquence directe est que nous ne devons plus, à partir du moment où le mode sécurisé est activé, reposer sur aucune zone mémoire inscriptible. En particulier, nous ne pourrons plus faire confiance à notre pile d'exécution. En effet, le thread du renderer pourrait malicieusement y réécrire l'adresse de retour de la fonction courante, de sorte à faire exécuter au thread non restreint du code réalisant ce qui lui chante. Deuxième effet kisscool : puisque c'est le seul moyen de contrôler complètement l'usage fait de la mémoire, tout le code exécuté à partir de ce moment est entièrement écrit en assembleur, et n'utilise aucune mémoire inscriptible. Le code est stocké en lecture seule, il est donc à l'abri des modifications. Notre thread a besoin de maintenir un certain nombre de variables tout au long de son exécution. Le seul espace de stockage dont nous disposons est le jeu de registres du processeur. Ces registres sont spécifiques à notre thread, le renderer n'y aura jamais accès. L'ensemble de registres généraux étant très restreint, en particulier en x86, le thread privilégié utilise les registres MMX comme espace tampon. C'est là par exemple qu'il stockera les différents descripteurs de fichiers qu'il utilise. Cette approche présente cependant des limites ; il est notamment impossible d'effectuer un appel système nécessitant un buffer mémoire. En effet, si l'on souhaite par exemple exécuter l'appel système **`open()`** avec comme paramètre *"/foo/bar"*, il est impossible de garantir que le thread du renderer ne va pas modifier le buffer contenant cette chaîne, pour la remplacer par *"/evil/bad"* entre le moment où l'on aura inspecté son contenu et le moment où l'on l'utilise comme argument à l'appel système (ou plus précisément, le moment où le noyau va récupérer son contenu). Il s'agit d'un problème classique de sécurité, où l'on ne peut garantir l'atomicité du test d'une valeur et son utilisation : *TOCTTOU*, pour « *time of check to time of use* ». Accessoirement, l'écriture de code directement en assembleur est complexe, et peu portable, en particulier si l'on ne dispose que de peu d'espace de stockage.

Afin de minimiser la quantité de code écrit sous ces contraintes, tous les tests complexes relatifs à la politique d'appels système sont délégués à un processus distinct, dédié, qui sera écrit en C++. Notre thread communique avec celui-ci via une paire de socket et une page mémoire partagée. Cette page est mappée en lecture seule dans l'espace du renderer, pour garantir la non-compromission des données par ce dernier. Cela nous permet de résoudre le problème du **`open()`** soulevé précedemment : notre thread envoie via le socket le chemin du fichier, qui peut être "*/foo/bar*" ou "*/evil/bad*" selon si le renderer a ou non tenté d'intervenir. Le processus stocker ce chemin dans sa mémoire propre, hors d'atteinte, et peut vérifier si l'accès à ce fichier doit être accordé au renderer. Si l'accès est refusé, il transmet cette information au thread sécurisé, qui renvoie un code d'erreur approprié au renderer ; s'il est validé, le processus écrit dans la zone mémoire partagée l'ensemble des arguments à utiliser, y compris le buffer contenant le chemin et le pointeur sur celui-ci, et en informe le thread. Ce dernier initialise ses registres directement à partir des valeurs trouvées dans la page mémoire, où seul l'autre processus peut écrire : l'appel système est alors garanti d'utiliser les paramètres tels qu'ils ont été testés. Le processus est notamment chargé de valider tous les appels systèmes relatifs au management de la mémoire virtuelle, comme **`mmap()`** ou **`mprotect()`**. Pour ceux-ci, il vérifie que l'appel ne modifiera pas les permissions ou le contenu de la page utilisée pour les communications avec le thread, ni d'aucune page contenant le code exécutable initial du zygote, utilisé par le thread sécurisé.

*Image : chrome-seccomp.png*

Mise en place de la sandbox seccomp, et utilisation d'**`open()`** par le renderer

*fin legende*

Cette sandbox, écrite par le googler Markus Gutschke, est fonctionnelle, mais elle présente dans certaines situations des dégradations de performances jugées trop importantes. Celles-ci sont liées à toute la gymnasique effectuée pour chaque appel système, y compris les appels bénins comme **`getpid()`**. Certains de ces appels sont optimisés dans l'implémentation en assembleur, cependant de manière insuffisante. C'est pourquoi elle est désactivée par défaut : utilisez l'argument *--enable-seccomp-sandbox* pour forcer son utilisation. Lorsqu'elle est activée, cette sandbox est exécutée à l'intérieur de la sandbox *setuid*, puisqu'elles ne sont pas incompatibles, et que l'on n'est jamais trop prudent.

# Seccomp Filter

Le modèle de *seccomp* est intéressant, mais, comme on vient de le voir, complexe à mettre en œuvre. Aussi les équipes de Google ont participé à formaliser et à pousser une nouvelle version de ce code qui devrait bientôt être incluse dans le noyau linux « vanilla » version 3.5. Ce nouveau mode, nommé *seccomp filter*, est beaucoup plus souple que le mode 1. Un langage spécifique permet de décrire un automate, qui s'applique à chaque appel système du thread, un peu à la manière des règles iptables. Ces règles définissent, en fonction du numéro d'appel système et éventuellement de ses paramètres, l'action à appliquer. On peut opter, au choix, pour :

- autoriser l'appel système,

- terminer la tâche,

- interdire l'appel système, en spécifiant le code d'erreur (*errno*),

- envoyer un signal **`SIGSYS`** au thread courant sans effectuer l'appel système.

Cette architecture est étudiée pour fournir, à travers une API officielle, le même type de fonctionnalités que la sandbox décrite plus haut. On peut tout d'abord autoriser les appels systèmes que l'on souhaite quasiment sans overhead, contrairement à l'implémentation précédente. Mais surtout, la fonctionnalité *SIGSYS* permet d'arriver au même résultat que si l'on avait manuellement patché chaque instruction réalisant un appel système dans le programme et toutes ses librairies, d'une manière infiniment plus propre et efficace. Le gestionnaire défini pour ce signal peut inspecter les informations fournies par le noyau, notamment le numéro du syscall et l'état des registres du processeur au moment de l'appel, afin de réaliser toute action nécessaire pour émuler l'appel système : ici on pourrait choisir de transmettre la requête à un autre thread ou processus pour vérifier la cohérence avec la politique configurée.

## NO_NEW_PRIVS

Ce mode est plus complexe que le mode seccomp strict, et cela introduit certaines subtilités. Notamment, il est maintenant possible d'autoriser le thread restreint à utiliser les appels systèmes **`fork()`** et **`execve()`**. Dans ce cas, le nouveau thread, processus, ou programme, héritera des règles mises en place. Cela peut introduire des failles de sécurité, car on pourrait par exemple modifier le comportement d'appels systèmes classiques de manière à surprendre un binaire setuid afin d'élever nos privilèges. Aussi, lors de l'introduction de ce nouveau mode, une option est apparue en parallèle : **`prctl(PR_NO_NEW_PRIVS)`**. Cette dernière indique au noyau que le processus courant ne doit pas pouvoir gagner de privilèges lors de l'exécution d'un nouveau programme, qu'il s'agisse d'un fichier setuid ou d'un fichier usant des « filesystem capabilities ». Celles-ci permettent de donner à un programme des *capabilities* spécifiques, par exemple **`CAP_NET_RAW`**, quelquesoit l'utilisateur qui l'exécute ; elles fonctionnent de manière analogue aux fichiers setuid. Lorsque *NO_NEW_PRIVS* est activé, le kernel ignorera tous ces changements de privilèges, et gardera les UID et capabilities du processus courant.

La mise en place de règles *seccomp-filter* n'est autorisée que lorsque cette protection est active.

La partie expérimentale de Chrome utilisant ce mode seccomp se trouve dans le répertoire */src/content/common/sandbox_init_linux.cc*. On y trouve le code responsable de la mise en place d'une politique de filtrage rudimentaire, destinée à s'appliquer, pour le moment, aux plugins utilisant la PPAPI (notamment *Pepper Flash* et le lecteur de PDF interne), et au processus interagissant avec le GPU dans le cadre de *WebGL*. Étant encore expérimental, et faute de tests poussés, il est actif uniquement pour l'architecture *amd64*. Ce code ne fait pas encore usage de la fonctionnalité SIGSYS, et se contente d'un filtrage plus grossier simplement sur les numéros d'appels système et leurs arguments directs. En l'occurence, l'implémentation actuelle ne requiert pas de thread ou de processus supplémentaire. De plus, elle n'aura d'effet que sous les distributions utilisant un noyau supportant ce patch expérimental, à savoir la dernière Ubuntu (12.04) et ChromeOS, en attendant le déploiement généralisé du noyau Linux v3.5.

*Note*

Pepper Flash est une version du plugin Flash d'Adobe, dont le code source est modifié par Google pour mieux s'intégrer à la sandbox. Cette version n'est disponible qu'au format binaire, et n'est distribuée qu'avec Chrome ; une version pour les utilisateurs de Chromium sera peut-être publiée un jour.

*Fin note*

# SELinux

Enfin Chrome peut aussi utiliser une fonctionnalité peu connue de *SELinux*, le changement dynamique de rôle. Celle-ci permet de modifier l'ensemble de règles appliqué à un programme, à sa demande (si cela est autorisé par la politique système). Cette police est fournie par Google pour faciliter le déploiement, par exemple sous Fedora. On peut la trouver dans le dépot source sous */src/sandbox/linux/selinux/*. Le code responsable de la transition de rôle se trouve dans la fonction *SELinuxTransitionToTypeOrDie()* du fichier */src/content/zygote/zygote_main_linux.cc*.

Si ce mode de confinement est sélectionné, il remplace la sandbox setuid, mais l'utilisation de *seccomp* n'est pas impactée.

# Conclusion

Par rapport à son concurrent, Chrome se place largement devant Firefox en ce qui concerne les aspects sécurité système sous Linux, puisque le navigateur de Mozilla n'utilise aucune technique de cloisonnement, alors que celui de Google use de techniques très sophistiquées.

Il est intéressant de noter que les développeurs de Chrome ne se sont pas arrêtés sur une technologie de confinement particulière, mais ont au contraire exploré toutes les pistes disponibles. Leur motivation est sans doute de faciliter l'intégration du navigateur dans les différentes distributions GNU/Linux, qui ont souvent des idées bien arrêtées sur le type de mécanismes qu'elles peuvent accepter ; par exemple RedHat ne jure que

par *SELinux*, ce qui n'est pas le cas de tout le monde. Cette démarche nous permet de faire un tour d'horizon des différentes technologies existantes, mais aussi des améliorations en cours. Sur ce point, Google à fait un effort particulier envers Linux, par rapport notamment à son projet Android, pour pousser de nouvelles fonctionnalités dans le noyau « mainline ». Saluons ce geste, qui contribuera, je l'espère, à améliorer encore le niveau de sécurité offert par ce système d'exploitation libre et accessible à tous.

**yoann.guillot@ofjj.net**
**Chercheur en securite. Auteur de metasm.**

*References:*

[chromium] https://git.chromium.org/chromium/src.git

[sandbox] https://www.cr0.org/paper/jt-ce-sid_linux.pdf

[privsep] https://www.citi.umich.edu/u/provos/ssh/privsep.html

[qmail] http://cr.yp.to/qmail.html

[setuid] https://code.google.com/p/setuid-sandbox/

[seccomp] http://www.imperialviolet.org/2009/08/26/seccomp.html

[seccompsrc] https://code.google.com/p/seccompsandbox/

[zygote] https://code.google.com/p/chromium/wiki/LinuxZygote