

DE LA RECHERCHE À L'INDUSTRIE



Sécurité des systèmes Linux

Mécanismes de prévention d'attaques
Cloisonnement de l'environnement d'exécution

Commissariat à l'Énergie Atomique et
aux Énergies Alternatives | Mathieu Blanc

4 décembre 2019

- 1 Protection contre les attaques**
 - De quoi veux-t-on se protéger ?
 - Mécanismes de contrôle sur les instructions
 - Mécanismes de restriction sur les appels système
- 2 Contrôle d'accès renforcé : MAC**
 - Modèles de politique de sécurité
 - La version Redhat : SELinux
 - La version Ubuntu : AppArmor

3 Conteneurs vs. Virtualisation

- Contexte et définitions
- Virtualisation
 - Émulation complète
 - Émulation niveau logiciel (type 2)
 - Émulation niveau hôte (type 1)
 - Disques durs virtuels
 - Exemple de vulnérabilité
- Conteneurs
 - Premier pas : chroot()
 - Mécanismes avancés du noyau Linux
 - Sandbox
 - LXC, LibContainer et Docker
 - Exemple de vulnérabilité

4 Applications

- Malware
- Provisioning
- Conteneurs
- Aide au développement
- Cloud
- Compartimentation

1 Protection contre les attaques

- De quoi veux-t-on se protéger ?
- Mécanismes de contrôle sur les instructions
- Mécanismes de restriction sur les appels système

2 Contrôle d'accès renforcé : MAC

- Modèles de politique de sécurité
- La version Redhat : SELinux
- La version Ubuntu : AppArmor

- Détournement du flux d'exécution du CPU sous Linux
 - Écrasement de la sauvegarde d'adresse de retour (**stack buffer overflow**)
 - Exploitation du chaînage des blocs du tas (**heap buffer overflow**)
 - Abus de l'interprétation de *format string*
 - Exploitation du dépassement de capacité d'une variable entière (**Integer overflow**)
 - Écrasement de pointeur de fonction
- Injection d'un *payload*
 - *Shellcode*, éventuellement en plusieurs étapes
 - Appels à des fonctions de bibliothèques dynamiques (*return to libc*)
 - *Return-Oriented Programming*
- Fonctions du *shellcode*
 - Ouverture de port TCP en écoute, connexion vers l'attaquant (*connect back, reverse shell*)
 - Modification du système (*rootkit*)
 - *Shellcode* évolué (CANVAS, Meterpreter)

- Première stratégie : prévenir les attaques
 - Comment ?
 - Produire des applications sans vulnérabilité ?
 - Difficile à garantir, problème de recherche toujours actif
 - Bloquer les attaques avant qu'elles n'arrivent sur la machine ?
 - Mais on ne sait pas toujours les repérer
 - **Renforcer les applications à la compilation**
 - Et si on ne dispose pas du code source ?
 - **Implémenter des protections dans l'environnement d'exécution**
 - Et pour des schémas d'attaque encore inconnus ?
 - Seconde stratégie : **limiter l'impact d'une attaque réussie**
 - En exécutant une application à risque de façon isolée dans le système d'exploitation
 - En limitant les droits d'accès de l'application sur le système au strict nécessaire
 - En exécutant un système d'exploitation isolé pour cette application

1 Protection contre les attaques

- De quoi veux-t-on se protéger ?
- **Mécanismes de contrôle sur les instructions**
- Mécanismes de restriction sur les appels système

2 Contrôle d'accès renforcé : MAC

- Modèles de politique de sécurité
- La version Redhat : SELinux
- La version Ubuntu : AppArmor

Protection de la pile

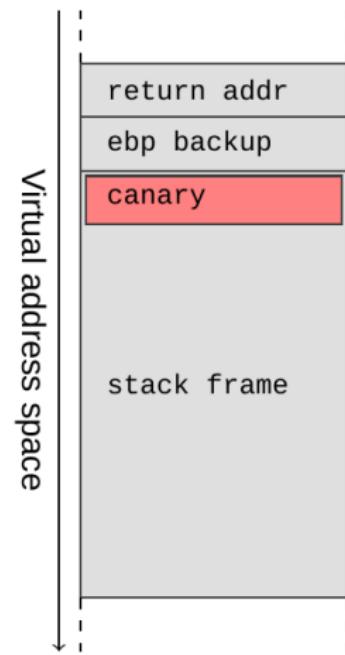
- Idée : détecter des débordements de tampon mémoire dans la pile
 - et arrêter le programme lorsqu'un tel événement est détecté
- Principe de fonctionnement :
 - Le compilateur repère les fonctions allouant des buffers de plus de quelques octets ;
 - il insère dynamiquement des instructions supplémentaires :
 - au début de la fonction, ajout d'un *canary* entre les variables locales et l'adresse de retour ;
 - avant le retour de la fonction, lecture du *canary* et vérification de la non-modification.
 - Le programme quitte par l'appel d'une fonction spécifique en cas de modification.

```
1 *** stack smashing detected ***: /sbin/vuln terminated
2 ===== Backtrace: =====
3 /lib32/libc.so.6(__fortify_fail+0x50)[0xf7677aa0]
4 /lib32/libc.so.6(+0xe4a4a)[0xf7677a4a]
5 /sbin/vuln[0x8048628]
6 [0x41414141]
7 ===== Memory map: =====
```

Protection de la pile

■ Protection à la compilation

```
080485f4 <main>:  
080485f4: push %ebp  
080485f5: mov %esp,%ebp  
080485f7: and $0xffffffff0,%esp  
080485fa: add $0xffffffff80,%esp  
080485fd: mov %gs:0x14,%eax  
08048603: mov %eax,0x7c(%esp)  
08048607: xor %eax,%eax
```



- Problème des attaques par *heap buffer overflow* :
 - Manque de contrôle lors de la désallocation d'un bloc alloué ;
 - Un *overflow* donne à l'attaquant le contrôle sur les pointeurs de blocs précédents et suivants ;
 - Possibilité pour un attaquant d'écrire une valeur choisie à une adresse choisie ;
- Principe de protection : renforcer le contrôle des pointeurs lors de la désallocation.
- Première implémentation dans le projet StackShield, puis intégration à gcc et la glibc.
- Pour activer les différentes protection, quelques options de gcc :
 - `-fstack-protector[-all]`
 - `-D_FORTIFY_SOURCE=2`

Historiquement, l'architecture x86 ne peut empêcher l'exécution de code dans une page mémoire particulière

- Problème : certaines zones mémoire sont marquées non-exécutables pour des raisons légitimes (pile, zones de données) ;
- Dommage collatéral : pendant longtemps les développeurs ont programmé avec le postulat que toute la mémoire est exécutable.

Pour protéger contre les attaques par débordement de buffer, la première protection est d'empêcher l'exécution dans les zones de données :

- Première implémentation logicielle avec le projet PaX (via une définition revue des permissions sur les segments de mémoire) ;
 - Aujourd'hui intégrée au noyau Linux standard ;
- A partir de 2004, intégration du support de pages mémoire non-exécutables dans les processeurs AMD (NX) et Intel (XD).

Aujourd'hui le réglage par défaut est l'utilisation de mémoire non-exécutable ;

- Cependant de nombreux programmes ont nécessité quelques changements dans leur code pour fonctionner ainsi (notamment le serveur X) ;
- En particulier, pour démarrer un programme avec une pile exécutable, il faut modifier une valeur de contrôle dans l'en-tête ELF (outil : execstack).

```
1 % objdump -p vuln1
2
3 vuln1:      file format elf32-i386
4
5 Program Header:
6 ...
7     STACK off      0x00000000 vaddr 0x00000000 paddr 0x00000000 align
8           2**2
9     filesz 0x00000000 memsz 0x00000000 flags rwx
10    ...
```

Suite à la mise en place de la fonction NX, une nouvelle technique d'exécution de code a été mise au point : *return-to-libc*

- Plutôt que d'injecter un shellcode dans la pile ou le tas, on va provoquer un appel à une fonction de la libc.
- Par exemple, pour exécuter un shell :
 - empilement des paramètres nécessaire pour faire un execve("/bin/sh", 0, 0);
 - écrasement de l'adresse de retour avec l'adresse de la fonction execve dans la bibliothèque standard.

Pour contrer ce nouveau type de *payload*, une technique appelée ASLR a été inventée.

- Principe : placer la pile et charger les bibliothèques dynamiques à des adresses aléatoires pour qu'un attaquant ne puisse pas prédire leurs adresses.
 - Première implémentation dans le projet PaX ;
 - Toute la sécurité repose sur l'entropie de variation des adresses, bien plus grande sur architecture 64 bits.
 - Les premières structures "randomisées" sont donc la pile, et les structures allouées par `mmap()` soit le tas et les bibliothèques dynamiques.

Dans la veine de l'attaque *return-to-libc*, d'autres techniques d'exécution de code ont été conçues, notamment regroupées sous le terme ROP (*Return-Oriented Programming*, ~2009) :

- L'attaquant va remplacer les instructions constituant un shellcode classique par l'exécution d'une série de *gadgets* ;
- Chaque *gadget* est une petite série d'instructions suivies d'un *ret*, contenues dans le programme vulnérable ;
- Ainsi l'attaquant n'a plus besoin d'injecter de code exécutible, il peut directement exécuter celui qui est contenu dans des parties du programme sans ASLR.

Pour contrer ces nouvelles techniques, l'ASLR a été étendu :

- au code du programme, qui doit être compilé en mode PIE (*Position-Independant Executable*) ;
- à la zone brk (suit immédiatement la zone de données du programme) ;
- et enfin à la zone VDSO (*Virtual Dynamic Shared Object*), qui contient le code binaire pour effectuer un appel système.

1 Protection contre les attaques

- De quoi veux-t-on se protéger ?
- Mécanismes de contrôle sur les instructions
- Mécanismes de restriction sur les appels système

2 Contrôle d'accès renforcé : MAC

- Modèles de politique de sécurité
- La version Redhat : SELinux
- La version Ubuntu : AppArmor

Restriction des appels système

S'il est capable d'exploiter une faille d'un service, un pirate va chercher à exécuter d'autres commandes sur la machine.

→ Pour réduire l'impact d'une exploitation réussie, on peut limiter l'environnement de ce service.

Le principe de la cage

- dissimuler une partie du système au programme
- ne lui laisser voir que ce dont il a besoin pour s'exécuter correctement
 - fichiers de configuration
 - bibliothèques
 - autres exécutables, scripts...
 - fichiers spéciaux (*pipes, sockets, devices...*)
 - journaux d'événements
 - appels systèmes

À retenir

On cherche à isoler au maximum le programme du reste du système

1 Protection contre les attaques

- De quoi veux-t-on se protéger ?
- Mécanismes de contrôle sur les instructions
- Mécanismes de restriction sur les appels système

2 Contrôle d'accès renforcé : MAC

- **Modèles de politique de sécurité**
- La version Redhat : SELinux
- La version Ubuntu : AppArmor

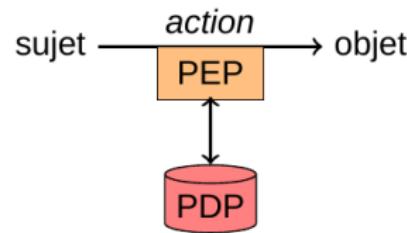
Définition formelle du contrôle d'accès

Le contrôle d'accès est le mécanisme en charge de décider si un *sujet* peut effectuer une *action* sur un *objet*.

- Sujet : entité active (processus, utilisateur) ;
- Objet : entité passive, ressource (fichier, socket) ou autre sujet ;
- Action : les appels système.

Le contrôle d'accès est effectué par deux entités :

- PEP (Policy Enforcement Point) : intercepte l'action et applique la décision d'accès ;
- PDP (Policy Decision Point) : rend une décision d'accès en fonction du triplet (sujet, objet, action).



Discretionary Access Control

- Les permissions sont déterminées par le propriétaire de la ressource ;
- L'utilisateur *root* outrepasse le contrôle DAC (sauf si retrait de la *capability* *DAC_OVERRIDE*) ;
- Par conséquent, il est difficile de restreindre les droits d'un processus exécuté avec le compte *root*.

Mandatory Access Control

- Les permissions d'accès sont définies dans une configuration centrale non modifiable par les utilisateurs (politique de contrôle d'accès) ;
- La politique est conservée dans un composant appelé le Moniteur de référence ;
- Tous les processus du système sont soumis à ce contrôle, même ceux appartenant à *root* ;
- Il devient possible de limiter précisément l'accès d'un processus aux seules ressources dont il a besoin pour s'exécuter.

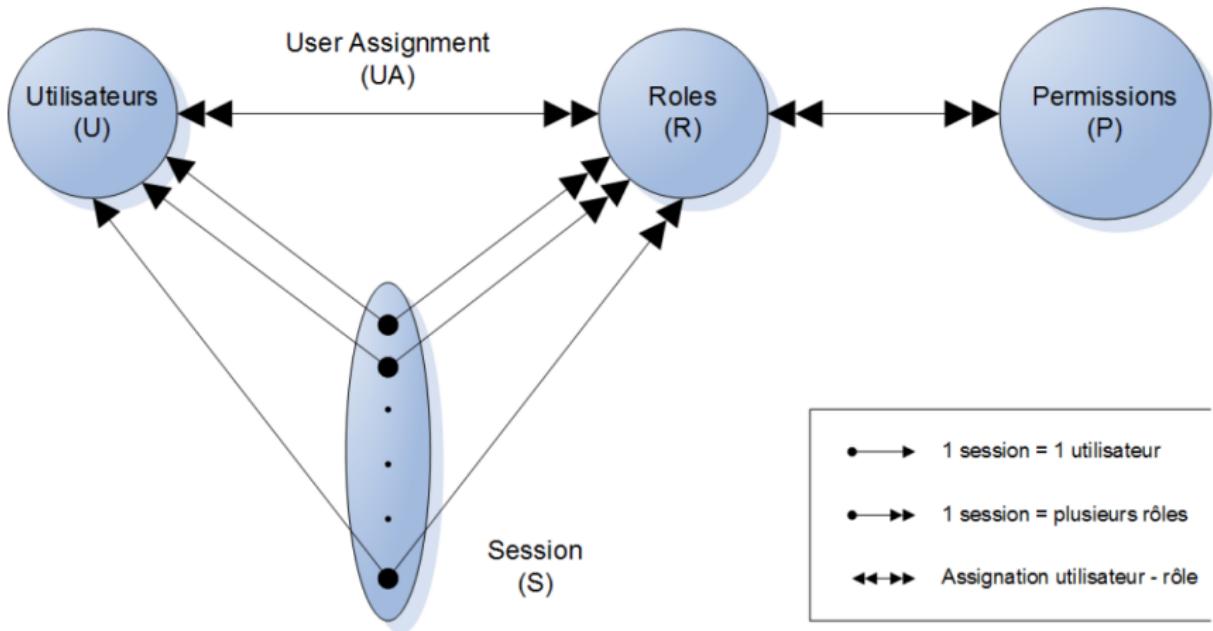
Le modèle Bell-LaPadula (she's my baby)

- Modèle de confidentialité pour l'accès à l'information dans la Défense ;
- Notion de label de sécurité : habilitation pour les sujets, classification pour les objets ;
- Notion de catégorie : répartition de l'information par type d'activité ;
- Deux règles de sécurité :
 - No Read Up
 - No Write Down

Le modèle Biba ('p-a-lula)

- Modèle de préservation de l'intégrité utilisé dans des systèmes industriels critiques ;
- Notion de niveau d'intégrité pour les sujets et objets ;
- Deux règles de sécurité :
 - No Read Down
 - No Write Up

Role-Based Access Control



Le projet *Linux Security Modules* (LSM) a pour objectif de fournir une API générique l'écriture de modules de sécurité.

- Support de tout type de mécanisme de sécurité (en théorie) ;
- API stable (enlève la nécessité de faire évoluer un patch) ;

Security Hooks

- Intégration de points de contrôle dans le noyau ;
- Au niveau des entrées vers les différents "sous-systèmes" du noyau Linux :
 - Fichiers (VFS) ;
 - Réseau ;
 - Gestion des processus ;
 - Mémoire partagée, IPC ;

Security Attributes

- Crédation d'un *namespace* dans les attributs étendus de fichiers, dédié aux informations de sécurité ;
- Nom et contenu dépendant du module de sécurité utilisé.

Modules actuellement intégrés : SELinux, AppArmor, Tomoyo, SMACK.

1 Protection contre les attaques

- De quoi veux-t-on se protéger ?
- Mécanismes de contrôle sur les instructions
- Mécanismes de restriction sur les appels système

2 Contrôle d'accès renforcé : MAC

- Modèles de politique de sécurité
- **La version Redhat : SELinux**
- La version Ubuntu : AppArmor

Security-Enhanced Linux

- Développement initial par la NSA (~ 1999) ;
- Intégration au noyau Linux vers 2001, donnant naissance au *framework LSM* ;
- Principes de conception
 - Séparation du composant de décision (*Security Server*) et du composant d'application (*hooks LSM*) ;
 - Labellisation persistante des fichiers ;
 - Règles d'accès indépendantes du système de fichiers ;
- Modèles de sécurité implémentés
 - RBAC, *Type Enforcement*
 - SELinux UID
 - Bell-LaPadula : *Multi-Level Security (MLS)*, *Multi-Category Security (MCS)*
- Format des contextes de sécurité
`user:role:type:levels.categories`

- Interface avec les appels système via LSM ;
- *Security Server (SS)* en espace noyau
 - Contient la configuration compilée (*policy.<version>*) ;
 - La configuration est modulaire ;
- Système de cache intégré : *Access Vector Cache (AVC)*
 - Les décisions d'accès sont calculées sous forme de vecteur de permissions ;
 - Dès qu'un vecteur est calculé par le SS, il est placé dans le cache ;
 - Les nouvelles entrées remplacent les plus anciennes ;
 - Nombre d'entrée dans le cache par défaut de 512 sur les systèmes RedHat ;
- Intégration dans les commandes standard
 - Code commun dans la *libselinux* ;
 - `ls -Z, ps -Z, id -a...`
- Commandes d'administration spécifiques
 - Etat : *sestatus, seinfo, semodule -l*
 - Compilation : *checkmodule, checkpolicy, semodule -i*
 - Labels : *setfiles, chcon, restorecon*

SELinux : définition des contextes de sécurité

Les modules de configuration se composent de trois parties :

- Définition de contextes (fichiers .fc);
- Macros publiques (fichiers .if);
- Règles d'accès (fichiers .te).

La définition des contextes à appliquer sur les fichiers se fait à l'aide d'expression régulières sur les chemins.

Par exemple :

```
HOME_DIR/.mozilla(/.*)? gen_context(system_u:object_r:mozilla_home_t  
,s0)  
/usr/lib(64)?/[^\/*firefox[^\/*firefox-bin -- gen_context(system_u:  
object_r:mozilla_exec_t,s0)
```

Après avoir chargé cette politique, on peut appliquer les contextes :

- Commande `setfiles` appliquée sur les différents systèmes de fichiers concernés;
- Ou touch `/.autorelabel` et redémarrage du système.

Langage de définition de règles d'accès

- Les règles d'accès comportent un sujet, un objet, une classe d'objet et un ensemble d'actions autorisées ;
- Ce qui n'est pas explicitement autorisé est interdit, et un message de *log* est généré ;
- On peut également définir des règles d'audit sur les actions autorisées ;
- Enfin on peut donner des contraintes à vérifier (mot-clé `neverallow`).

L'écriture des règles est (relativement) facilité par l'existence de nombreuses macros. Par exemple :

SELinux : règles de contrôle d'accès

```
type mozilla_t;
type mozilla_exec_t;
type mozilla_home_t;
application_domain(mozilla_t, mozilla_exec_t)
kernel_read_system_state(mozilla_t)
allow mozilla_t self:capability { sys_nice setgid setuid };
allow mozilla_t self:process { sigkill signal setsched getsched };
allow mozilla_t self:fifo_file rw_fifo_file_perms;
allow mozilla_t self:socket create_socket_perms;
allow mozilla_t mozilla_conf_t:file read_file_perms;
```

```
interface('kernel_read_system_state',
    gen_require(
        type proc_t;
    )
    read_files_pattern($1, proc_t, proc_t)
    read_lnk_files_pattern($1, proc_t, proc_t)
    list_dirs_pattern($1, proc_t, proc_t)
')
```

SELinux : en pratique

- SELinux est intégré dans la plupart des distribution Linux, notamment :
 - Fedora, RedHat → activé par défaut
 - Suse
 - Ubuntu, Debian
- On trouve en général quatre configurations :
 - la *Reference Policy* disponible sur le site de MITRE;
 - les déclinaisons *strict* et *targeted* créées par RedHat;
 - enfin la version MLS, seule version avec le support des niveaux.
- La configuration est placée dans /etc/selinux/<variante>.

1 Protection contre les attaques

- De quoi veux-t-on se protéger ?
- Mécanismes de contrôle sur les instructions
- Mécanismes de restriction sur les appels système

2 Contrôle d'accès renforcé : MAC

- Modèles de politique de sécurité
- La version Redhat : SELinux
- **La version Ubuntu : AppArmor**

AppArmor, le MAC simplifié

■ AppArmor

- Patch noyau développé par Immunix puis Novell pour Suse Linux (~1998)
- Récemment intégré au noyau Linux comme LSM, maintenu en particulier par Canonical (~2009)

■ Langage de configuration simple

- Définit des profils de confinement
- Un sujet est représenté par un chemin vers un exécutable
- Contrôle l'accès aux fichiers, au réseau, aux *capabilities*
- Peut désigner les fichiers dont le propriétaire est le même que celui du processus
- Définition de variables globales

- Chacun des profils peut être chargé en mode *enforcing* ou en mode *complain*
- Dossier de configuration : /etc/apparmor et /etc/apparmor.d

AppArmor, exemple de configuration

Extrait de la configuration pour Firefox :

```
#include <tunables/global>
/usr/lib/firefox-8.0/firefox{,*[^s][^h]} {
    #include <abstractions/ubuntu-browsers.d/firefox>
    /etc/firefox*/ r,
    /etc/firefox/** r,
    deny /usr/lib/firefox-8.0/** w,
    deny /usr/lib/firefox-addons/** w,
    owner @{HOME}/Downloads/ r,
    owner @{HOME}/Downloads/* rw,
    owner @{HOME}./.{firefox,mozilla}/ rw,
    owner @{HOME}./.{firefox,mozilla}/** rw,
    owner @{HOME}./.{firefox,mozilla}/**/*.{db,parentlock,sqlite}* k,
```

3 Conteneurs vs. Virtualisation

■ Contexte et définitions

■ Virtualisation

- Émulation complète
- Émulation niveau logiciel (type 2)
- Émulation niveau hôte (type 1)
- Disques durs virtuels
- Exemple de vulnérabilité

■ Conteneurs

- Premier pas : chroot()
- Mécanismes avancés du noyau Linux
- Sandbox
- LXC, LibContainer et Docker
- Exemple de vulnérabilité

Besoin

- Financier
 - Consommation électrique
 - Taux d'utilisation faible (moins de 15%)
 - Opérations de maintenance régulières (mises à jour, pannes)
 - Espace
- Simulation
 - Simuler des systèmes et réseaux complexes
 - Simuler des matériels spécifiques
 - Disposer d'une architecture de qualification
- Isolation
 - Isoler des programmes et services à risques
 - Créer des environnements non rémanents
- Pédagogique :)

Virtualisation

Faire fonctionner un ou plusieurs systèmes d'exploitation / applications comme un simple logiciel, sur un ou plusieurs ordinateurs - serveurs / système d'exploitation.

On parle de VM (Virtual Machine), VPS (Virtual Private Server) ou encore VE (Virtual Environnement)

À retenir

En virtualisation *matérielle*, on émule *complètement* une machine.

Conteneur / Isolateur

Un isolateur fait fonctionner des environnements isolés les uns des autres dans des *conteneurs* partageant **le même noyau** et une plus ou moins grande partie du système hôte.

Le conteneur apporte une **virtualisation de l'environnement d'exécution** (Processeur, Mémoire vive, réseau, système de fichier...) et non pas de la machine.

À retenir

Dans un conteneur, le programme tourne *directement* sur la machine

Notions employées

- Couches d'abstractions (matérielle / logicielle)
- Système hôte
- Système invité (ou virtualisé)
- Partitionnement / isolation
- Manipulation d'image
- Réseau virtuel

Principes associés

- Émulation (exemple : console de jeux)
- Prise d'instantané (*Snapshotting*)
- Migration / téléportation
- Licences

3 Conteneurs vs. Virtualisation

- Contexte et définitions
- **Virtualisation**
 - Émulation complète
 - Émulation niveau logiciel (type 2)
 - Émulation niveau hôte (type 1)
 - Disques durs virtuels
 - Exemple de vulnérabilité
- Conteneurs
 - Premier pas : chroot()
 - Mécanismes avancés du noyau Linux
 - Sandbox
 - LXC, LibContainer et Docker
 - Exemple de vulnérabilité

Émulation complète

- Aucune instruction de l'environnement virtualisé n'est exécutée sur le processeur
- Le microprocesseur, la mémoire de travail (RAM) ainsi que la mémoire de stockage sont émulés
- → Tout est émulé

Avantages et inconvénients

- Isolation totale
- Très lent

Exemples

- Qemu
- jor1k : émulateur en javascript
- Miasm

Hyperviseur de type 2

- Hyperviseur exécuté au sein d'un système d'exploitation
- Une partie des instructions (non critique, détectée statiquement/dynamiquement) est exécuté directement par la machine

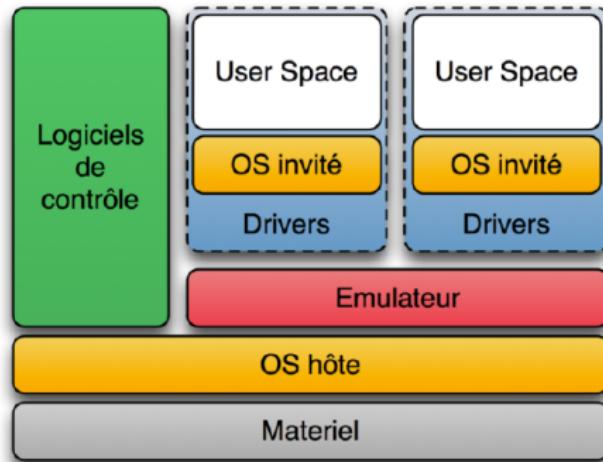
Avantages et inconvénients

- Isolation quasi totale
- Suffisamment rapide pour faire tourner correctement des applications
- Nécessite beaucoup de ressources matérielles

Exemples

- VirtualBox
- VMWare Workstation
- Parallels Desktop
- QEMU-KVM

Hyperviseur de type 2



- Pour améliorer les performances, support des instructions de virtualisation
- Sur x86 : VTX, AMD-V
- Manipulation *hardware* des espaces d'adressage mémoire (IOMMU, VT-x)
- Limitation de la mémoire RAM de l'*host* : *ballooning*

```
$ egrep '(vmx|smx)' /proc/cpuinfo
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mttr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
xsave avx f16c rdrand lahf_lm abm ida arat epb pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
xsaveopt
```

Hyperviseur de type 1

- Hyperviseur léger exécuté directement sur la machine
- *Bare metal*
- Tous les systèmes d'exploitation s'exécutent dans un environnement virtualisé

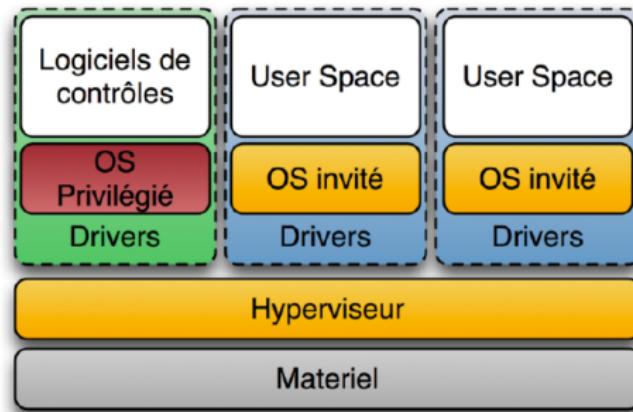
Avantages et inconvénients

- Tout est virtuel
- Idéal pour le partage des ressources d'une machine
- Aucune application ne peut tourner directement sur la machine

Exemples

- VMWare vSphere (anciennement VMWare ESX)
- Xen
- Hyper-V

Hyperviseur de type 1



Para-virtualisation

Lorsque le système d'exploitation invité a conscience d'être dans une machine virtuelle, on parle de para-virtualisation

Disponibilité

- Projet *Xen Windows GPLPV project* : kit de para-virtualisation de drivers pour Windows
- *paravirt-ops* (IBM, VMware, Xen, and Red Hat) : interface entre l'hyperviseur et le noyau de l'hôte
- Disponible depuis Linux 2.6.23

Idée

- La VM a besoin de voir des disques virtuels
- Ces disques virtuels doivent être “stockés” quelque part

Formats

- Tout le monde a son format
- → OVA, OVF : *Open Virtual machine Archive/Format*
- Migration inter-format risquée
- Montage possible

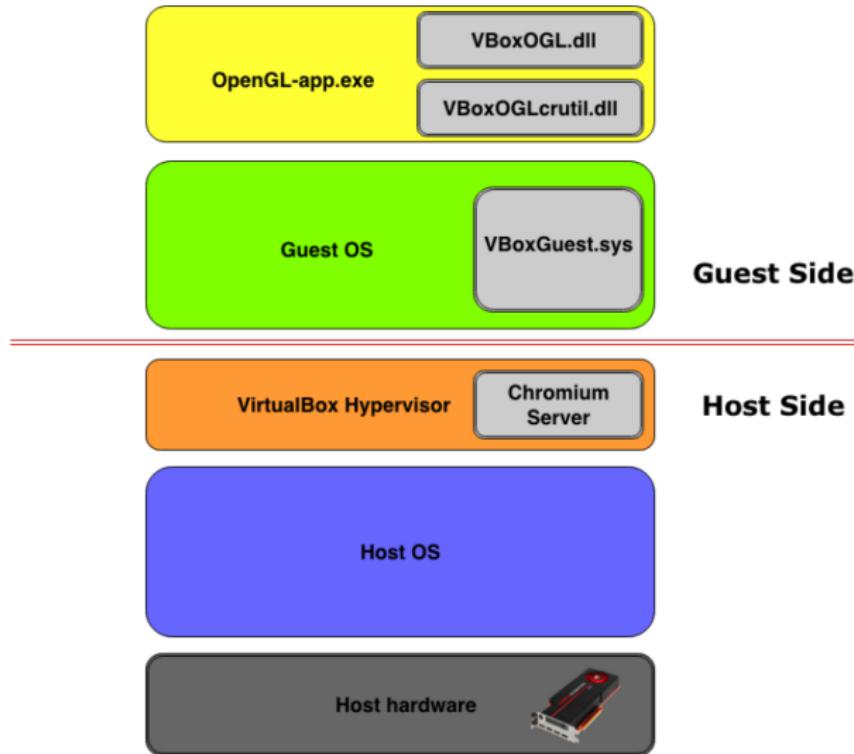
Technologies

- Block device
 - Partition
 - LVM
 - DD physique
- Fichier
 - Statique (thick-provisioning)
 - Dynamique (thin-provisioning)
- Compression
- Snapshoting (incrémental)

Breaking out of VirtualBox through 3D Acceleration

- Francisco Falcon - Recon 2014
- Guest Additions : Accélération 3D
- Installe un driver sur le guest, et communique avec l'host via des I/O
- Accélération 3D : implémentation de Chromium (!= navigateur) → client / serveur

Exemple de vulnérabilité : Virtualbox accélération 3D



Breaking out of VirtualBox through 3D Acceleration

- Possibilité de demander d'allouer des buffers de taille arbitraire
 - Normalement, pour passer des paramètres
 - → heap spraying!
- Le protocole utilise des “network pointers”
- ... CVE 2014-0981, CVE 2014-0982, CVE 2014-0983
- Exploitation
 - Structure mémoire un peu complexe ...
 - Write-What-Where + Heap spraying
 - → Exécution de code sur l'host
 - Amélioration de VUPEN : pas de crash

LibVirt

- API commune pour la manipulation de solutions de virtu hétérogènes
- Pour certaines solutions, support des disques durs, réseaux virtuels, ...
- QEMU-KVM, XEN, LXC, OpenVZ, UML, VirtualBox, VMWare, Hyper-V, ...
- Wrapper C, Python
- CLI : virsh
- GUI : virt-manager

```
#!/usr/bin/env python
import libvirt

## Hypervisor hosts
hv = [ "tiffy.tuxgeek.de", "ernie.tuxgeek.de" ]

for hv_host in hv:

    uri = "qemu+ssh://virtuser@" + hv_host + "/system"
    conn = libvirt.openReadOnly(uri)

    hypervisor_name = conn.getHostname()

    print "The\u00a0following\u00a0machines\u00a0are\u00a0running\u00a0on :\u00a0" + hypervisor_name

    # List active hosts
    active_hosts = conn.listDomainsID()
    for id in active_hosts:
        dom = conn.lookupByID(id)
        print "System\u00a0" + dom.name() + "\u00a0is\u00a0UP."
```

3 Conteneurs vs. Virtualisation

- Contexte et définitions
- Virtualisation
 - Émulation complète
 - Émulation niveau logiciel (type 2)
 - Émulation niveau hôte (type 1)
 - Disques durs virtuels
 - Exemple de vulnérabilité
- Conteneurs
 - Premier pas : chroot()
 - Mécanismes avancés du noyau Linux
 - Sandbox
 - LXC, LibContainer et Docker
 - Exemple de vulnérabilité

Idée

On fait passer un répertoire du système pour la racine (/) de celui-ci, l'application ne pouvant (en théorie) rien voir en dehors de ce répertoire.

Remarque : Bon entraînement avant de passer à SELinux ou grsecurity !

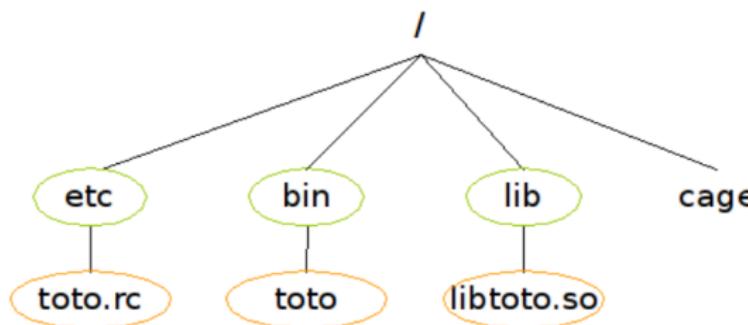
Virtualisation de la racine du système de fichiers : chroot()

- Exemple de mise en cage avec chroot

- Fichier exécutable : /bin/toto
- Configuration : /etc/toto.rc
- Bibliothèque : /lib/libtoto.so
- Nouvelle racine : /cage

- On reproduit l'arborescence sous /cage

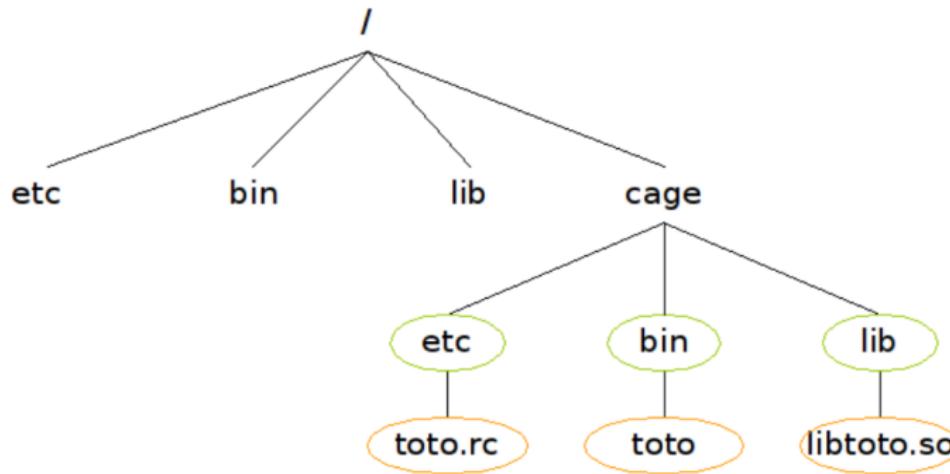
- mkdir /cage/etc
- mkdir /cage/bin
- mkdir /cage/lib



Virtualisation de la racine du système de fichiers : chroot()

- On déplace les fichiers du service :

- cp /bin/toto /cage/bin/toto
- cp /etc/toto.rc /cage/etc/toto.rc
- cp /lib/toto.so /cage/lib/libtoto.so



Virtualisation de la racine du système de fichiers : chroot()

On démarre enfin le programme comme suit :

```
chroot /cage /bin/toto [options]
```

Comment déterminer les fichiers nécessaires au programme et devant se trouver dans la cage ?

On peut distinguer deux cas :

- les bibliothèques dynamiques
 - liste avec ldd
 - on les copie dans l'arborescence de la cage
 - ou on compile le programme avec l'option -static
- le reste
 - les divers fichiers accédés par le programme lors de son exécution peuvent être déterminés avec un outil tel que strace

Certains programmes ont des options d'exécution pour faire appel à chroot() .

Limites de chroot()

On peut difficilement considérer `chroot()` comme un outil de sécurité à part entière

- Plusieurs possibilités d'évasion ;
- Ne pas faire tourner de code *root* à l'intérieur, ce n'est pas assez résistant ;
- → Démo !

→ Toutefois, ce mécanisme peut être renforcé par des améliorations dans grsecurity.

Principe général du conteneur sous Linux

- Gestion de vues partielles du système ;
- Rattachées à un ou plusieurs groupes de processus ;
- Toutes les ressources peuvent être restreintes :
 - Système de fichiers (*Mount points*) ;
 - Processus voisins ;
 - Réseau (interfaces et adresses) ;
 - Nom d'hôte (*hostname*) ;
 - Mapping des utilisateurs ;
 - Mémoire vive, périphériques matériels...

Deux mécanismes sous Linux

- *Control Groups ou cgroups*
 - Restrictions sur les processeurs utilisables, la mémoire vive et les périphériques matériels, arrêt / pause de groupes de processus ;
- *Namespaces*
 - Restriction de la vision sur les fichiers, les processus, le réseau, les utilisateurs, le nom d'hôte, les communications inter-processus.

Manipulation des Control Groups

- Interface sysfs : contrôle via le pseudo-système de fichiers monté dans /sys/fs/cgroup ;
- Fournit un ensemble de « contrôleurs », qui sont les types de ressources gérées par cgroups.

```
% ls /sys/fs/cgroup
blkio/  cgmanager/  cpu/  cpuacct/  cpuset/  devices/  freezer/
       hugetlb/  memory/  perf_event/  systemd/
```

- Outils cgroup : commandes cg* qui encapsulent l'accès direct aux fichiers dans /sys/fs/cgroup.

Manipulation des Control Groups

■ Création d'un cgroup

- Création d'un groupe nommé test1 à la racine : deux techniques
 - # mkdir /sys/fs/cgroup/cpuset/test1
 - # cgcreate -g cpuset:test1

■ Configuration du groupe test1

- # echo 2-3 > /sys/fs/cgroup/cpuset/test1/cpuset.cpus
- # cgset -r cpuset.cpus=2-3 test1

■ Affichage de la configuration du groupe test1

- # cat /sys/fs/cgroup/cpuset/test1/cpuset.cpus
- # cgget -r cpuset.cpus=2-3 test1

Implémentation des Control Groups

- Chaque processus est associé dans le noyau à une structure `task_struct`;
- Cette structure possède un champ de type `struct css_set` qui référence le control group associé à ce processus;
- CSS veut dire Cgroup Subsys State (donc état d'un sous-système de cgroup, qui peut être comme vu précédemment cpuset, devices, memory...).

Manipulation des Namespaces

- En espace utilisateur, la manifestation des namespaces se fait lors de la création d'un nouveau processus ;
- Les namespaces ne peuvent pas être visualisés facilement, ils se manifestent concrètement par une vision restreinte des ressources accessibles ;
- Lors de la création d'un nouveau processus (appel système `clone()`), le processus père peut donc limiter les ressources accessibles par le fils.
- Un processus ne peut que diminuer les ressources visibles par ses fils, jamais en ajouter.

Il existe 6 namespaces : Mount Points, UTS (nom d'hôte), Network (interfaces, adresses, règles de filtrage), IPC, PID (autres processus visibles) et Users (correspondance processus - UID propriétaire).

Création d'un namespace

- Pour créer un nouveau namespace, il faut passer des paramètres particuliers à la fonction `clone()` :
 - `CLONE_NEWNS` : le premier implémenté, il concerne les mount points visibles, il généralise le principe de `chroot()` ;
 - `CLONE_NEWUTS`
 - `CLONE_NEWIPC`
 - `CLONE_NEWPID`
 - `CLONE_NEWNET` : ces 4 paramètres implémentent l'équivalent de `jail` sur les systèmes FreeBSD ;
 - `CLONE_NEWUSER` : essentiel, ce mécanisme permet d'avoir des uid différents associés à un même processus suivant le namespace (par exemple, root dans un conteneur, non privilégié en dehors).

Commandes associées

- La commande `unshare` permet de créer un nouveau processus en réinitialisant une ou plusieurs catégories de namespace.
 - Exemple pour avoir un hostname différent dans un conteneur :

```
# unshare -u bash  
# hostname new_hostname
```

- Cette série de commandes ne change pas le hostname visible en dehors du conteneur créé par `unshare`.
- Il reste beaucoup plus facile et efficace d'utiliser un moteur de gestion de conteneurs de haut niveau comme LXC ou Docker.

Certains programmes sont particulièrement vulnérables parce qu'ils traitent des données complexes, comme les moteurs de rendu HTML ou PDF.

→ Ce type de code peut être isolé avec le mode seccomp.

- **Mode strict** : après avoir été placé dans le mode seccomp, un processus est limité à quatre appels système : `read`, `write`, `exit` et `sigreturn`.
- En résumé, ce processus peut uniquement utiliser des *file descriptors* déjà ouverts ou se terminer.
- Ce mode est pensé au départ pour des codes complexes tels que des machines virtuelles Java ou Flash, ou des codes de calcul intensif.

- **Mode avec filtre** : lors du passage en mode seccomp, le programme passe en argument un filtre de type BPF qui déterminera dynamiquement quels sont les appels système autorisés ou interdits ;
- Ainsi on peut généraliser la restriction des appels système.

Configuration du mode seccomp

- Le processus doit d'abord ouvrir tous les fichiers nécessaires ;
- Le changement de mode se fait par la commande `prctl`, qui est appelée par un processus pour changer ses paramètres de fonctionnement ;
- Mode strict : `prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT)`
- Mode filtre : `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, bpf_prog)`
- Si le processus utilise un appel système non autorisé, il reçoit immédiatement un `SIGKILL` et se termine.
- Nurse de Chrome

Interfaces de haut niveau

La manipulation directe des mécanismes cgroups et namespaces est plutôt fastidieuse, et donc plusieurs interfaces de haut niveau ont été créées.

Parmi celles-ci, on trouve :

- **LXC** pour LinuX Containers ;
 - Automatisation de la configuration cgroups, namespaces et dans une moindre mesure de la création du système de fichiers ;
- **Docker** ;
 - Automatisation poussée de la gestion de systèmes de fichiers par conteneur ;
- **Libvirt** (via LXC) ;
 - Plutôt orienté gestion de machines virtuelles, ne permet pas de configurer finement les mécanismes cgroups et namespaces.

LXC : LinuX Containers

LXC est la première interface de haut niveau créée pour gérer des conteneurs sous Linux. Elle simplifie grandement la manipulation des cgroups et namespaces.

Les éléments importants :

- Le *repository* de conteneurs, généralement dans le dossier `/var/lib/lxc/`, puis chaque conteneur est un sous-dossier ;
- Le fichier de configuration, qui indique la configuration cgroups et namespaces, ainsi que l'emplacement du rootfs spécifique s'il existe ;
- Le rootfs spécifique du conteneur, qui peut contenir une distribution entière, seulement quelques applications, ou être vide.

Ces éléments se trouve généralement sous :

```
/var/lib/lxc/<name>/config  
/var/lib/lxc/<name>/rootfs
```

LXC : LinuX Containers

Les commandes ont la forme `lxc-*`. Parmi les plus essentielles :

- `lxc-create` pour créer un conteneur, à partir d'un script d'automatisation. Plusieurs scripts sont fournis dans `/usr/share/lxc/templates/`, en particulier pour créer des conteneurs de distributions complètes ;
 - `# lxc-create -n <name> -t debian -- <template-options>`
 - `# lxc-create -n <name> -t ~/mon_script`
- `lxc-ls` pour obtenir la liste des conteneurs définis. Équivalent à un `ls /var/lib/lxc;`
- `lxc-destroy` pour effacer un conteneur précédemment créé avec `lxc-create`;

- lxc-start démarre un conteneur, en appelant la commande /sbin/init à l'intérieur ou une commande spécifiée ;
 - # lxc-start -n <name> -d
 - # lxc-start -n <name> /bin/bash
- lxc-execute exécute une commande spécifiée dans un conteneur, similaire à la deuxième forme de lxc-start ;
- lxc-attach exécute une commande spécifiée, ou à défaut un shell, dans un conteneur déjà démarré ;
- lxc-stop arrête ou redémarre un conteneur de façon douce, ou de façon brutale avec le paramètre -k ;
- lxc-freeze et lxc-unfreeze permettent de mettre en pause et de reprendre l'exécution de tous les processus d'un conteneur.

LXC : LinuX Containers

Le fichier de configuration d'un conteneur LXC définit les paramètres pour les mécanismes cgroups et namespaces par conteneur, et donc les ressources disponibles / accessibles dans ce conteneur. Pour la sécurité, il est important que cette configuration soit correcte, et donc la plus serrée possible.

Voici quelques paramètres classiques :

- `lxc.utsname` définit le hostname dans le conteneur;
- `lxc.cap.{keep|drop}` définit les *capabilities* à conserver (toutes les autres sont abandonnées) ou à abandonner (les autres sont conservées). Il est prudent d'avoir au minimum la ligne :
 - `lxc.cap.drop = sys_module mac_admin mac_override sys_time`
- `lxc.network.type` définit le type d'interface réseau disponible dans le conteneur. L'interface sera appelée par défaut `eth0` à l'intérieur;

LXC : LinuX Containers

- `lxc.rootfs` est l'emplacement du rootfs du conteneur, donc généralement
`/var/lib/lxc/<name>/rootfs`;
- `lxc.mount.entry` définit un mount point visible dans le conteneur, par exemple
 - `lxc.mount.entry = /bin bin none ro,bind 0 0`
 - `lxc.mount.entry = /lib lib none ro,bind 0 0`
- `lxc.mount` définit un fichier avec une liste de mount point à rendre visibles dans le conteneur.

Docker

- Déploiement automatique d'applications à l'aide de conteneurs
- Open source, écrit en Go
- En pleine expansion
- DockerHub : hub.docker.com

Principes

- Se base sur LXC ou LibContainer (par défaut)
- Les images Docker sont des suites de couches de fichier (UnionFS)
- Possibilité de faire de la gestion sur ces couches : pull/diff/commit/push
- Création de conteneurs scriptée : Dockerfile, *Automated build*

Architecture

- Démon local, écoute sur une socket Unix
- Pour parler sur cette socket, il faut être dans le groupe docker
- Pour le moment, c'est équivalent à être root
- Bridge docker0, gestion automatique des règles de pare-feu
- Système de *registries* privés et publics
- Couches situées dans /var/lib/docker

Exemple de commande

- docker images : liste les images installées
- docker pull ubuntu : télécharge l'image officielle ubuntu depuis le site de Docker
- docker run -i -t ubuntu bash : créer un conteneur basé sur l'image ubuntu, lance bash à l'intérieur et redirige les entrées/sorties
- docker diff : affiche les différences apportées par la dernière couche
- docker run -d -p 127.0.0.1:8080:9090 : lance docker en mode démon, et bind le port 8080 local avec le port 9090 du conteneur

Gestion des capabilities

- Les *capabilities* permettent de gérer les droits du super-utilisateur
- La capability CAP_DAC_READ_SEARCH permet de contourner les vérifications de droit lors d'un accès
- Elle s'utilise via l'appel système `int open_by_handle_at`

Vulnérabilité

- Docker utilisait une *blacklist* pour interdire des *capabilities*
- `/.dockerinit` est tout le temps présent, c'est un *handle* ouvert par le démon docker
- Brute-force des handles pour récupérer et lire les fichiers de l'host
- Exploit `shocker.c`

4 Applications

- Malware
- Provisioning
- Conteneurs
- Aide au développement
- Cloud
- Compartimentation

Blue pill

- But : écrire un rootkit indétectable, même lorsque l'on connaît son mécanisme
- Idée : l'host devient une machine virtuelle, Blue pill l'hyperviseur
- → la pilule bleue pour retourner dans la Matrix
- Repose sur les instructions de virtualisation
- Indétectable, pas de ralentissement, accès aux périphériques, ...



Détection d'un environnement virtualisé

Détection d'un environnement virtualisé

- Nom des interfaces et partages réseaux
- Modèle du disque dur
- Hook de fonctions système
- Faible taille de disque dur
- Souris immobile
- Nombre de CPUs
- Uptime
- Erreur de calcul de flottant

Détection d'un environnement virtualisé : Pafish

```

[-] Debuggers detection
[*] Using IsDebuggerPresent() ... OK

[-] CPU information based detections
[*] Checking the difference between CPU timestamp counters <rdtsc> ... OK
[*] Checking the difference between CPU timestamp counters <rdtsc> forcing UM exit ... OK
[*] Checking hypervisor bit in cpuid feature bits ... OK
[*] Checking cpuid vendor for known UM vendors ... OK

[-] Generic sandbox detection
[*] Using mouse activity ... OK
[*] Checking username ... OK
[*] Checking file path ... OK
[*] Checking common sample names in drives root ... OK
[*] Checking if disk size <= 60GB via DeviceIoControl() ... OK
[*] Checking if disk size <= 60GB via GetDiskFreeSpaceEx() ... OK
[*] Checking if Sleep() is patched using GetTickCount() ... OK
[*] Checking if NumberOfProcessors is < 2 via raw access ... OK
[*] Checking if NumberOfProcessors is < 2 via GetSystemInfo() ... OK

[-] Hooks detection
[*] Checking function DeleteFileW method 1 ... OK

[-] Sandboxing detection
[*] Using GetModuleHandle<sbi.dll> ... OK

[-] Wine detection
[*] Using GetProcAddress(wine_get_unix_file_name) from kernel32.dll ... OK

[-] VirtualBox detection
[*] Scsi port->bus->target id->logical unit id-> 0 identifier ... OK
[*] Reg key <HKEY\HARDWARE\Description\System "SystemBiosVersion"> ... OK
[*] Reg key <HKEY\SOFTWARE\Oracle\VirtualBox Guest Additions> ... OK
[*] Reg key <HKEY\HARDWARE\Description\System "VideoBiosVersion"> ... OK
[*] Reg key <HKEY\HARDWARE\ACPI\SDT\UBOX> ... OK
[*] Reg key <HKEY\HARDWARE\ACPI\Padt\UBOX> ... OK
[*] Reg key <HKEY\HARDWARE\ACPI\Reset\UBOX> ... OK
[*] Reg key <HKEY\SYSTEM\ControlSet001\Services\UBox*> ... OK
[*] Reg key <HKEY\HARDWARE\DESCRIPTION\System "SystemBiosDate"> ... OK
[*] Driver Files in C:\WINDOWS\system32\drivers\UBox* ... OK
[*] Additional system files ... OK
[*] Looking for a MAC address starting with 00:00:27 ... OK
[*] Looking for pseudo devices ... OK
[*] Looking for Ubox fire windows ... OK
[*] Looking for Ubox network share ... OK
[*] Looking for Ubox processes <vboxservice.exe, vboxtray.exe> ... OK

[-] VMware detection
[*] Scsi port 0.1.2 ->bus->target id->logical unit id-> 0 identifier ... OK
[*] Reg key <HKEY\SOFTWARE\VMware, Inc.\VMware Tools> ... OK
[*] Looking for C:\WINDOWS\system32\drivers\vmmouse.sys ... OK
[*] Looking for C:\WINDOWS\system32\drivers\vhgfs.sys ... OK

[-] Qemu detection
[*] Scsi port->bus->target id->logical unit id-> 0 identifier ... OK
[*] Reg key <HKEY\HARDWARE\Description\System "SystemBiosVersion"> ... OK

```

Analyse de malware

- Idée : faire tourner une cible et l'observer
- → l'environnement risque d'être infecté !
- Jeu du chat et de la souris
- Quid des échappements de VM ?

Exemple d'outil

- Cuckoo Sandbox
- Anubis
- Malwr.com

Principe

- 1 Création d'une VM (tip : avec un faux historique)
- 2 Snapshot de la VM dans un état *Clean*
- 3 Import de la cible
- 4 Mise en place de la chaîne d'observation
 - Réseau
 - Hooks systèmes
 - Base de registre
 - Écriture disque
 - Anti-détection de VM
 - ...

Principe

5 (Optionnel) Mise en place de faux éléments

- Faux serveur C&C (ex : IRC)
- Faux Internet
- ...

6 Lancement de la cible

7 Arrêt

- Timeout
- Inactivité

8 Création et exportation du rapport

9 Retour à l'état 2

Cuckoo Sandbox

Starts servers listening on 0.0.0.0:38917, 127.0.0.1:26093 → **Communications réseau**

File has been identified by at least one AntiVirus on VirusTotal as malicious

The binary likely contains encrypted or compressed data.

Executed a process and injected code into it, probably while unpacking

Collects information to fingerprint the system (MachineGuid, DigitalProductId, SystemBiosDate)

Detects VirtualBox through the presence of a file → **Sandboxing détecté !!!**

Creates Zeus (Banking Trojan) mutexes

Zeus P2P (Banking Trojan)

**Probablement un
dérivé de Zeus**

Creates a slightly modified copy of itself

Installs itself for autorun at Windows startup → **Persistance**

Cuckoo Sandbox

Hosts Involved

IP Address
8.8.8.8
81.236.49.249
194.9.95.75

DNS Requests

Domain	IP Address
gourmetfood.se	81.236.49.249
audiodirekt.se	194.9.95.75

Multiplexage d'AVs

- Différences de détection
 - Heuristique
 - Spécialisation
 - Format supporté
 - Méthodes (dynamique, ...)
 - Base de signature
 - Pays d'origine
- → Utilisation de plusieurs anti-virus !

Exemple

- VirusTotal.com
- IRMA (irma.quarkslab.com)

Principe

- 1 Démarrage de plusieurs VMs avec différents AVs
- 2 Analyse en parallèle
- 3 Récupération des résultats et conclusion
- 4 Restauration des VMs



SHA256: 083f7ca7eb64b4a3d897ac5e61dd3e0d67e47ea7e0447e817ed7d138209bf640

File name: 083f7ca7eb64b4a3d897ac5e61dd3e0d67e47ea7e0447e817ed7d138209bf640

Detection ratio: 28 / 48

Analysis date: 2013-09-17 06:35:44 UTC (6 days, 7 hours ago)



More details

[Analysis](#)[File detail](#)[Relationships](#)[Additional information](#)[Comments](#)[Votes](#)

Antivirus	Result	Update
Agnitum	✓	20130916
AhnLab-V3	Win-PUP/Helper.PrimeAd.911872	20130917
AntiVir	DR/Delphi.Gen	20130917
Anti-AVL	Trojan/Win32.Genome.gen	20130917

4 Applications

- Malware
- Provisioning**
- Conteneurs
- Aide au développement
- Cloud
- Compartimentation

Provisioning

- “Allocation automatique de ressources”
- Gestion du déploiement / maintien de la configuration
- Idée : configuration abstraite, “programmée”
- Utilisé pour mettre dans un état donné un conteneur, une VM

Produits

- Ansible
- Chef
- Saltstack
- Puppet
- ...

Provisionning

```
resolver::args:  
  domain: example.com  
  search:  
    - mon.domain  
    - monautre.domain  
nameservers:  
  - 192.168.1.250  
  
krb5::client::realms:  
  OLYMPE.COM:  
    kdc: kdc.olymppe.com  
    admin_server: adm.olymppe.com  
  
apache_vhosts:  
  'www.example.com':  
    ip: 192.168.1.100  
    port: 443  
    servername: 'www.example.com'  
    docroot: '/var/www/public/'  
    options:  
      - 'Indexes'  
      - 'FollowSymLinks'
```

Orchestrator

- Orchestre le déploiement complet
 - 1 Création des instances virtuelles
 - 2 Préparation des environnements (shares, réseau, ...)
 - 3 Déploiement des configurations → provisioning
- Permet de décrire des structures complexes, multi-services, multi-hosts

Vagrant

- Abstraction de la couche de virtualisation
 - VirtualBox, VMWare
 - Docker
 - AWS, Google Cloud Platform
- Support de plusieurs moteur de provisioning
 - Ansible, Puppet, Saltstack, ...
- Pas encore réellement prêt pour la production
- Vagrantfile, vagrant up

Vagrantfile

```
Vagrant.configure("2") do |config|
    config.vm.box = "precise32"
    config.vm.hostname = "myprecise.box"
    config.vm.network :private_network, ip: "192.168.0.42"

    config.vm.provider :virtualbox do |vb|
        vb.customize [
            "--modifyvm", :id,
            "--cpusexecutioncap", "50",
            "--memory", "256",
        ]
    end

    config.vm.provision :puppet do |puppet|
        puppet.manifests_path = "puppet/manifests"
        puppet.manifest_file = "site.pp"
        puppet.module_path = "puppet/modules"
    end
end
```

4 Applications

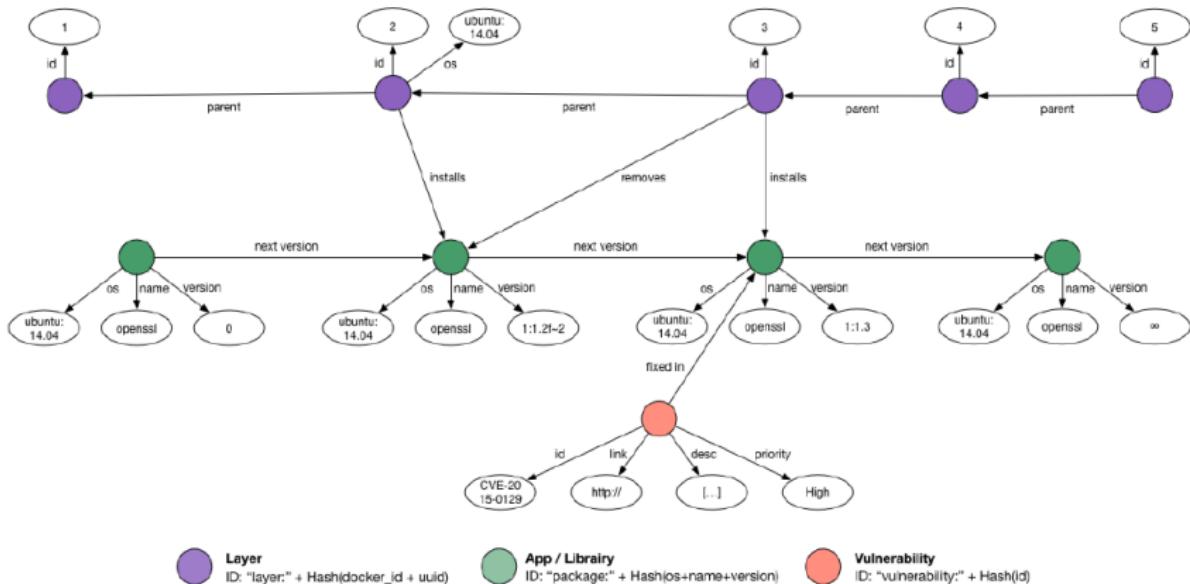
- Malware
- Provisioning
- Conteneurs**
- Aide au développement
- Cloud
- Compartimentation

Simulateur

- Projet interne (pour le moment)
- Idée : simuler, grâce à la faible empreinte des conteneurs :
 - plusieurs machines
 - plusieurs services
 - des configurations réseaux complexes
 - provisioning
- Intérêts
 - Simuler tout ou partie du réseau
 - Simuler des dysfonctionnements
 - Dev & Test du provisioning "from scratch"
 - ...

CLAIR

- Container Vulnerability Analysis Service
- Analyse des couches des containers à la recherche de vulns connues
- Stratégie
 - Check des hashes de tout les binaires → lent
 - Utilisation du versioning via les packages managers → rapide, incomplet



4 Applications

- Malware
- Provisioning
- Conteneurs
- Aide au développement**
- Cloud
- Compartimentation

Build automatique

- Build pour plusieurs architectures
- Avec parfois beaucoup de dépendances (NPM ?)
- Répétitif, historique non voulu
- → conteneurs, VMs
- Jenkins

Continuous Integration

- Souvent abrégé “CI”
- Build et lancement automatique de tests de régression
 - à chaque commit
 - avant de merger des branches
 - avant de relâché une version
 - sur différents systèmes (debian, centos, ...)
 - avec différentes versions (Python 2.7, 3.4, ...)
- Exemple : TravisCI, binding Github, conteneur et VMs

The screenshot shows a GitHub pull request page with a green checkmark icon indicating successful CI builds. The status message says "All checks have passed" with "1 successful check". It lists a single green checkmark for "continuous-integration/travis-ci/pr — The Travis CI build passed" with a "Details" link. Below that, another green checkmark says "This branch is up-to-date with the base branch" with the note "Merging can be performed automatically." At the bottom, there's a green button to "Merge pull request" or view command line instructions.

All checks have passed
1 successful check

continuous-integration/travis-ci/pr — The Travis CI build passed

This branch is up-to-date with the base branch
Merging can be performed automatically.

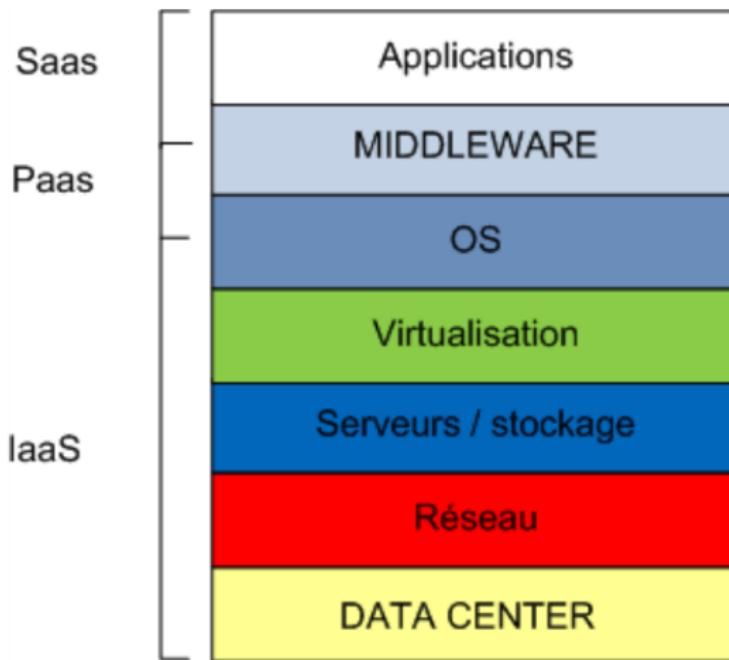
Merge pull request or view command line instructions.

4 Applications

- Malware
- Provisioning
- Conteneurs
- Aide au développement
- Cloud**
- Compartimentation

Cloud

- “Dans le nuage”
 - Hébergement de service
 - Gestion de la charge
 - anti-DDoS
 - Puissance de calcul (ex : Google Photos)
- Modèle économique à la ressource
- Infrastructure / Platform / Software as a Service



Cloud

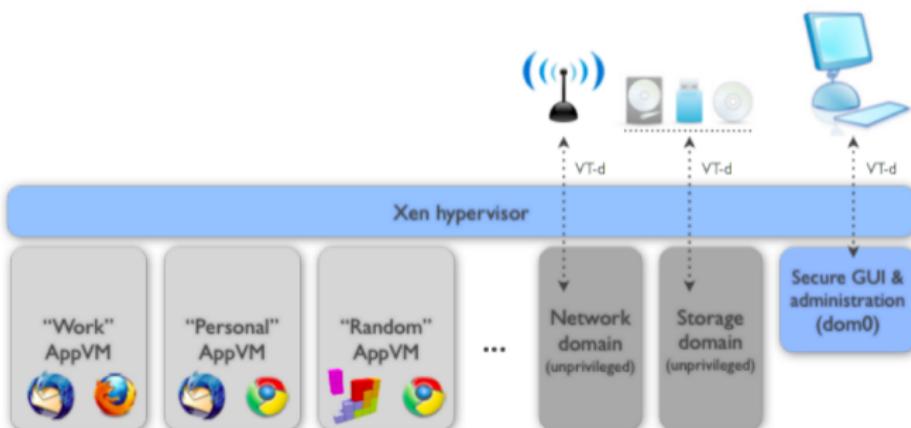
- Amazon Web Service, Google Cloud Platform, ...
- CloudStack, OpenStack, ...
- Fourniture de templates de machine
- Accès souvent via clé SSH
- Partage d'hyperviseur
- Partage de machine virtuelle (ex : hébergement Web)

4 Applications

- Malware
- Provisioning
- Conteneurs
- Aide au développement
- Cloud
- Compartimentation

Principe

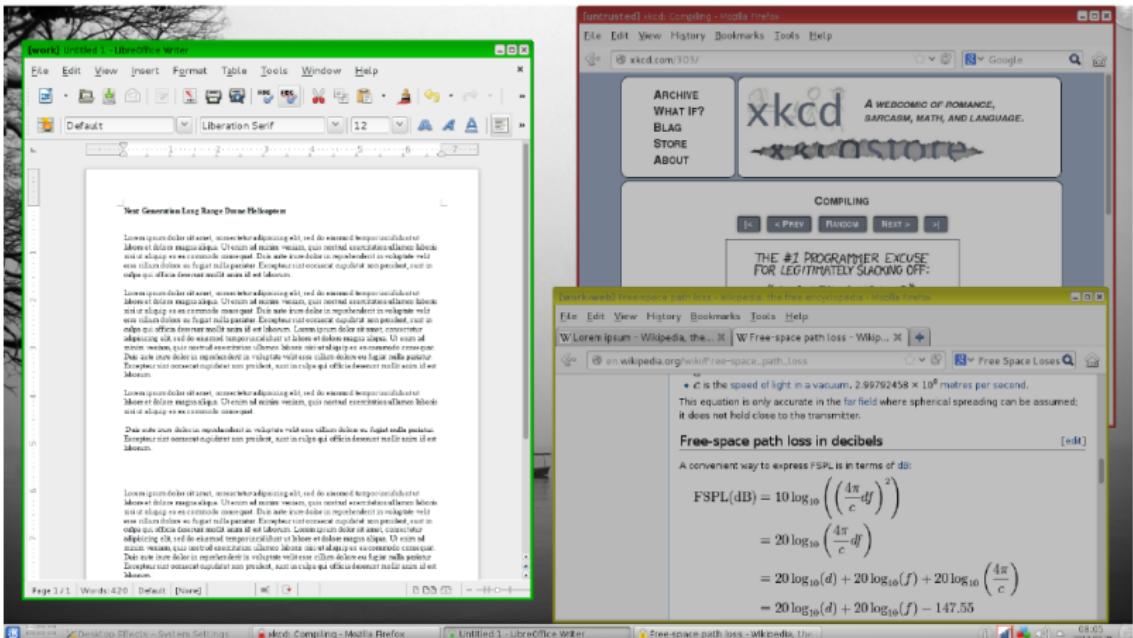
- Hyperviseur de type 1
- Utiliser la virtualisation pour compartimenter
- Intégrer le tout dans un environnement “utilisable”



Produits

- CLIP
 - ANSSI
 - Potentiellement permettre plusieurs niveau de classification (ACID, diode)
 - Mais seulement pour les opérateurs des OIV
- QubesOS (qubes-os.org)
 - Open-source
 - Basé sur XEN
 - Intégré aux classiques GUIs (XFCE, KDE, Windows, ...)
 - Expérimental / utilisable

Sécurité par la compartimentation



Des questions ?

Commissariat à l'énergie atomique et aux énergies alternatives
Centre de Bruyères-le-Châtel | 91297 Arpajon Cedex
T. +33 (0)1 69 26 40 00 | F. +33 (0)1 69 26 40 00
Etablissement public à caractère industriel et commercial
RCS Paris B 775 685 019

CEA DAM
DSSI
CTSI