Sécurité des OS et des systèmes

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Présentation des intervenants

- Aurélien Wailly
  - Amazon depuis 7 ans
  - Thèse : *End-to-end Security Architecture and Self-Protection Mechanisms for Cloud Computing Environments*
  - Sécurité OS, reverse-engineering, réponse à incidents
  - `http://aurelien.wail.ly`
  - `aureliew@amazon.com`

Sécurité des systèmes et des OS

- Objectifs
  - Fonctionnement des OS modernes, notamment du point de vue de la sécurité
  - Failles, exploitation, compromission
  - Dans le domaine de la sécurité, il est indispensable de comprendre comment marche un système pour l'exploiter
- Cibles
  - Linux, les principes restent valables pour les UNIX
  - Windows

Plan

1. Introduction
2. OS, CPU et sécurité
3. Mémoire, segmentation et pagination (et sécurité !)
4. Sécurité des utilisateurs - UNIX
5. Sécurité des utilisateurs - Windows
6. Analyse de binaires - Concepts et outils
7. Failles applicatives
8. Protection des OS

Votre OS ?



www.dilbert.com/

Ce qui ne sera pas abordé

La sécurité est un sujet vaste !

- Réseau
- Virus et malwares
- Techniques logicielles avancées
  - Shellcoding poussé
  - Protection logicielle
- Exploitation Web
  - Cross Site Scripting
  - Injections SQL

Un scénario d'attaque distante

- Scan réseau et découverte de services
- Recherche de vulnérabilités dans le programme
- Exploitation : exécution de code non-privilégié
- Elevation de privilèges
- Persistence : *backdoors* et *rootkits*

Dive deep

Plus de détails sur `https://ghostbin.com/paste/6kho7` avec l'exploitation de la Hacking Team.

Un scénario d'attaque locale

- Démarrage du système sur un *Live CD*
- Récupération du condensat du mot de passe
- Cassage du mot de passe
- Compromission du système

OS, CPU et sécurité

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

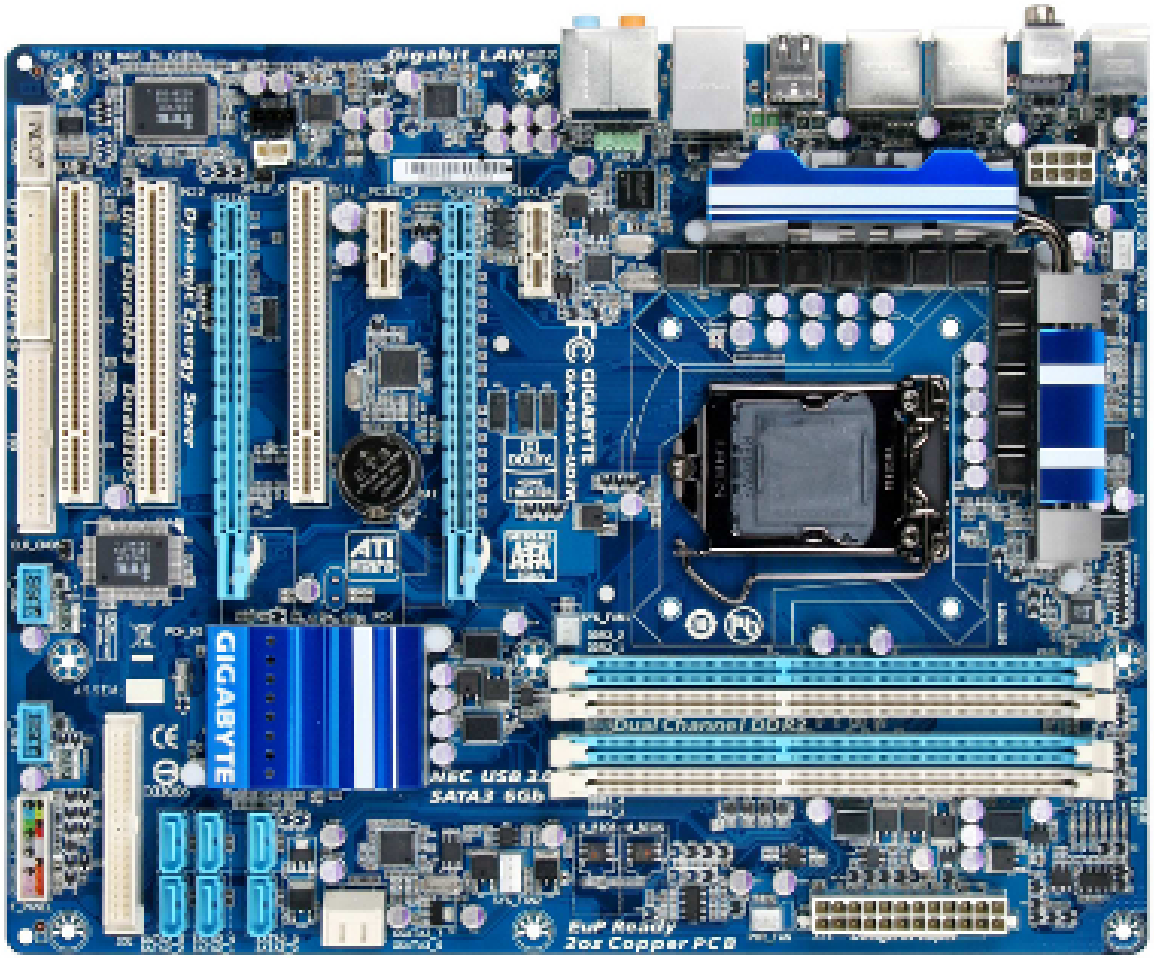
Tenable?
Paris

30 Mai 2022

Architecture des ordinateurs - Le boîtier



Architecture des ordinateurs - La carte mère



Architecture des ordinateurs - Vitesses de transfert

Rapide (ex-Northbridge)

- **CPU** / **RAM** : 20 GB/s
- **CPU** / **PCIe 16x** : 8 GB/s
- **CPU** / **South Bridge** : 2 GB/s

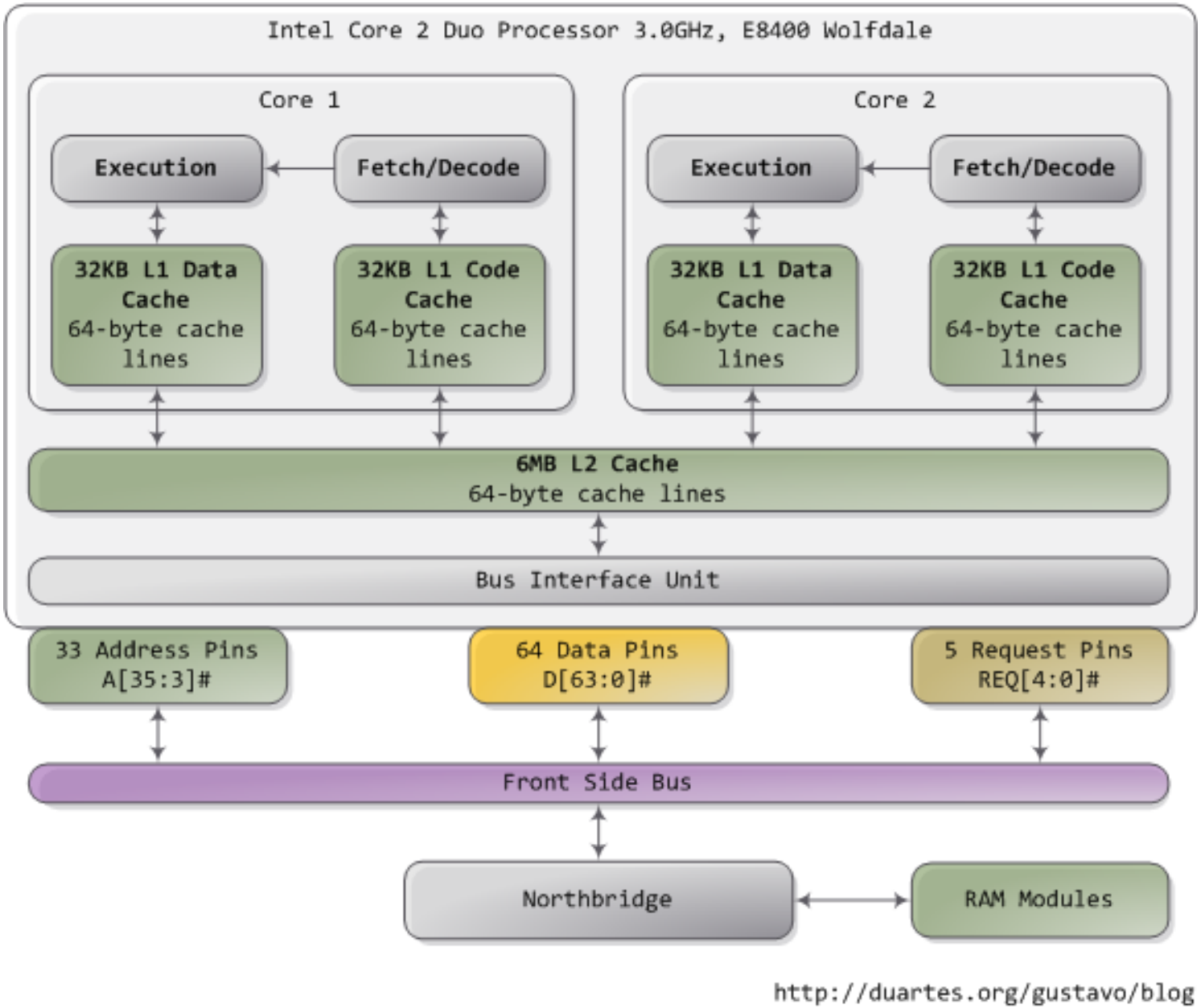
Lent (actuel Southbridge)

- **South Bridge** / **PCIe 1x** : 500 MB/s
- **South Bridge** / **USB** : 60 MB/s
- **South Bridge** / **SATA** : 300 MB/s

Architecture des ordinateurs - Le CPU

- Le CPU exécute du code présent en RAM.
- L'utilisateur interagit avec l'OS au moyen de périphériques qui génèrent des interruptions.

Architecture des ordinateurs - Le CPU



Operating System

- Ensemble de programmes qui gère les ressources de l'ordinateur :
  - Disque dur
  - Mémoire
  - Carte graphique
  - ...
- Fonctionnalités des OS modernes :
  - Processus
  - Interruptions
  - Gestion de la mémoire
  - Système de fichiers
  - Réseau
  - Droits des utilisateurs
  - ...

Le kernel

- **Kernel** : le composant essentiel de l'OS.
  - Chargé en mémoire au démarrage
  - Fourni les fonctionnalités essentielles de l'OS aux autres programmes
  - Gère les opérations bas niveau
  - Interface entre les applications et le hardware
- Garant de la sécurité de l'OS !
  - Contrôle d'accès aux ressources
  - Ordonnancement
  - Isolation entre les processus

Le kernel

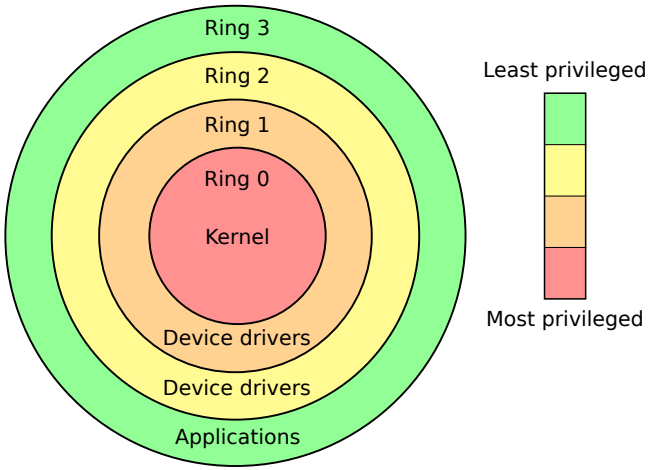
- Généralement écrit en C et assembleur
- Exemple pour Linux 2.6.37 :

code/linux_sloccount.txt

```
Totals grouped by language (dominant language first):
ansic:       8932526 (97.15%)
asm:          235122 (2.56%)
perl:          12143 (0.13%)
sh:             3743 (0.04%)
cpp:            3395 (0.04%)
yacc:           2993 (0.03%)
lex:            1825 (0.02%)
python:         1795 (0.02%)
awk:             708 (0.01%)
lisp:            218 (0.00%)
pascal:          121 (0.00%)
sed:              30 (0.00%)
```

Espace mémoire - Notion de rings

- Le noyau dispose d'un espace mémoire privilégié : **kernel-land**
- Les applications classiques s'exécutent en **user-land**
  - MS-DOS : les applications utilisent directement le hardware
  - OS modernes : les applications demandent au kernel, qui accepte ou non d'effectuer l'opération



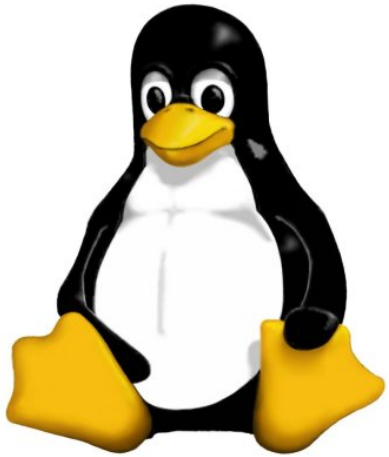
- Fonctionnalité du CPU
  - **Hardware !**
- Ring 0 : kernel et drivers
- Ring 3 : userland

Systèmes multi-utilisateurs

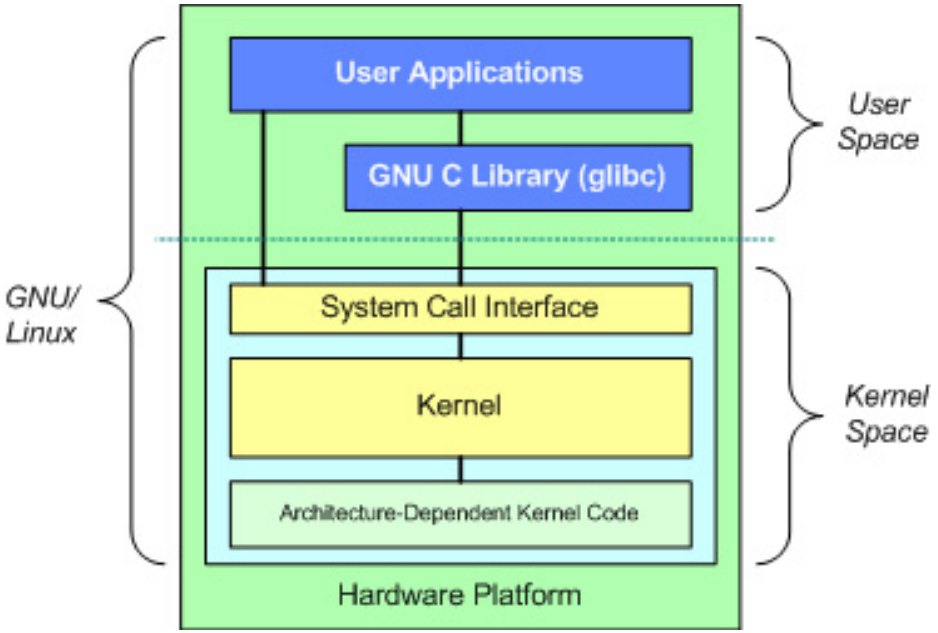
- Un OS moderne est capable d'exécuter simultanément des applications appartenant à plusieurs utilisateurs.
  - Partage des ressources (CPU, disque dur, . . .)
  - Une application userA n'a pas besoin d'avoir la connaissance d'une autre application userB .
  - Un bug dans userA ne doit pas perturber userB.
  - Un calcul compliqué dans userA ne doit pas ralentir userB (sauf si le kernel l'accepte).
  - userA ne doit pas être capable de regarder ce que fait userB (sauf si userB est d'accord).

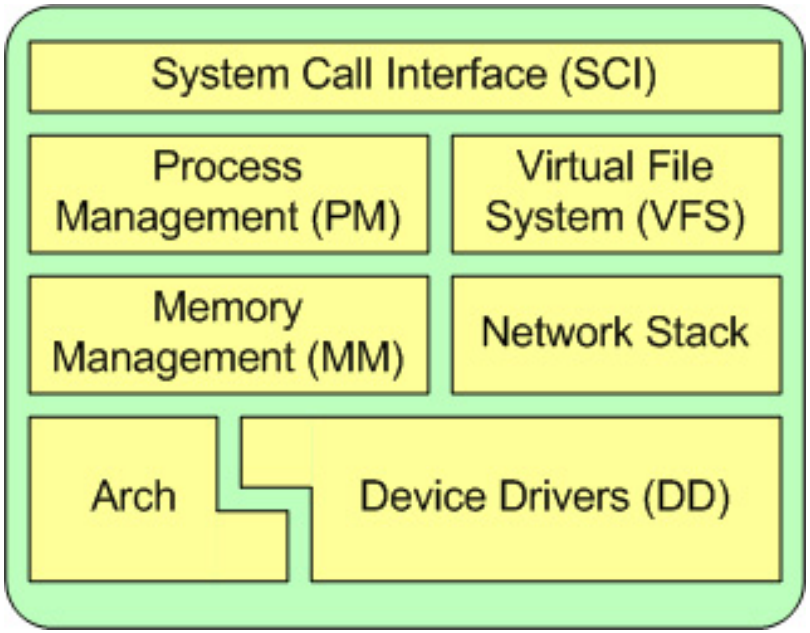
Le noyau Linux

- **Monolithique**
  - Un seul programme
  - Un seul espace d'adressage
  - Très courant pour les UNIX *(exception : Mac OS X, GNU Hurd)*
- **Modulaire** : drivers
- Préemptif
- *Free as in speech* : open-source, documenté, modifiable.
- *Free as in beer* : gratuit (attention, on parle du kernel et pas de l'OS !)

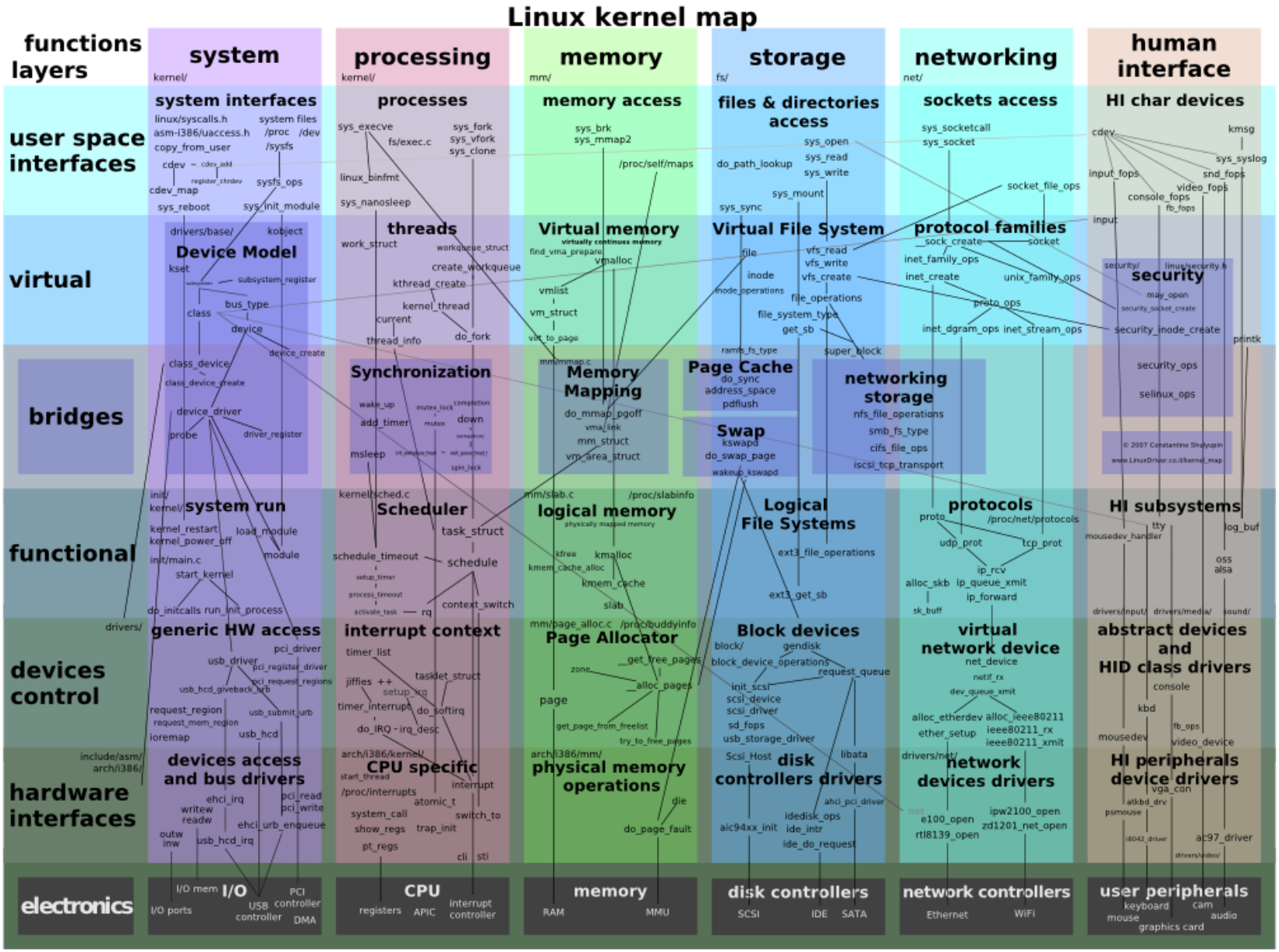


Le noyau Linux



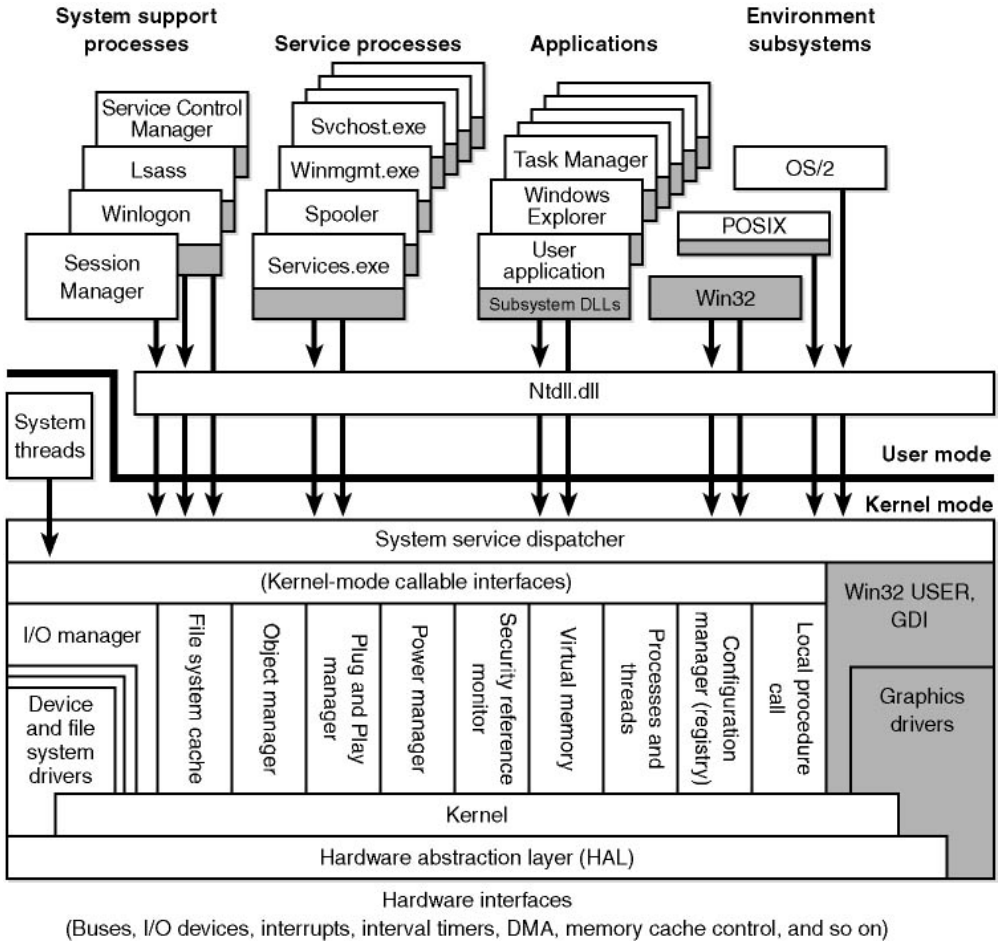


Le noyau Linux



Le noyau Windows

- **ntoskrnl.exe**
- **Micro-kernel hybride**
- Pas open-source : seule l'API offerte par les DLL système est documentée.



Notion de processus

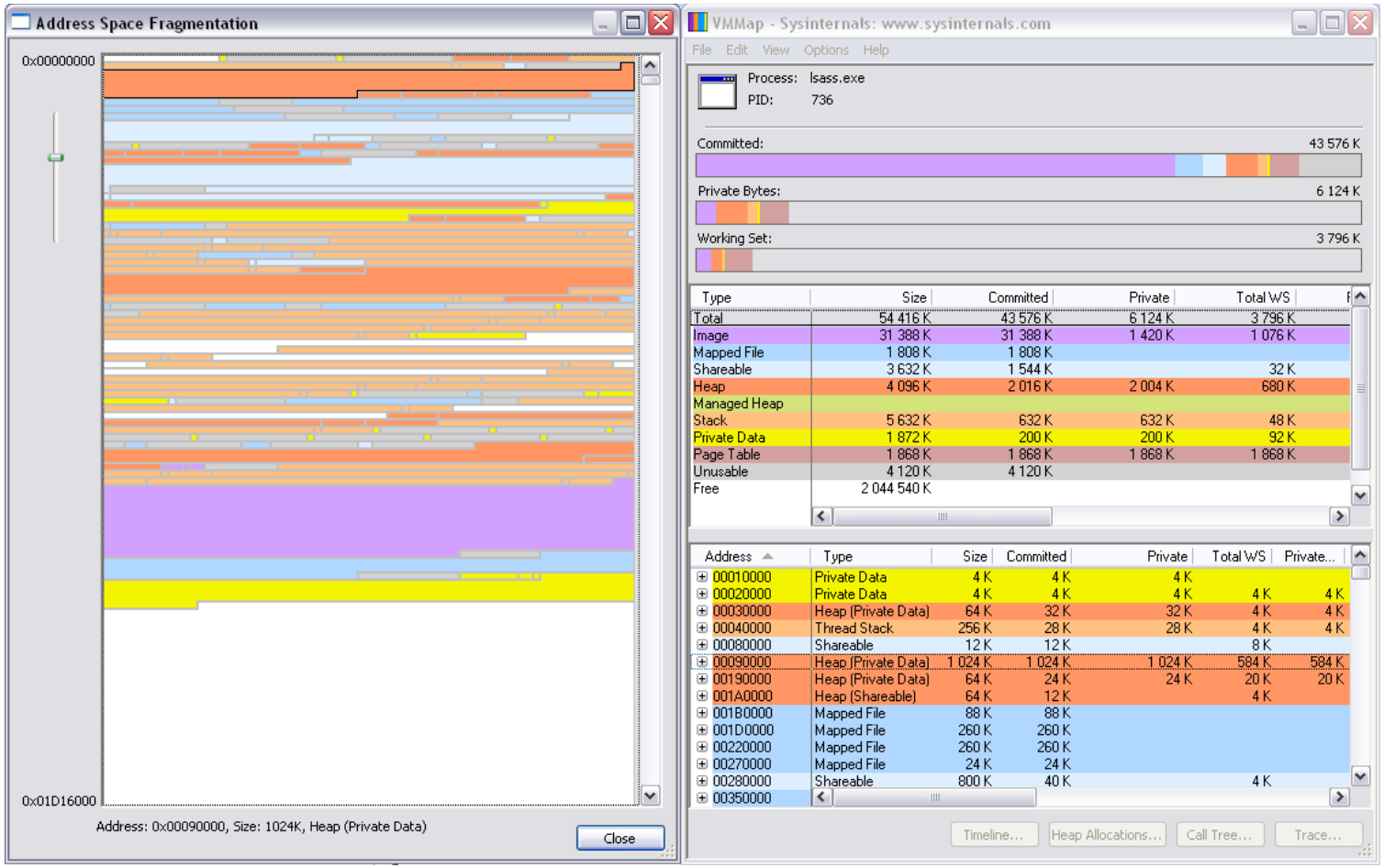
Processus

Instance d'un programme en exécution.

- Isolés : ne peuvent communiquer entres eux qu'au moyen de mécanismes mis en place par l'OS *(IPC : mémoire partagée, signaux, pipes, sockets)*.
- Chaque processus a l'impression qu'il est le seul à s'exécuter *(mémoire virtuelle)*.
- **UNIX** : commande *ps*
- **Windows** : application *Gestionnaire des tâches / vmmap*
- **PID** : process ID

Affichage processus

Démo : linux (ps + /proc/self)



Processus et changement de contexte

- Un CPU est capable d'exécuter une seule chose à la fois.
- **Scheduler** : détermine quel processus s'exécute.
- Quand on change de processus, il faut sauvegarder son état :
  - Registres, fichiers ouverts, mémoire, ...
  - C'est le **contexte** du processus.

- **Hiérarchie de processus**
  - Un processus peut en créer un autre *(processus fils)* dont il sera le *père*.
  - `fork() / execve()` sous UNIX
    - ⋆ *NOTE : distinction entre le processus et le programme qu'il exécute*
  - `CreateProcess()` sous Windows

Notion de thread

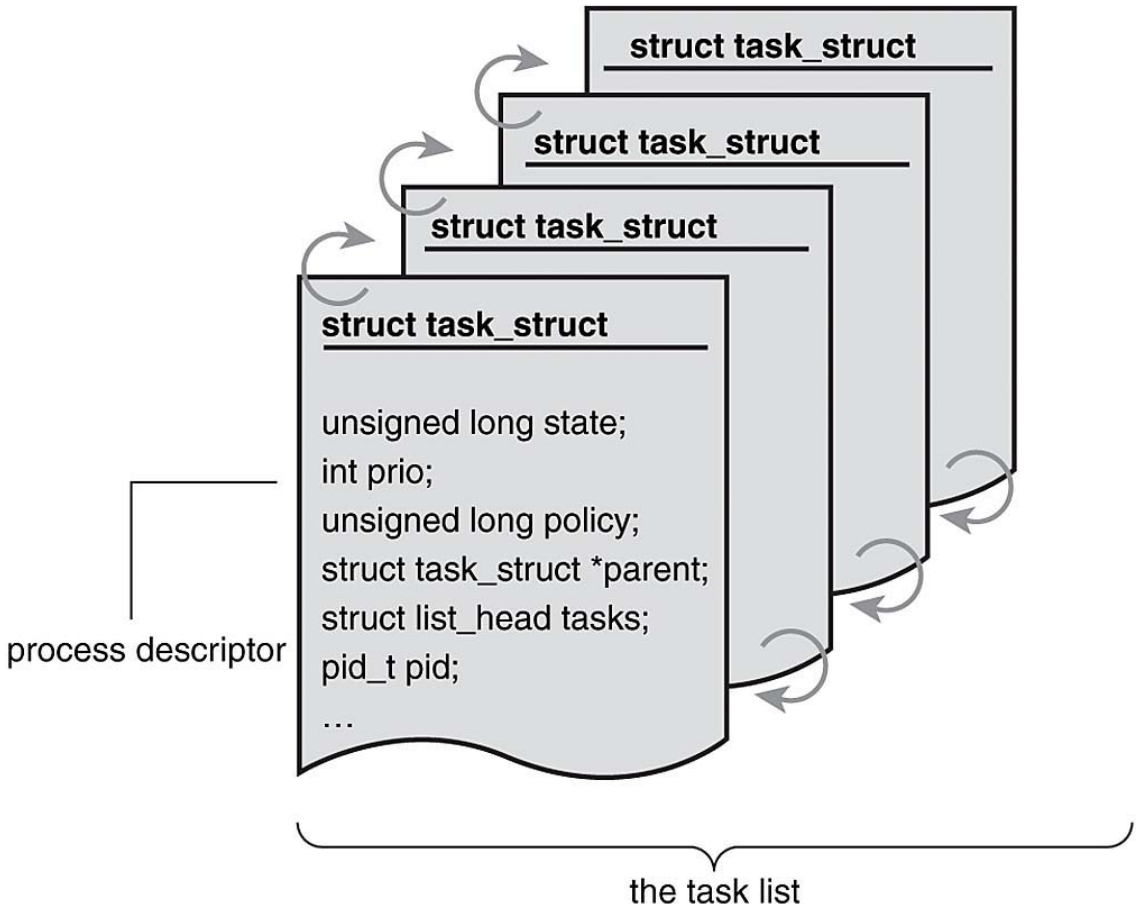
Thread

Fil d'exécution, processus léger.

- Changer de contexte est coûteux en cycles CPU.
- Les threads d'un même processus partagent le même espace mémoire.
- Implémentation sous Linux :
  - Processus == Thread pour le kernel *(task)*
  - Création d'un process :
    ```
    clone(SIGCHLD, 0);
    ```
  - Création d'un thread :
    ```
    clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
    ```

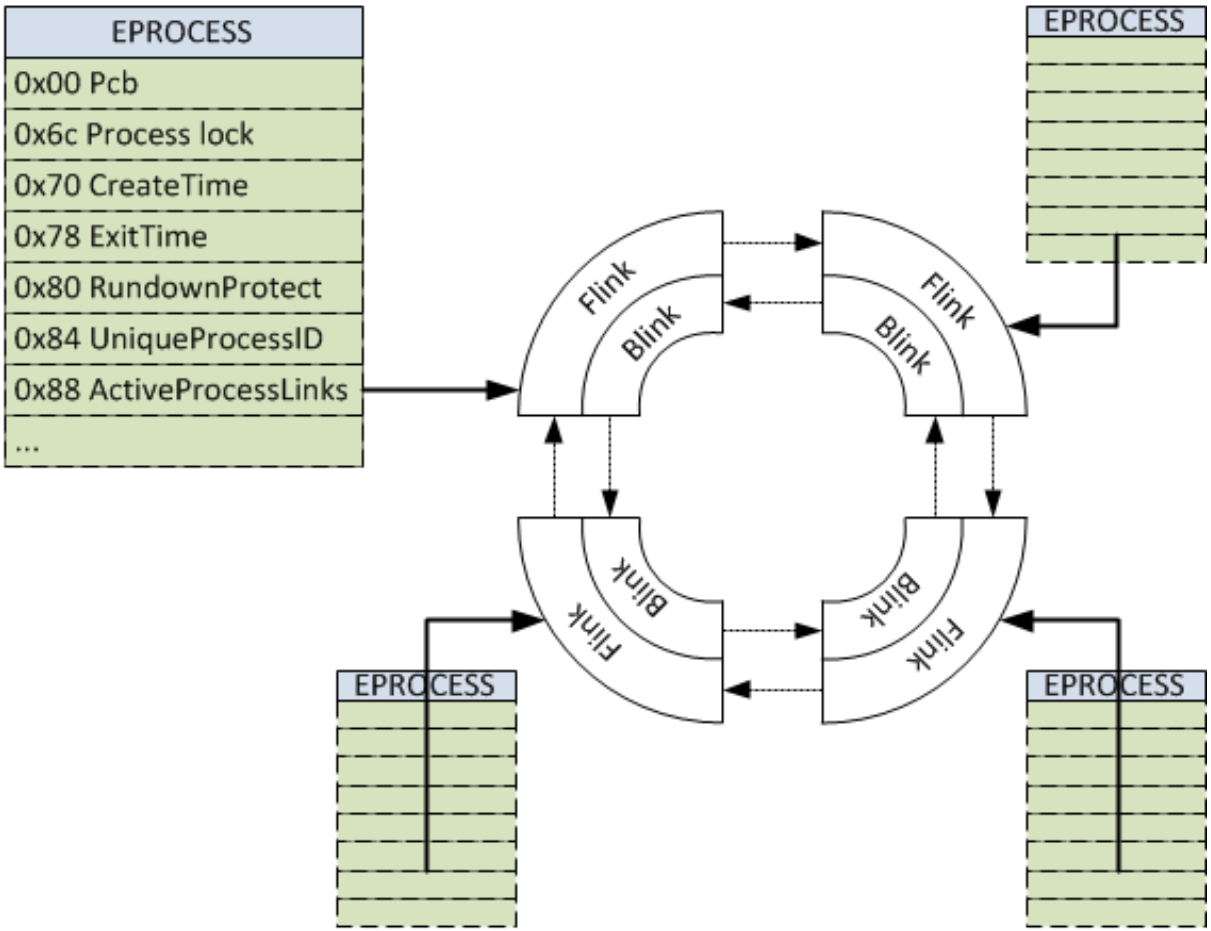
Gestion des processus - Linux

- Le noyau garde une **liste circulaire doublement chaînée**.
- Chaque élément est une *task_struct*.



Gestion des processus - Windows

- Chaque processus est représenté par une structure EPROCESS
- Chaque structure contient un pointeur vers un élément d'une **liste circulaire doublement chaînée**.



Récapitulatif - Le kernel

- **Composant logiciel** principal d'un OS
- **Interface** entre le hardware et les applications "classiques" :
  - Firefox ne peut pas accéder à la carte réseau (nous verrons pourquoi dans le chapitre suivant)
  - Il demande au kernel d'envoyer et de recevoir des paquets réseau pour lui
  - Il demande à la carte graphique d'afficher le rendu des pages web
- Garant de la sécurité



Récapitulatif - Les processus

- Un processus est l'**instance d'un programme** en exécution
- Comprend le **code** exécuté, ainsi que toutes les **ressources** annexes :
  - *fichiers ouverts, valeurs stockées dans les registres, signaux pas encore délivrés, . . .*
- **Isolés** : les processus ne peuvent communiquer entre eux que par des mécanismes mis en place par l'OS
  - Communication directe impossible
  - Utilisation des IPC fournis par le kernel



Références

- Cours de Sécurité OS, Telecom ParisTech 2010, Ryad Benadjila
- *The Linux Programming Interface*, Michael Kerrisk
- *Understanding The Linux Kernel, 3rd Edition*, Daniel P. Bovet, Marco Cesati
- *Linux Kernel Development*, Robert Love
- `http://www.ibm.com/developerworks/linux/library/l-linux-kernel/`
- Wikipedia
- `man clone`
- CanardPC Hardware
- `http://download.sysinternals.com/files/VMMap.zip`

MMU et OS

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

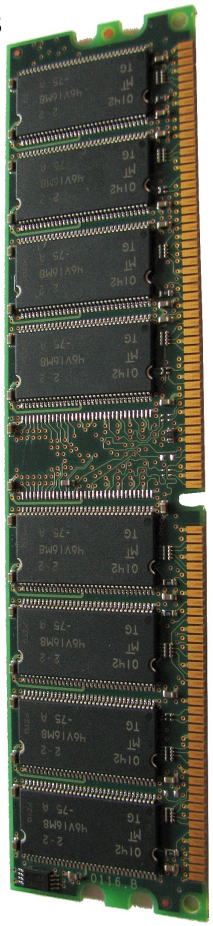
Tenable?
Paris

30 Mai 2022

Objectif

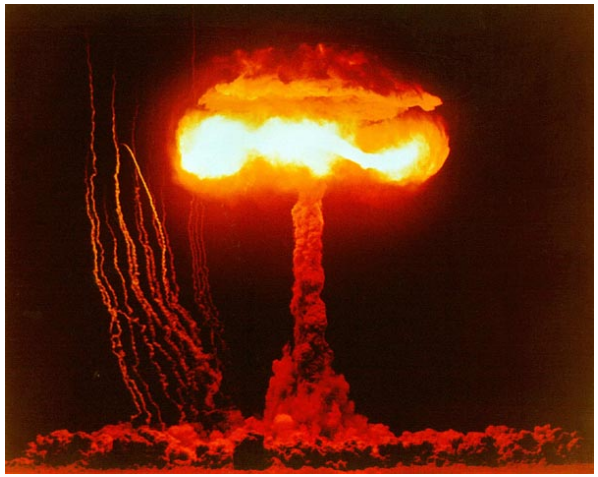
- **Objectif** : décrire comment les OS cloisonnent les processus dans leur propre espace mémoire.
- L'OS utilise une partie du CPU *(hardware)* : **MMU** (Memory Management Unit).
- La MMU est commune à une grande partie des CPU modernes.
  - Dans notre cas, **x86**.
- Valable pour Linux et Windows.

Mémoire physique

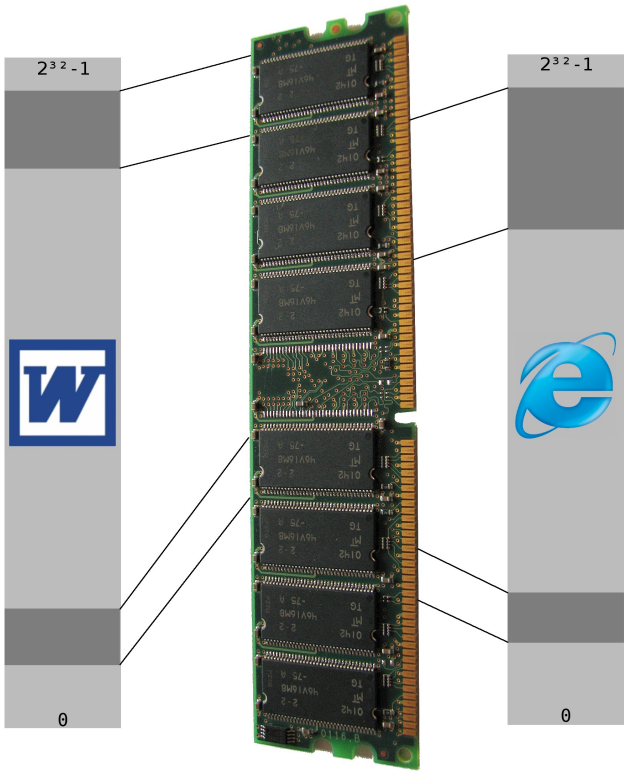


blog.ksplice.com

- Qu'est-ce qu'un pointeur ?
- OS "anciens" : pointeurs vers la mémoire physique
- Différents programmes et le noyau partagent la même mémoire
- Que se passe-t-il si un programme écrit dans la mémoire d'un autre ? Et si la mémoire du noyau est corrompue ?



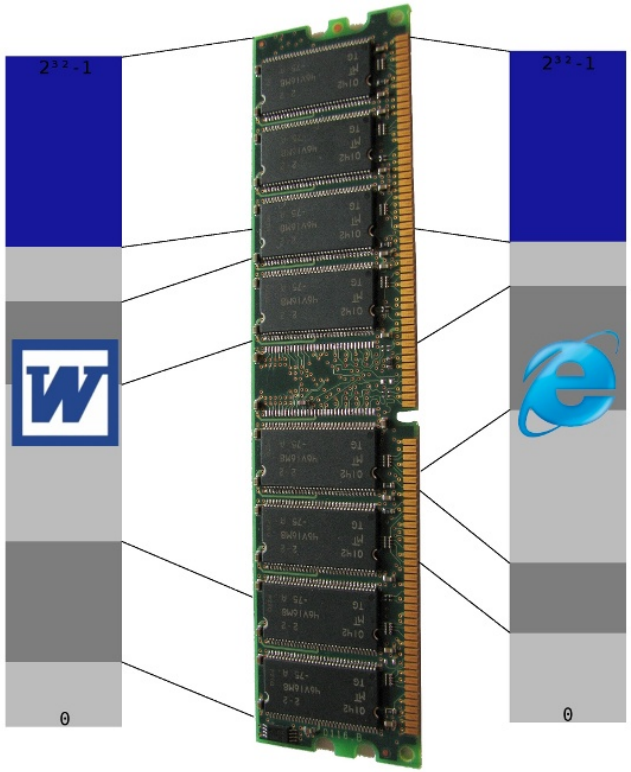
Mémoire virtuelle



blog.ksplice.com

- Chaque programme a son propre espace d'adressage.
- `0x1000` pour IE est différent (hardware) de `0x1000` pour Word.
- La mémoire virtuelle peut utiliser autre chose que de la RAM *(ex : swap sur le disque dur)*.

Mémoire kernel



blog.ksplice.com

- Changer de contexte est couteux.
- Cela arrive souvent (syscalls).
- La mémoire du kernel est *mappée* dans l'espace d'adressage de tous les processus.

Mémoire virtuelle - Double avantage

- On ne s'occupe plus de la mémoire physique au niveau de l'application (abstraction).
- Isolation des processus.

Les différentes architectures

- La taille de la mémoire virtuelle est déterminée par l'architecture :
  - `x86` : $2^{32} - 1 = 4Go$
    - ⋆ Indépendant de la RAM installée
    - ⋆ Si RAM < 4 Go, on utilise le disque dur comme extension
    - ⋆ *Swap*
  - `x86_64` : $2^{48} - 1 = 256To$

Digression sur les architectures

- **x86** : descendante des Intel 8086 16 bits (1978) (CISC)
- **AMD64** : version boostée de x86 (CISC)
- Intel a dû s'aligner sur AMD (**EM64T**)...
- **x86_64** : désigne AMD64 et EM64T
- Ne pas confondre avec **IA-64** (Itanium) d'Intel !

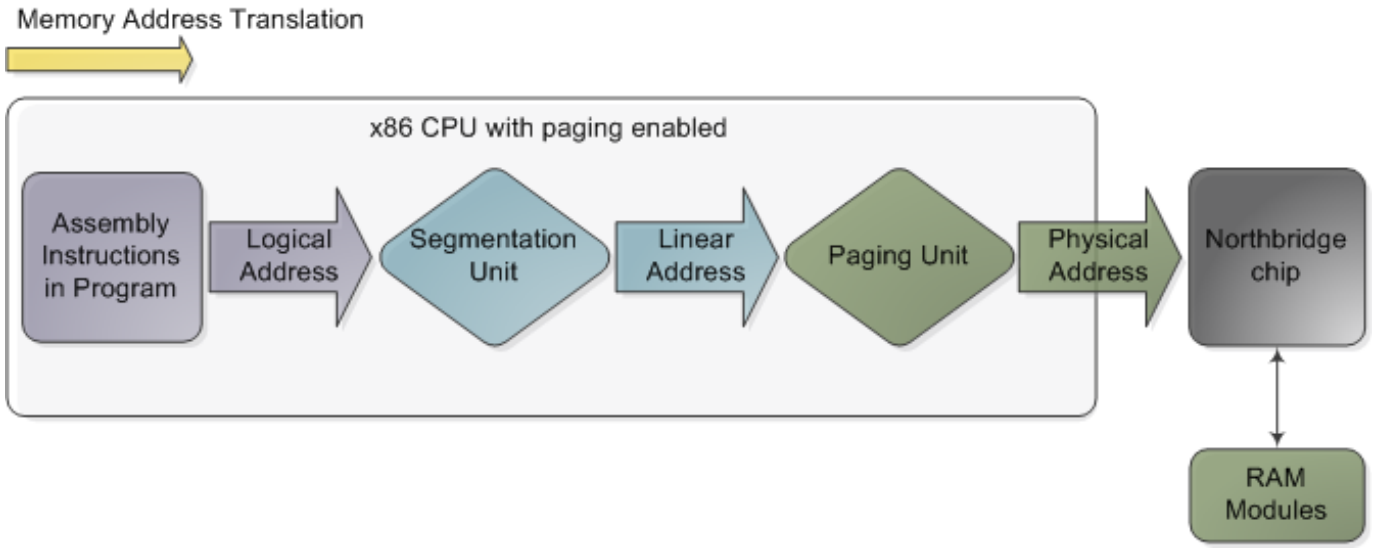
Back on track

Nous nous intéresserons plus spécifiquement à l'architecture **x86**.

Traduction d'adresse - Principe

- Trois types d'adresses :
  - **Logique** : segment + offset relatif
  - **Linéaire (ou virtuelle)** : entier de 32 bit
  - **Physique** : signal électrique sur le bus mémoire

- **Memory Management Unit** : Segmentation Unit + Paging Unit



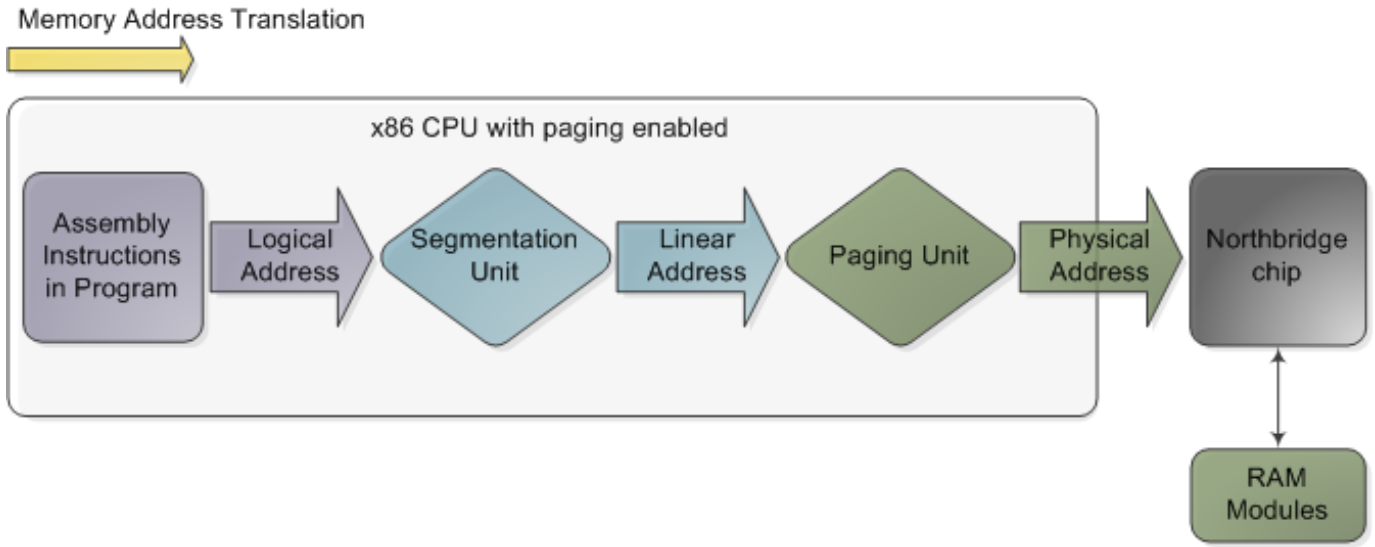
duartes.org/gustavo/blog/

Traduction d'adresse - Segmentation - Hardware

Regardons d'abord le hardware...

Traduction d'adresse - Segmentation - Hardware

- Pourquoi le software n'utilise pas directement des adresses linéaires ?



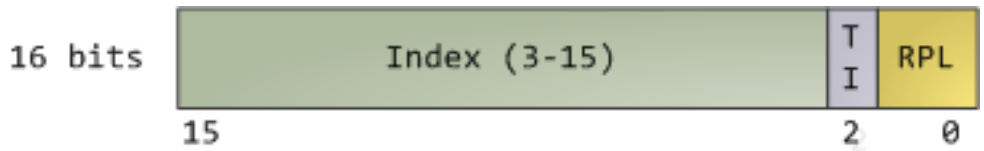
`duartes.org/gustavo/blog/`

- *Intel 8086* : registres de 16 bits
- Comment augmenter la taille de la mémoire adressable (64K) ?
  - Ajout de registres de segment
  - Indique dans quel morceau de 64K on travaille

Traduction d'adresse - Segmentation - Hardware

Registre de segment

Registre de 16 bit contenant un **sélecteur de segment**.

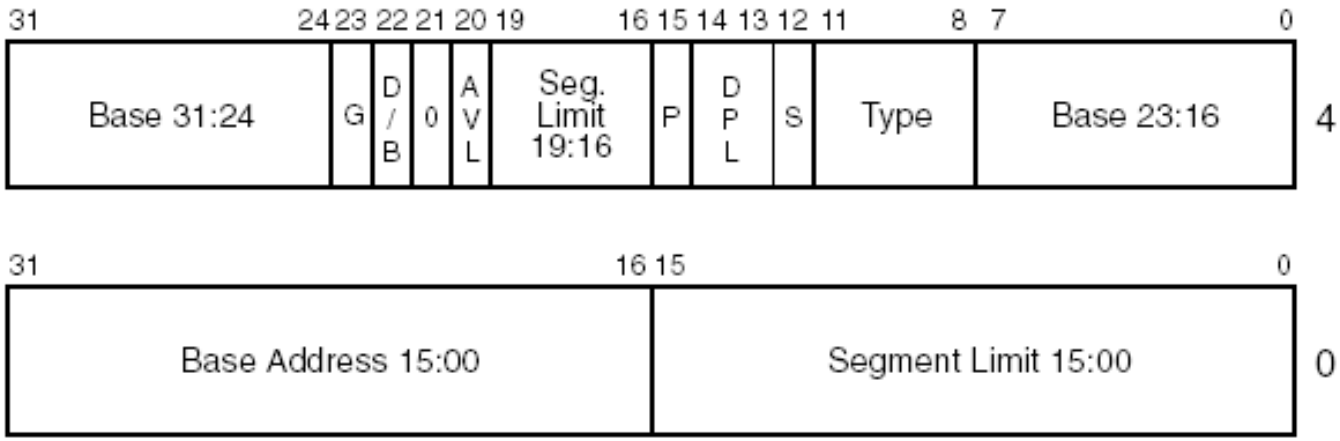


duartes.org/gustavo/blog/

- **cs** : Code
- **ds, es, fs, gs, ss** : Data
- **Requested Privilege Level** : niveau de privilège
- **Index** : position du **descripteur de segment** dans la **Global** / **Local Descriptor Table** (suivant la valeur du **Table Indicator**)
- L'adresse physique de la GDT/LDT est stockée dans le registre **gdtr/ldtr**

Traduction d'adresse - Segmentation - Hardware

- **Descripteur de segment**



AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

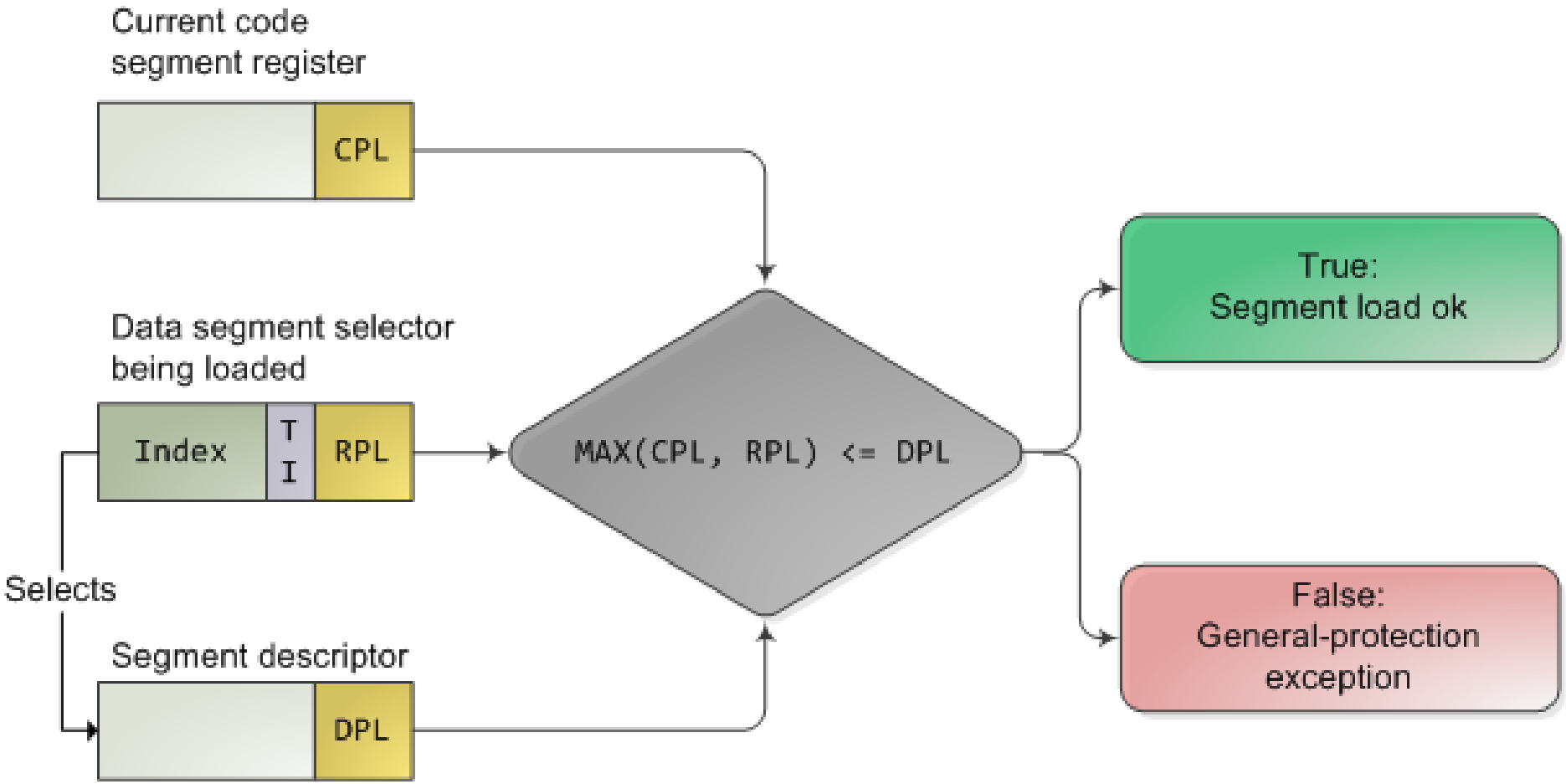
Protection de la mémoire - Segmentation - Hardware

Niveaux de privilège

- **Current Privilege Level** (CPL) : niveau de privilège courant du CPU (*i.e* les 2 premiers bits de `cs`). Cela correspond au segment de code où le CPU lit les instructions.
- **Descriptor Privilege Level** (DPL) : niveau de privilège du descripteur de segment sélectionné. Cela correspond au niveau nécessaire pour accéder au segment.
- **Requested Privilege Level** (RPL) : niveau de privilège présent dans le registre de segment. En pratique, égal au CPL.

Note : le RPL donne la possibilité de charger un segment avec des privilèges amoindris.

Protection de la mémoire - Segmentation - Hardware



duartes.org/gustavo/blog/

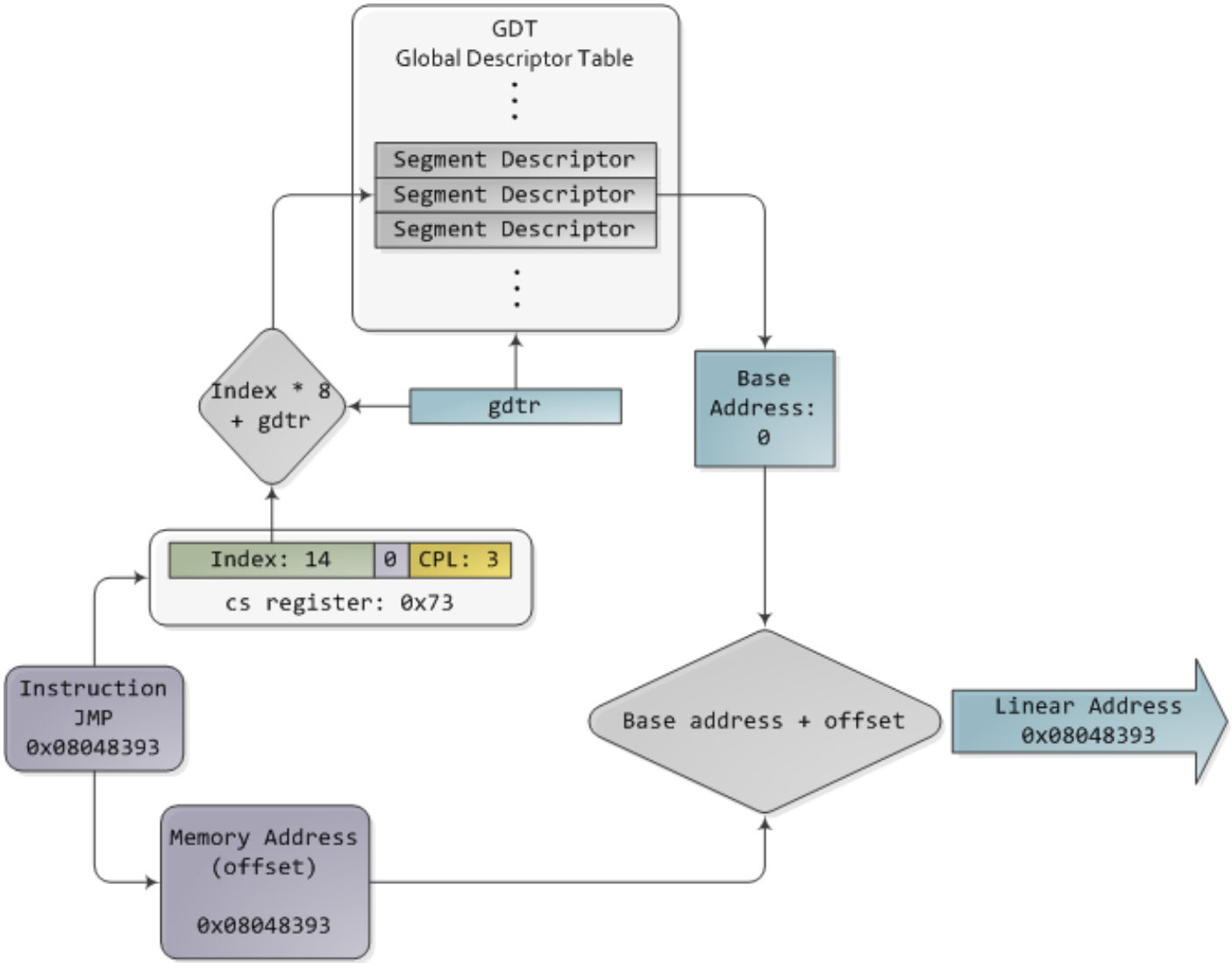
Segmentation - Linux

Et maintenant, en pratique sous Linux. . .

Segmentation - Linux

- **Flat memory model** :
  adresse de base des segments = `0x00000000` , limite = `0xffffffff`.
  - *Il n'y a pas de séparation des addresses kernel-land / user-land au niveau de la segmentation !*
  - Dans le *flat memory model*, la segmentation n'intervient pas lors de la **traduction d'adresse**.
  - **grsec** / **UDEREF** : patch de sécurité qui corrige ce problème.
- Utilisation de seulement 2 niveaux de privilèges : 0 (kernel-land) et 3 (user-land) : `#define USER_RPL 0x3`
- Quatre sélecteurs de segment possibles :
  - `__USER_CS, __USER_DS, __KERNEL_CS, __KERNEL_DS`
  - User : ring 3 // Kernel : ring 0
  - Code : RX // Data : RO

Traduction d'adresse - Segmentation - Linux



duartes.org/gustavo/blog/

Utilisation implicite des registres de segment

Les registres de segments sont utilisés de façon implicite :

- instructions CPU, cibles de `jmp` et `call` : **cs** (code)
- références mémoire : **ds** (data)
- opération sur la stack (`push`, `pop`) : **ss** (stack)
- opération sur les chaines de caractères (`stos`, `movs`) : **es**

On peut surcharger ces valeurs.

Traduction d'adresse - Pagination - Hardware

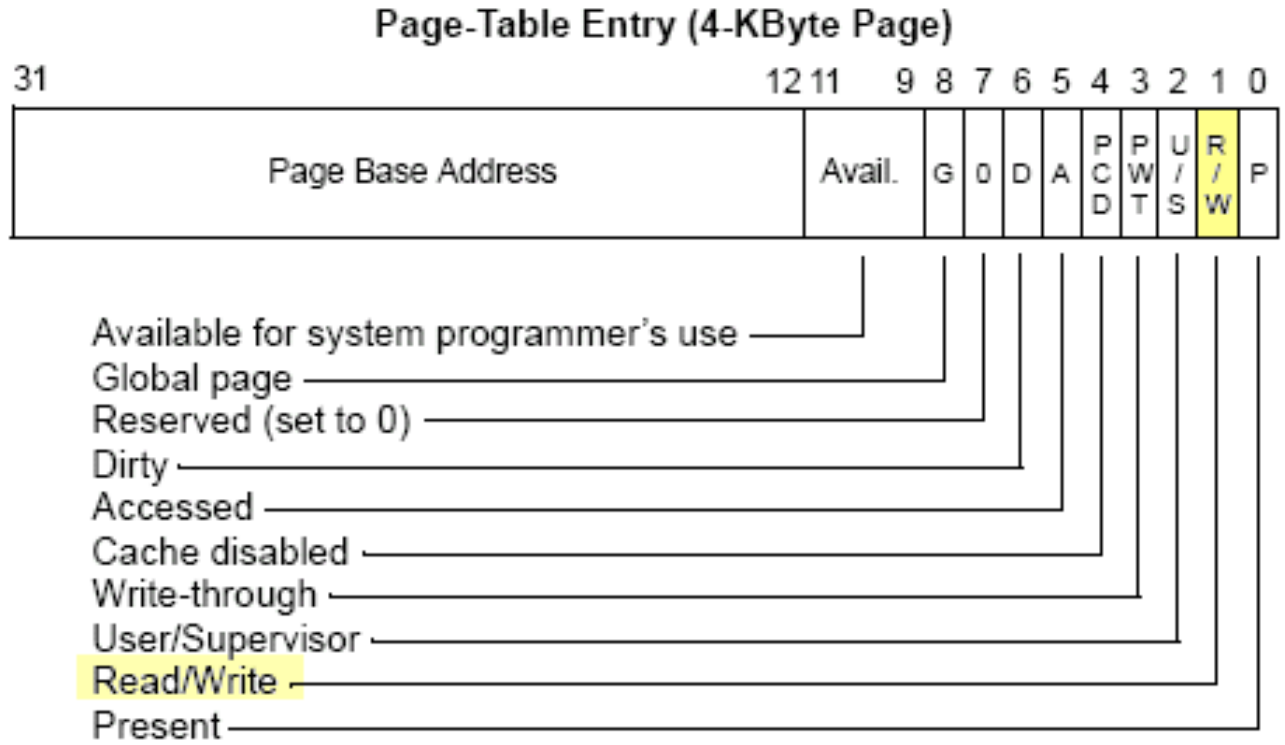
Paging Unit

- Unité hardware qui traduit une adresse linéraire en adresse physique.
- Fonction principale : isoler les processus entre eux.
  - Il s'agit du principal mécanisme de protection mémoire.
- Performance : les adresses linéaires sont groupées en blocs d'adresses physiques contigües (4Ko).

Page Table

- Structure (en mémoire) qui maintient le mapping entre adresses physiques et adresses linéraires.
- Elle contient des **Page Table Entries** (PTE).
- Son adresse physique est stockée dans le registre `cr3`.

Traduction d'adresse - Pagination - Hardware



- **Present** flag : la page est en RAM
- **Read** / **Write** flag : droit RW / RO (pas de eXecute !)
- **User** / **Supervisor** flag : niveau de privilège requis
  - User : peut toujours être adressée
  - Supervisor : peut être adressée uniquement si **CPL < 3**

Traduction d'adresse - Pagination - Hardware

- **Note** :
  - Uniquement 2 niveaux de privilèges (User et Supervisor).
  - Uniquement 2 types d'accès : lecture seule ou lecture / écriture (!)
  - Processeurs récents : **bit NX** (No eXecute) ou **XD** (eXecute Disable).
    - ⋆ On parle de **RO+X** et **RW+NX**.

Changement de ring et segmentation

- Le code user-land ne peut absolument pas agir sur “l’extérieur” :
  - Ouvrir des fichiers, envoyer des paquets réseaux, . . .
- Il faut changer de ring : passer de 3 à 0.
- L’architecture x86 fournit plusieurs méthodes. Voici les plus utilisés :
  - Appel direct : *far jump* et *call*
  - Appel système (utilise l’instruction **sysenter**, **syscall** ou **int 0x80**)
  - Interruption matérielle ou logicielle

Interruptions et exceptions

- **Interruptions** : asynchrones, déclenchées par le hardware
  - Clavier, souris, ...
- **Exceptions** : synchrones, déclenchées par le software
  - Division par 0, page fault, ...
  - Appel direct : instruction `int`
    - `int 0x80` : appel système
    - `int 0x3` : breakpoint
- Lorsqu'on rencontre une interruption/exception, le kernel prend la main.
- **Interrupt Descriptor Table**
  - On associe un nombre à chaque interruption : **vector**.
  - Le CPU utilise ce vecteur comme un index dans l'IDT.
  - Détermine la réponse du kernel associée à une interruption donnée.
  - L'adresse physique de l'IDT est stockée dans le registre **idtr**.

Segmentation et instructions réservées

Certaines instructions ne sont accessibles qu'en kernel-land ($CPL = 0$) :

- `lldt`, `lgdt`, `lidt` : modifier la LDT / GDT / IDT
- manipulation des **control registers** (`cr0`, `...`, `cr4`)
- `vmcall`, `vmlaunch`, `...`

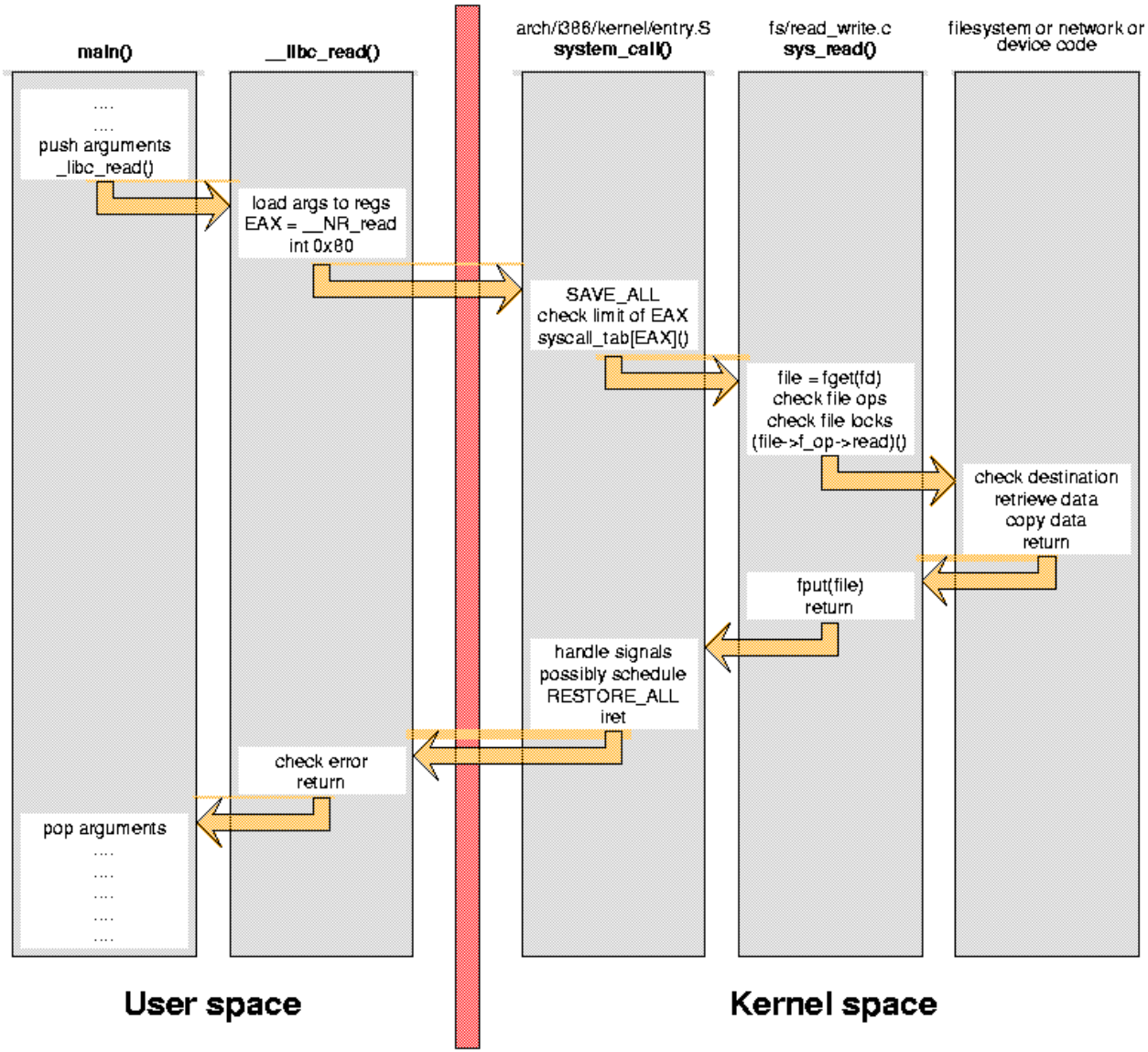
Appel système

Appel système (syscall)

Point d'entrée dans le kernel, permettant à un processus de demander au kernel de faire une action à sa place

- Les appels systèmes changent l'état d'exécution du CPU (usermode ⇒ kernelmode)
- Ils sont identifiés pas un nombre *(syscall number)*, voir `/usr/include/asm/unistd_32.h`
- Ils ont une liste d'arguments, qui permettent de passer des données depuis l'user-land vers le kernel-land
- `http://syscalls.kernelgrok.com/`
- En assembleur :
  - Ancienne méthode : `int 0x80`
  - Nouvelle méthode : `sysenter`
    - `syscall` pour les CPU `x86_64`.

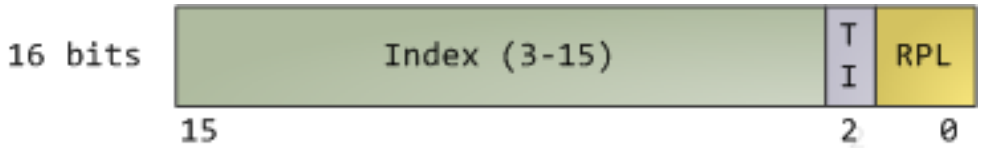
Appel système



www.linux.it/~rubini/

Récapitulatif - La segmentation

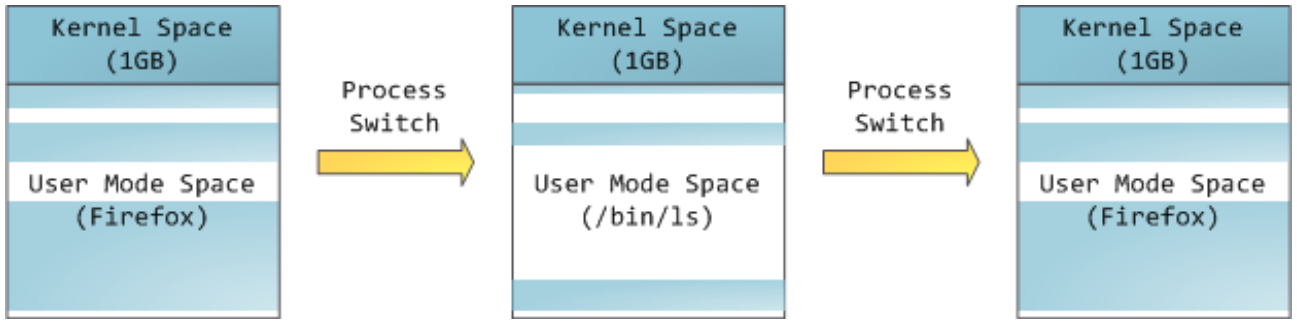
- Mécanisme de traduction d'une **adresse logique** *(seg :off)* en **adresse virtuelle** *(vaddr)*.
- Les **registres de segment** contiennent des **sélecteurs de segments**, ainsi que le **niveau de privilège** du segment associé.
- Les **descripteurs de segments** contiennent l'adresse de base, ainsi que le **niveau de privilège** requis pour accéder au segment (ce niveau DPL est comparé au RPL/CPL pour déterminer si l'accès est autorisé).





Récapitulatif - La pagination

- Mécanisme de traduction d'une **adresse virtuelle** en **adresse physique**.
- **Page** : bloc continu d'adresses physiques (4 Ko).
- Permissions : R/W, U/S, NX.
- Process user-land en ring 3, ne peuvent accéder aux adresses du kernel car celles-ci on le flag **S** fixé dans leur **Page Table Entry**.



duartes.org/gustavo/blog/



Références

- Cours de Sécurité OS, Télécom ParisTech 2010, Ryad Benadjila
- *A Guide To Kernel Exploitation*, Enrico Perla, Massimiliano Oldani
- *IA-32 Intel Architecture*, `http://www.intel.com/products/processor/manuals/`
- *The Linux Programming Interface*, Michael Kerrisk
- `http://blog.ksplice.com/2010/03/null-pointers-part-i/`, Nelson Elhage
- `http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation`, Gustavo Duartes
- `http://grsecurity.net/~spender/uderef.txt`, Brad Spengler
- `http://esec-lab.sogeti.com/post/2011/07/05/Linux-syscall-ABI`
- `http://0xax.gitbooks.io/linux-insides/content/Theory/Paging.html`

Sécurité des utilisateurs - Linux

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Introduction

- Deux grands sujets :
  - **Cloisonnement** des processus et des fichiers opérés par le noyau : contrôle de l'accès des utilisateurs aux ressources.
  - **Authentification** des utilisateurs au système.
- Jusqu'à maintenant : notions communes à Linux et Windows
  - user-land / kernel-land, processus, mémoire virtuelle.
- Linux et Windows gère la sécurité des utilisateurs différemment.
  - Dans cette partie : Linux
  - Dans la partie suivante : Windows

Systèmes multi-utilisateurs

- Plusieurs utilisateurs peuvent utiliser le système *en même temps*.
  - Authentification : de quel utilisateur s'agit-il ?
  - Cloisonner les données : un utilisateur ne doit pas pouvoir lire / modifier les données d'un autre (sauf si on lui donne la permission).
  - Gérer les niveaux de privilèges des utilisateurs.
  - Gérer des groupes d'utilisateurs.

Le super-utilisateur

- Utilisateur possédant le niveau de privilèges le plus élevé
  - Linux : **root**
  - Windows : **Administrateur**
- *Attention* : ne pas confondre avec les privilèges au sens CPU (rings).
  - Dans le noyau (i.e en ring 0) : pas de notion d'utilisateurs.
  - Même espace mémoire, pas de cloisonnement.
  - Processus lancés en user-land par le super-utilisateur restent sous le contrôle du noyau.
- Il peut en revanche faire exécuter facilement du code par le noyau (contrairement aux utilisateurs "normaux").
  - Installation de drivers (`insmod`).

Utilisateurs et groupes

- Sous Linux : notions d'**utilisateurs** et de **groupes**.
- Chaque user a un **uid**, chaque groupe a un **gid**.
- Liste des users et de leur uid : `/etc/passwd`
- Liste des groupes et de leur gid : `/etc/group`

```
lp:x:7:daemon,manu
wheel:x:10:root,manu
network:x:90:manu
audio:x:92:manu,mpd
users:x:100:manu,mpd
vboxusers:x:108:manu
android:x:420:manu
```

`group_name:password:GID:user_list`

Utilisateurs et groupes - /etc/passwd

Format

```
login:password:UID:GID:comment:home:shell
```

Extrait

```
root:x:0:0:root:/root:/bin/bash
http:x:33:33:http:/srv/http:/bin/false
manu:x:1000:1000::/home/manu:/bin/zsh
```

Système de fichiers

- Linux est orienté fichiers : presque toutes les ressources sont accessibles à travers des lectures / écritures dans des fichiers :
  - Périphériques physiques (clés USB, ...) : `/dev`
  - Processus et mémoire virtuelle : `/proc`
  - Mémoire physique : `/dev/mem, /proc/kcore`
  - Mémoire virtuelle du noyau : `/dev/kmem`
  - Configuration du noyau : `/proc/sys/kernel`
  - Logs : `/var/log`
  - Configuration : `/etc`
- Gérer les droits d'accès aux fichiers revient à gérer les droits d'accès des utilisateurs et de leur processus associés aux ressources.

Discretionary Access Control

- Contrôle d'accès de type DAC : les droits accès à une ressource sont fixés par le propriétaire de la ressource.
- Différent du **Mandatory Access Control** (MAC) : une seule autorité décide des droits d'accès.
- DAC avec super-utilisateur : *root* peut modifier les droits de tous les fichiers (même ceux des autres).
- Le système de fichier implémente un **Access Control List** (ACL) limité : les droits *RWX-UGO*.

Les droits RWX-UGO

```
$ ls -l MMU.tex
-rw-rw-r-- 1 manu manu 9020 Apr  5 07:19 MMU.tex
$ ls -ld picts/
drwxrwxr-x 2 manu manu 4096 Mar 24 13:59 picts/
$ ls -l /dev/sda1
brw-rw---- 1 root disk 8, 1 Apr  5 06:56 /dev/sda1
$ ls -l /dev/tty0
crw--w---- 1 manu manu 4, 0 Apr  5 06:56 /dev/tty0
$ ls -l ~/bin/ida
lrwxrwxrwx 1 manu manu 26 Feb 17 13:22 ~/bin/ida -> ~/ida/ida6.0/idaq
```

- **Type** : - = fichier, d = directory, b = block device, c = character device, l = link
- **Droits Utilisateur** : Read, Write, eXecute
- **Droits Groupe**
- **Droits Other**

Les droits RWX-UGO

- On peut aussi exprimer les droits en base octale :
  - `R=4, W=2, X=1`
  - `777 = rwxrwxrwx`
  - `755 = rwxr-xr-x`
  - `600 = rw----`
- *chmod / chown* : changement de droits / propriétaire

```
$ ls -l test
-rw-rw-r-- 1 manu manu 0 Apr  5 08:39 test
$ chmod u+x test
$ ls -l test
-rwxrw-r-- 1 manu manu 0 Apr  5 08:39 test
$ chmod ug-r,o+x test
$ ls -l test
--wx-w-r-x 1 manu manu 0 Apr  5 08:39 test
$ chown http:wheel test
$ ls -l test
--wx-w-r-x 1 http wheel 0 Apr  5 08:39 test
```

Les droits RWX-UGO

- Les droits RWX ne signifient pas la même chose suivant s'il s'agit de fichiers ou répertoire.
- Read :
  - Fichier : **lire** le contenu
  - Répertoire : **lister** le contenu
- Write :
  - Fichier : **écrire** dans le fichier
  - Répertoire : **ajouter** / **supprimer** des fichiers répertoires
- eXecute :
  - Fichier : **exécuter** le fichier
  - Répertoire : **parcourir** le répertoire

Les droits RWX-UGO et processus

- Quand un user *uid* exécute un fichier, le processus a le droit *uid*.
- Certains fichiers doivent pouvoir être lancés par tous les utilisateurs et avoir les droits root.
  - Ex : `passwd` sert à changer de mot de passe, donc modifier `/etc/shadow` et `/etc/passwd`
  - Ce fichier n'est bien entendu pas modifiable par un utilisateur normal.
  - Un utilisateur normal doit pouvoir changer son mot de passe quand même.
- Il faudrait que `passwd` puisse obtenir temporairement les droits root...

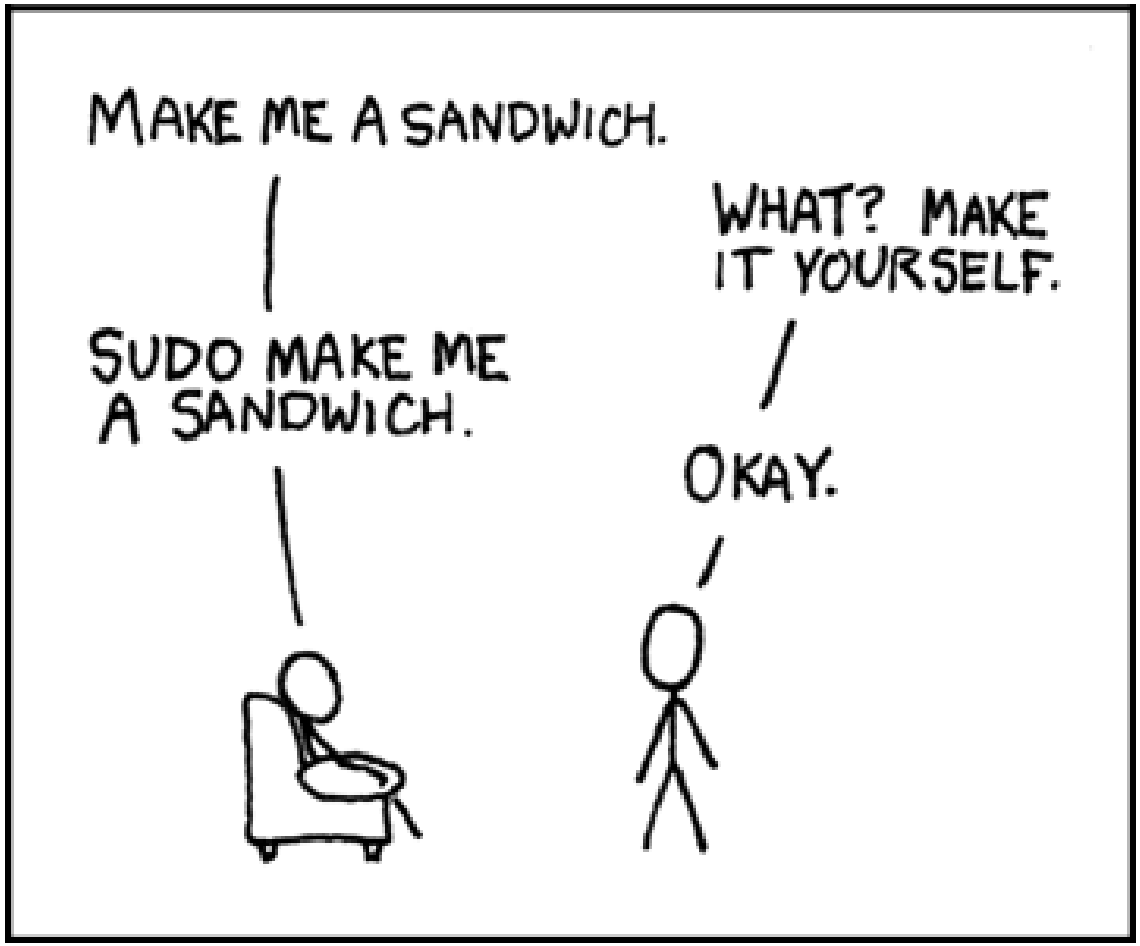
SUID et SGID

- Droits **SUID** et **SGID** : donne le privilège du *propriétaire* du fichier, quel que soit l'exécutant.
- `chmod ug+s`

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 19704 Mar  2 22:22 /usr/bin/passwd
$ ls -l /bin/su
-r-sr-xr-x 1 root root 30274 Feb  4 23:55 /bin/su
```

- Pour un processus, on a donc :
  - *uid* : id réelle de l'utilisateur
  - *euid* : id effective du processus (i.e du propriétaire du fichier dans le cas d'un binaire SUID)
- Même chose avec GID

SUID et SGID - sudo



xkcd.com

Sticky bit

- Le nom vient d'une ancienne fonctionnalité : le programme reste en mémoire après exécution.
- Pour les dossiers : on ne peut effacer que les fichiers dont on est le propriétaire et sur lesquels ont a les droits d'écriture...
  - ...et non plus tous les fichiers +w !
- Utilité : dossier `/tmp`
  - ugo+w (pour que tout le monde puisse écrire des fichiers temporaires)
  - On ne veut pas qu'un programme efface les fichiers des autres

```
$ ls -ld /tmp/
drwxrwxrwt 7 root root 200 Apr  5 18:01 /tmp/
$ cd /tmp/
$ touch test
$ chmod 777 test
$ su dummy
Password:
$ ls -l test
-rwxrwxrwx 1 manu manu 0 Apr  5 18:02 test
$ rm test
rm: cannot remove ‘test’: Operation not permitted
```

Dangers du SUID

- Il faut bien faire attention aux fichiers SUID, notamment aux droits qui leurs sont associés. . .
- Exemple : un programme SUID possède une vulnérabilité qui permet de modifier son exécution
  - L'attaquant peut exécuter du code avec les privilèges du propriétaire. . .
- *Note* : l'exploitation sera vue dans ce cours plus tard.

Principe du moindre privilège

- On ne peut pas se passer du SUID (ex : `passwd`).
- On ne peut garantir qu'un programme ne présente pas de vulnérabilité.
- Quelle est la solution ?

Principe du moindre privilège

- N'accorder à un processus que le privilège minimum pour qu'il fasse son travail.
- Limiter les droits d'accès aux fichiers au maximum.
- Limiter le nombre de programmes SUID.
  - `find /bin /sbin /usr/bin /usr/sbin -perm -4000` (38 sur mon système !)
- Limiter le nombre de daemons privilégiés.

Abandon temporaire de privilèges

code/suid.c

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>

int main(){
    uid_t ruid, euid, suid;
    getresuid(&ruid, &euid, &suid);
    printf("ruid␣:␣%d\teuid␣:␣%d\tsuid␣:␣%d\n", ruid, euid, suid);
    // drop privileges
    seteuid(ruid);
    getresuid(&ruid, &euid, &suid);
    printf("ruid␣:␣%d\teuid␣:␣%d\tsuid␣:␣%d\n", ruid, euid, suid);
    // recuperation privileges
    seteuid(suid);
    getresuid(&ruid, &euid, &suid);
    printf("ruid␣:␣%d\teuid␣:␣%d\tsuid␣:␣%d\n", ruid, euid, suid);
    return 0;
}
```

Abandon temporaire de privilèges

```
$ ls -l mysuid
-rwsrwxr-x 1 root root 4886 Apr 10 19:27 mysuid
$ ./mysuid
ruid : 1000 euid : 0    suid : 0
ruid : 1000 euid : 1000 suid : 0
ruid : 1000 euid : 0    suid : 0
```

Abandon définitif de privilèges

- En cas de faille permettant d'exécuter du code : `setuid(0)`
  - L'attaquant devient root quand même !
- **setuid** : abandon définif (affecte ruid, euid **et** suid)

```
$ ls -l mysuid2
-rwsrwxr-x 1 root root 4973 Apr 10 19:32 mysuid2
$ ./mysuid2
ruid : 1000 euid : 0    suid : 0
ruid : 1000 euid : 1000 suid : 1000
ruid : 1000 euid : 1000 suid : 1000
```

Abandon définitif de privilèges

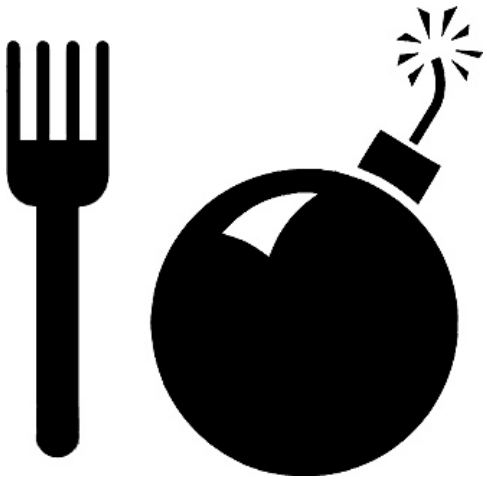
code/suid2.c

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>

int main(){
    uid_t ruid, euid, suid;
    getresuid(&ruid, &euid, &suid);
    printf("ruid : %d\teuid : %d\tsuid : %d\n", ruid, euid, suid);
    // drop definitif privileges
    setuid(ruid);
    getresuid(&ruid, &euid, &suid);
    printf("ruid : %d\teuid : %d\tsuid : %d\n", ruid, euid, suid);
    // verification privileges
    seteuid(suid);
    getresuid(&ruid, &euid, &suid);
    printf("ruid : %d\teuid : %d\tsuid : %d\n", ruid, euid, suid);
    return 0;
}
```

Politique de gestion des ressources

- On ne peut pas toujours faire confiance aux utilisateurs...
  - Fork bomb : `:() { : | :& }; :`
  - Un volontaire ? :-)
- Il faut appliquer une politique de gestion des ressources.



ulimit

- **ulimit** : user limits
- Configuré dans `/etc/security/limits.conf` .

```
$ ulimit -a
-t: cpu time (seconds)          unlimited
-f: file size (blocks)          unlimited
-d: data seg size (kbytes)      unlimited
-s: stack size (kbytes)         8192
-c: core file size (blocks)     unlimited
-m: resident set size (kbytes) unlimited
-u: processes                   27862
-n: file descriptors            1024
-l: locked-in-memory size (kb) 40000
-v: address space (kb)          unlimited
-x: file locks                  unlimited
-i: pending signals             27862
-q: bytes in POSIX msg queues  819200
-e: max nice                    30
-r: max rt priority             65
```

quotas

- **quota** : gestion de l'utilisation maximale du disque.
- `man 2 quotactl`
- Peuvent être définis par utilisateurs ou par groupes.
- Limitations au montage des disques (`/etc/fstab`) :
  - Pas d'exécution : **noexec** (`/tmp` !)
  - Hélas, certaines applications mal codées cesseront de fonctionner...

chroot et jail

- *chroot* : permet de changer la racine “/” d’un processus.
- Permet d’enfermer les utilisateurs :
  - Fichiers visibles, programmes qu’ils peuvent exécuter.
- Inutile si on exploite une faille pour passer root : il est alors trivial de sortir.
- `man 1 chroot; man 2 chroot`
- *jail* : sous BSD. Plus complet que chroot.

Authentification des utilisateurs - Historique

- Autrefois :
  - `/etc/passwd` contient les mots de passe des utilisateurs.
  - `/bin/login` tourne en root sur tous les terminaux.
  - Un utilisateur donne son username et son mot de passe, `login` le vérifie et lance un shell avec les droits correspondants.

Authentification des utilisateurs - Méthode actuelle

- Si les mots de passe sont en clair, on peut les récupérer *offline*
  - Toujours avoir une clé USB bootable sur soi !
- On utilise la cryptographie pour les protéger : c'est le *hash* des mots de passe qui est stocké.
- On rajoute aussi un **salt** : pour éviter les pré-calculs.

Authentification des utilisateurs - /etc/shadow

Format

```
login:hashed password:last changed:min age:max age:
warning period:inactivity period:expiration
```

Exemple

```
dummy:$1$j2ZPpRpR$mRHvqfdFfXQAVILenkgTT.:14875:0:99999:7::
```

Cela ne signifie pas que le mot de passe est sûr...

```
$ sudo john --users=dummy /etc/shadow
Created directory: /root/.john
Loaded 1 password hash (FreeBSD MD5 [32/32])
dummydummy       (dummy)
guesses: 1  time: 0:00:00:00 100.00% (1) (ETA: Tue Apr  5 19:32:24 2011)  c/s: 6
trying: dummydummy
```

Méthodes de hashage

Format

- `$id$salt$hash`
- `dummy:$1$j2ZPpRpR$mRHvqfdFfXQAVILenkgTT.:14875:0:99999:7::`

Algorithmes

- Par défaut : DES
- `$1$` : MD5
- `$2a$` : Blowfish
- `$5$` : SHA-256
- `$6$` : SHA-512

Quelques chiffres

```
$ ./john --test --format=crypt
Benchmarking: generic crypt(3) DES [?/64]... DONE
Many salts:275232 c/s real, 275232 c/s virtual

$ ./john --test --format=md5
Benchmarking: FreeBSD MD5 [SSE2i 12x]... DONE
Raw:30252 c/s real, 30252 c/s virtual

./john --test --format=raw-sha512
Benchmarking: Raw SHA-512 [64/64]... DONE
Raw:1530K c/s real, 1546K c/s virtual

$ ./john --test --format=xsha512
Benchmarking: Mac OS X 10.7+ salted SHA-512 [64/64]... DONE
Many salts:1545 K c/s real, 1561K c/s virtual

$ ./john --test --format=cryptsha512
Benchmarking: crypt SHA-512 rounds=5000 [OpenSSL 64/64]... DONE
Raw:297 c/s real, 297 c /s virtual
```

Authentification des utilisateurs - PAM

- Il faut factoriser les méthodes d'authentification.
  - Sinon tous les programmes qui ont besoin d'authentifier les utilisateurs doivent les re-implémenter...
  - ssh, ftp, telnet, login, ...
- **Pluggable Authentication Module** : brique logicielle centralisant toutes les méthodes d'authentification.
- Configurable dans `/etc/pam.d` , chaque service peut être configuré spéparemment.

```
$ ls /etc/pam.d
chage chfn chgpasswd chpasswd chsh cups groupadd
groupdel groupmems groupmod login newusers other
passwd polkit-1 postgresql samba screen shadow
slim sshd su sudo useradd userdel usermod xscreensaver
```

Authentification des utilisateurs - PAM

- `type control module args`
  - **type** : account (date / heure, ressources utilisées), auth (authentification d'utilisateurs).
  - **control** : que faire si le module échoue (**required** : on passe au module suivant, **requisite** : on quitte).

/etc/pam.d/su

```
#
# The PAM configuration file for the Shadow ‘su’ service
#

# This allows root to su without passwords (normal operation)
auth       sufficient pam_rootok.so

# Uncomment this to force users to be a member of group root
# before they can use ‘su’. You can also add "group=foo"
# to the end of this line if you want to use a group other
# than the default "root" (but this may have side effect of
# denying "root" user, unless she’s a member of "foo" or explicitly
# permitted earlier by e.g. "sufficient pam_rootok.so").
# (Replaces the ‘SU_WHEEL_ONLY’ option from login.defs)
# auth       required   pam_wheel.so

# Uncomment this if you want wheel members to be able to
# su without a password.
# auth       sufficient pam_wheel.so trust
```

Les capabilities

- Problème : l'utilisateur root peut tout faire...
- La gestion des droits n'est pas assez fine.
- Linux 2.2 : introduction des *capabilities* :
  - Division des droits root en plusieurs unités.
  - *CAP_CHOWN* : changer les droits d'accès de fichiers avec `chmod`
  - *CAP_MKNOD* : créer des devices avec `mknod`
  - *CAP_NET_RAW* : utiliser des raw sockets
  - *CAP_SYS_MODULE* : charger des modules (i.e du code kernel)
  - *CAP_SYS_PTRACE* : debugger un autre processus

Capabilities des processus

Chaque processus possède 3 ensembles de capabilities :

- *Permitted*
  - Ensemble maximum des capabilities qu'un processus peut avoir.
  - Lorsqu'on acquiert une capability, c'est cet ensemble qu'on regarde.
  - Abandonner une capability dans cet ensemble est définitif.
- *Effective*
  - Ensemble des capabilities courant.
  - Lors d'une vérification de permission, c'est cet ensemble qu'on regarde.
  - Analogie : real UID et effective UID.
- *Inheritable*
  - Ensemble des capabilities transmettable à l'ensemble *permitted* lors d'un `exec`.

- `grep Cap /proc/PID/status`

Capabilities des fichiers

À chaque fichier on peut associer 3 ensembles similaires de capabilities.

Exemple : CAP_CHOWN

fs/attr.c

```
int inode_change_ok(const struct inode *inode, struct iattr *attr)
{
  // ...

  /* Make sure a caller can chown. */
  if ((ia_valid & ATTR_UID) &&
      (current_fsuid() != inode->i_uid ||
      attr->ia_uid != inode->i_uid) && !capable(CAP_CHOWN))
    goto error;
```

Exemple : CAP_SYS_PTRACE

kernel/ptrace.c

```
int __ptrace_may_access(struct task_struct *task, unsigned int mode)
{
  const struct cred *cred = current_cred(), *tcred;

  tcred = __task_cred(task);
  if ((cred->uid != tcred->euid ||
       cred->uid != tcred->suid ||
       cred->uid != tcred->uid  ||
       cred->gid != tcred->egid ||
       cred->gid != tcred->sgid ||
       cred->gid != tcred->gid) &&
      !capable(CAP_SYS_PTRACE)) {
    rcu_read_unlock();
    return -EPERM;
  }
```

Reférences

- Cours de Sécurité OS, Télécom ParisTech 2010, Ryad Benadjila
- man !

Sécurité des utilisateurs - Windows

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Description générale

- Séparation des privilèges par des droits et des groupes
- L'utilisateur ayant les pleins pouvoirs est le compte Administrateur (équivalent de root sous Unix)
- Windows utilise un DAC avec override du super utilisateur comme sous Unix
- Le DACL (Discretionary Access Control List) contient des ACEs (Access Control Entries)

Composants du système de sécurité

- **Security reference monitor (SRM)** : Composant qui définit une structure de données pour l'accès au token représentant un contexte de sécurité, effectue des vérifications de sécurité sur les objets, manipule les privilèges et génère des messages. (*ntoskrnl.exe*)
- **Local security authority subsystem (lsass)** : Un processus utilisateur s'occupant de la politique de sécurité locale définie dans le registre sous HKLM\SECURITY (accès aux sessions, mots de passe), l'authentification des utilisateurs et de l'envoi des messages au gestionnaire d'évènements. (*lsass.exe*)
- **Security Accounts Manager (SAM)** : Un ensemble de routines permettant d'accéder et de modifier la base de données contenant les noms d'utilisateurs et les groupes contenus dans la base de registre HKLM\SAM. (dans le processus lsass)

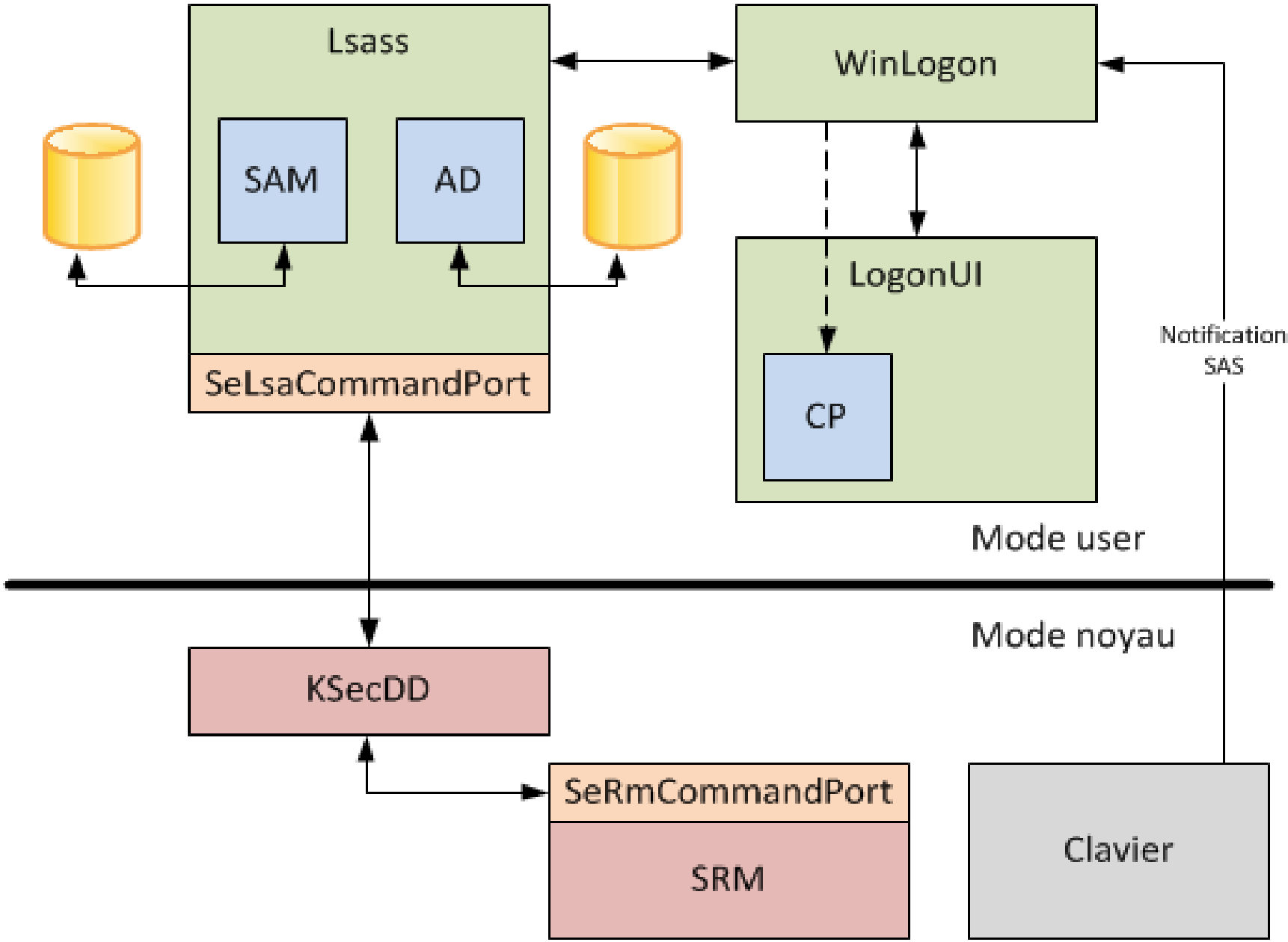
Composants du système de sécurité

- **Active Directory** : Un répertoire de services contenant une base de données qui détient les informations des objets propres à un domaine. Un domaine est un ensemble de machines et des groupes de sécurité associés. Ce répertoire est dupliqué sur l'ensemble des machines qui contrôlent le domaine. Implémenté dans le processus de lsass.
- **Win logon** : Un processus utilisateur qui gère l'authentification des sessions intéractives. Il crée une interface utilisateur à chaque fois qu'un utilisateur se connecte. Il récupére les informations d'identification d'un utilisateur (le login, les certificats sur une carte à puce) via les objets COM Credential Providers.
- **Logon user interface** : Une interface qui permet aux utilisateurs de s'identifier en sollicitant les fournisseurs de credentials. (*logonui.exe*)

Composants du système de sécurité

- **Credential Providers** : Objets COM tournant dans le processus LogonUI fournissant les noms d'utilisateurs, les mots de passe, les codes PIN et autres données biométriques. (*System32\authui.dll*)
- **Network logon service** : Un service windows qui met en place un canal sécurisé vers un contrôleur de domaine via lequel sont envoyé des requêtes sécurisées
- **Kernel Security Device Driver (KSecDD)** : Une librairie noyau implémentant des fonctions de LPC (local procedure call) utilisée lors de la communication avec lsass. (*Ksecdd.sys*)

Intéractions



Exemple d'intéractions

SRM (Noyau)

Crée un port ALPC (*Advanced Local Procedure Call*) au lancement : SeRmCommandPort

Lsass (User)

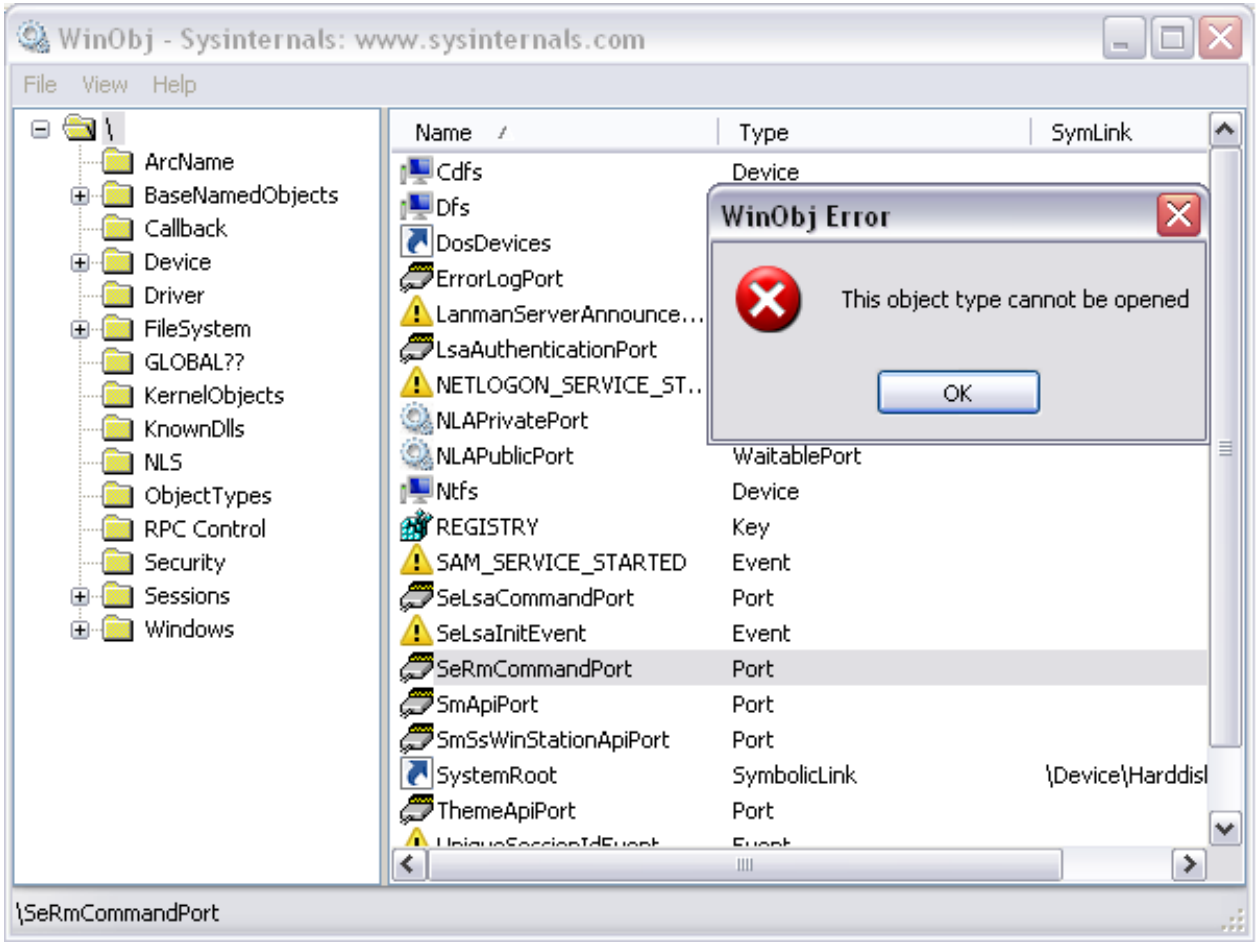
Créé un port ALPC au lancement : SeLsaCommandPort
Le SRM s'y connecte et créé une zone mémoire partagée

Fin d'échange

Les deux ports ne sont plus en écoute

Affichage processus

Démo : ALPC avec winobj.exe



La notion d'objet

- Chaque élément présent dans l'OS est considéré comme un objet
  - Process, Thread
  - Clefs de registre
  - Fichier
  - Ports de communication, mutexes...

Structure d'un objet

- Nom (permet de résoudre un handle)
- Répertoire
- Sécurité : droits d'accès, utilisation et suppression
- Quota
- Compteur de références
- Liste d'objets ayant référencé cet objet
- Type

La notion de handle

- Référence OS fournie à un processus lorsqu'il ouvre un objet (équivalent d'un PID mais pour les objets en général)
- Permet des référencements rapides sans résolution des noms à la volée
- La politique de sécurité Windows repose sur l'obtention/utilisation de handles sur les objets
  - Un thread veut utiliser une certaine ressource, un fichier par exemple
  - Il va demander au noyau via le gestionnaire d'objets un handle sur cet objet par son nom
  - Le noyau vérifie si le thread a les droits nécéssaires, et fournit le handle
- Même vérification à l'utilisation du handle après son ouverture

Protéger les objets

- L'architecture du noyau Windows est basée 3 types d'objets distincts
  - **Executive objects** Des objets servant de primitives à la construction d'objets
  - **Kernel objects** Des objets noyau internes au fonctionnement de celui-ci. Les executives objects en encapsulent
  - **GDI/User objects** Des objets créés par l'utilisateur
- Exemples d'executive objects
  - Process
  - Thread
  - Tokens
  - Fichier

Remarque

Le gestionnaire d'objets joue un rôle clef dans la sécurité des objets car les objets exportés côté utilisateur sont implémentés comme des objets côté noyau

Accès aux objets

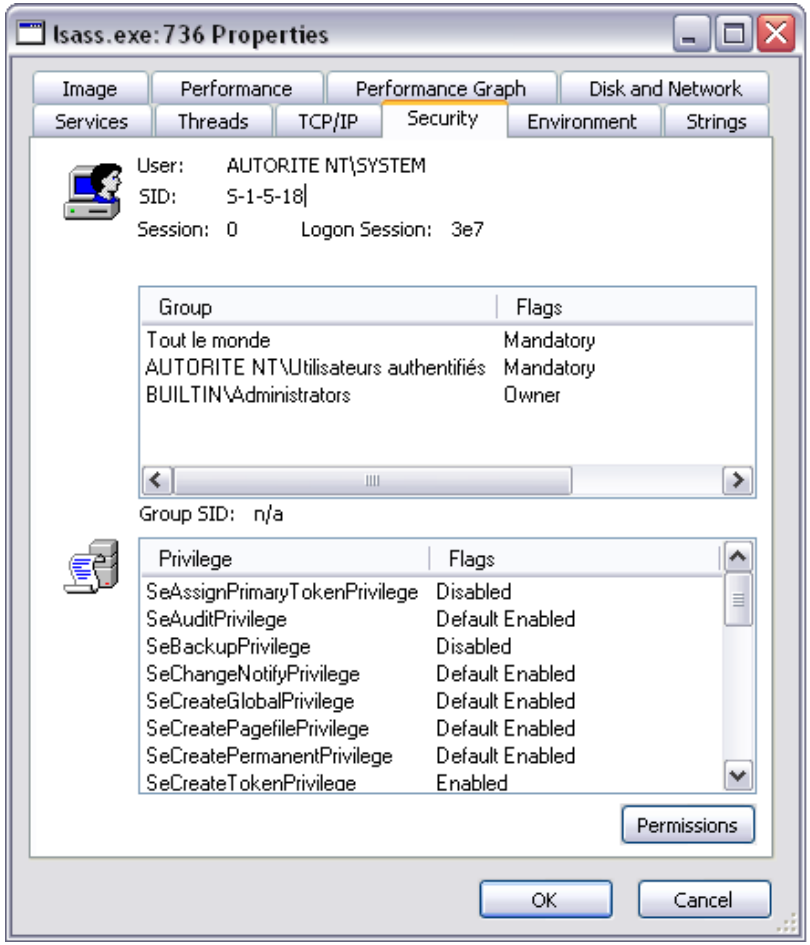
- Lorsqu'un processus veut accéder à un objet, le manager vérifie que l'identifiant de sécurité de l'apellant est bel et bien autorisé à récupérer un handle
- Cependant, les threads peuvent avoir accès à un contexte de sécurité différent via le mécanisme d'impersonation
- Le manager appelle le *SRM* afin de valider l'accès au handle demandé

La notion de SID

- Le SID est l'élément de base d'identification d'une entité : utilisateurs, groupes, machines, domaines
- Les noms ne sont pas forcément uniques !
- Format : SID :S-1-5-21-domain-501
  - Numéro de révision
  - Identifier authority value (48 bits)
  - Une ou plusieurs Subauthorities (32 bits)
  - Un ou plusieurs Relative Identifier (RID)
- Les RID servent à différencier les utilisateurs d'une même machine et d'un même domaine :
  - Utilisateurs, machines et domaine ont le même IA, mais se différencient par leur RID

Exemple SID

Démo : SID avec process explorer.exe



La notion de token

- Chaque processus et thread sous Windows se voit attribuer un objet particuler : le token
- Il est utilisé par le SRM afin d'identifier le contexte de sécurité d'un élément
- Structure d'un token
  - Privilèges (`SeDebug`, `SeLoadDriverPrivilege`, ...)
  - Les SIDs des comptes associés au processus/thread
  - DACL par défaut
  - Un token authentication ID (check modifications)
  - Niveau d'impersonation : token utilisé par les threads pour changer d'identité

DACL

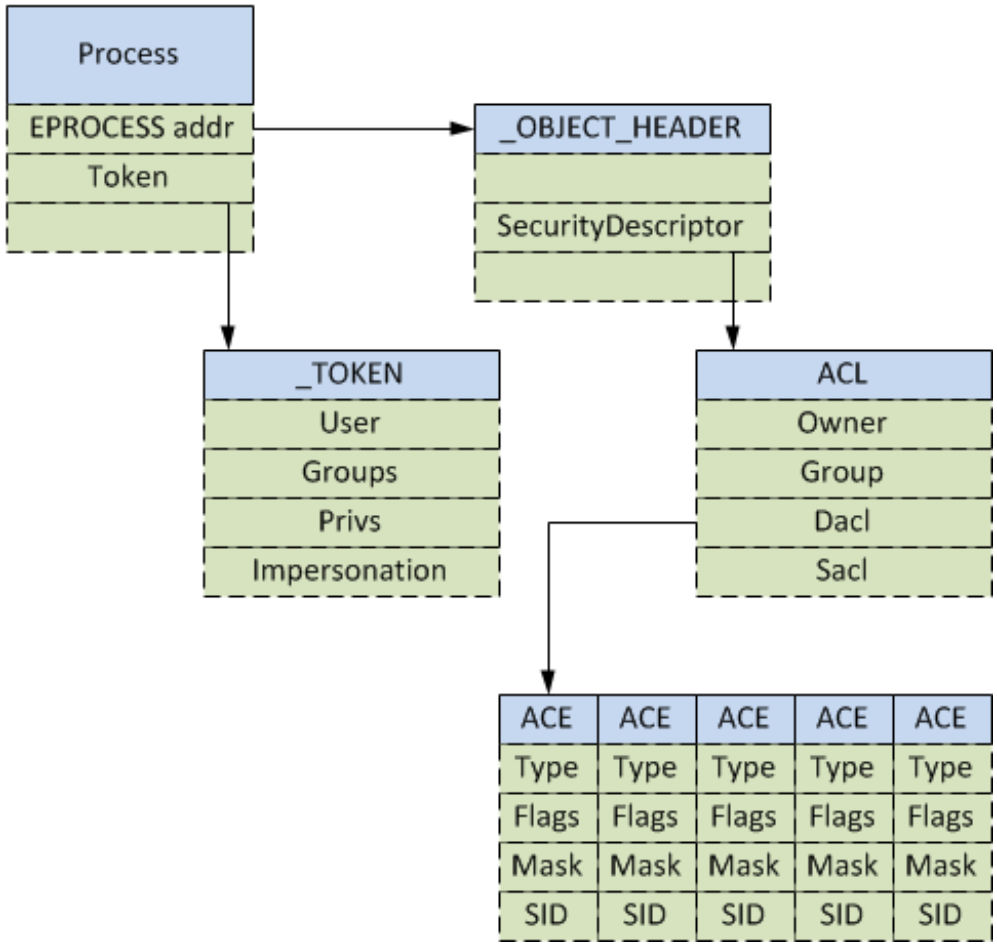
- Le processus de décision du DACL : parcours d'une liste chaînée, notion d'ordre des ACEs

La notion d'impersonation

- Permet à un processus de prendre l'identité SID/RID d'un autre utilisateur
- Ressemble à un `seteuid()` Unix
- Exemple : Service de partage de fichier : Lorsqu'un utilisateur veut supprimer un fichier, le démon fait une impersonation en local sur le serveur pour vérifier les droits
- Eviter de faire les vérification "manuellement"
- API : `RPCImpersonateClient()` en RPC
- L'impersonation, outre le changement de SID/RID, permet de changer d'autres champs du token : enlever les privilèges, marquer des SID, API `CreateRestrictedToken()`, ...

Récapitulatif

Démo : Propriétés de sécurité d'un fichier



Système de fichier et ACL

- Le système de fichiers NTFS utilise une ACL sur les fichiers et répertoires ayant les attributs suivants :
  - Parcours d'un dossier
  - Liste d'un dossier
  - Lecture de méta-données
  - Ajout de fichier
  - Ajout de répertoire
  - Ajout de données à un fichier existant
  - Modification des droits
  - Suppression
  - Lecture
  - Appropriation
  - Exécution

Mots de passe

- Comment sont stoqués les mots de passe ?
- Deux algorithmes possibles : LM Hash et NTLM
  - LM Hash
  - NTLM est utilisé par défaut (avec rétrocompatibilité LM Hash) sous Vista et au delà.

Mots de passe : LM

- Deux chiffrements DES effectués sur deux fois 7 caractères : les deux demi-mdp sont utilisés pour chiffrer une chaîne constante
- Plusieurs problèmes : minuscules transformées en majuscules, DES faible $\Rightarrow$ crack en quelques heures par bruteforce (*john*)
- Aucun “sel” n’est utilisé $\Rightarrow$ les attaques par tables précalculées sont possibles $\Rightarrow$ crack en quelques minutes

Mots de passe : NTLM

- Basé sur le MD4 : *hash_ntlm = MD4(password)*
- Limité à 255 caractères
- Prend en charge l'Unicode

Génération Hash NTLM

```
import hashlib,binascii
hash = hashlib.new('md4', "thisiscool".encode('utf-16le')).digest()
print binascii.hexlify(hash)
```

Exemple

Génération Hash

```
>>> import smbpasswd
>>> passwd = 'thisiscool'
>>> smbpasswd.lmhash(passwd)
'85B9118E40F913A0BB4798EF9CA65D2D'
>>> smbpasswd.nthash(passwd)
'389C1B670DED73B1E36BFEA49708D8D9'
```

- Démonstration LM Hash
- Résultats :

Cassage du hash

```
~ % echo 85B9118E40F913A0BB4798EF9CA65D2D > passwd
~ % john passwd
Loaded 4 password hashes with no different salts (LM DES [128/128 BS SSE2-16])
OOL              (?:2)
guesses: 1  time: 0:00:00:41 0.01% (3)  c/s: 75405K  trying: HFRIDD - HFRGTE
THISISC          (?:1)
```

Mots de passe

- Hash de mots de passe dans la base SAM, sous format protégé
- Cracker le mot de passe suppose avoir accès aux hashs
- Protection de SAM : chiffrement DES (syskey, utilisation du RID de l'utilisateur), mais plusieurs attaques existent :
  - Récupération à chaud via injection de code dans LSASS (*pwdump2*)
  - Récupération offline en récupérant (*samdump2*) et cassant (*bkhives*)

Références

- *Windows Internals Fifth edition*, Mark E. Russinovich and David A. Solomon with Alex Ionescu
- *Cours de Sécurité OS, Télécom ParisTech 2010*, Ryad Benadjila
- *LM/NTLM Documentation - http ://technet.microsoft.com/en-us/library/dd277300.aspx*, Microsoft
- `http://download.sysinternals.com/files/WinObj.zip`
- `http://download.sysinternals.com/files/ProcessExplorer.zip`

Analyse de binaires : concepts et outils

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Les binaires

- Comment sont construits les binaires ?
- Comment “fonctionnent”-ils ?
- Objectif : préparer le terrain pour l’exploitation.
- Deux grands formats : **ELF** et **PE**
  - Dérivés du **Common Object File Format** (COFF)
  - Le format sur disque varie, mais la description mémoire est identique
  - Nous utiliserons ELF comme exemple !

Qu'est-ce qu'un CPU ?

Un CPU est composé de :

- Registres généraux
- Registres spéciaux
- Unités de calcul
- Flags

L'architecture x86 - 32 bits

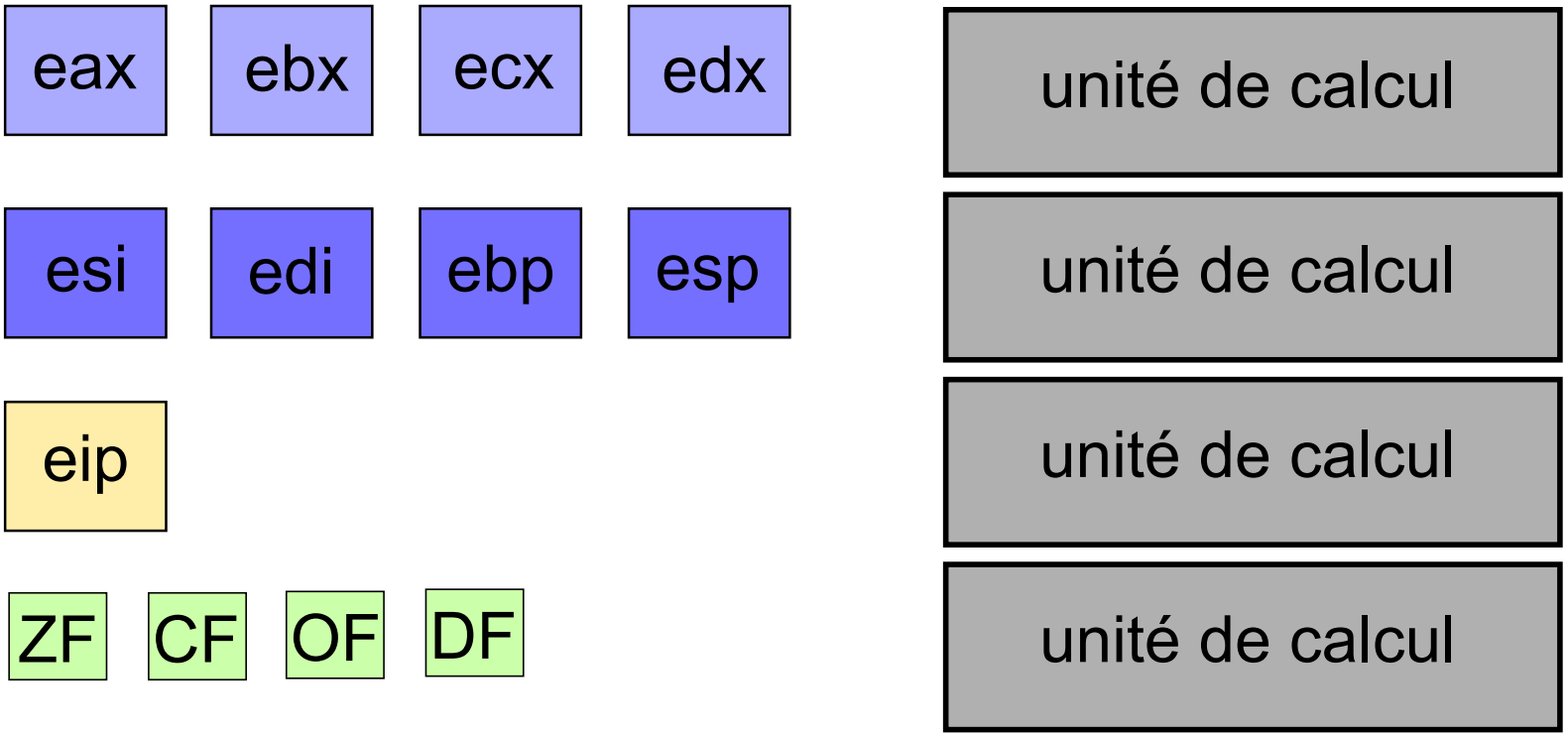
Les registres

- **eax, ebx, ecx, edx, esi, edi** : registres généraux
- **eip** : compteur programme
- **esp, ebp** : indiquent la base et le sommet de la pile
- registres de segment, registres de contrôle, registres de calcul en virgule flottante...

Les Flags

- **ZF** : Zero Flag
- **CF** : Carry Flag
- **OF** : Overflow Flag
- **DF** : Direction Flag

L'architecture x86 - 32 bits

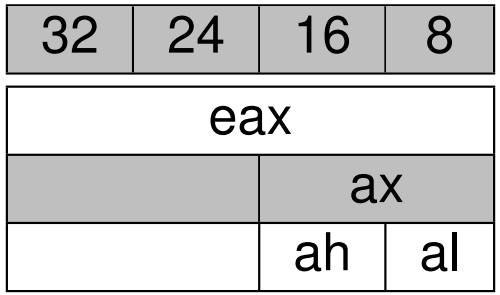


Les registres x86 - 32 bits

- **eax** : accumulateur
  - Registre préféré pour les opérations de base (*add, xor, mul, and, or*).
  - Valeur de retour : `return 1;` $\Rightarrow$ `eax = 1`
- **edx** : données
  - Extension de l'accumulateur pour les calculs (*e.g : pour calcul sur 64 bits, on utilise eax et edx*).
- **ecx** : compteur
  - Il s'agit du *i* dans `for (i=0; i<10; i++)` .
- **esi, edi** : source et destination
  - Toute boucle qui créé / lit des données en mémoire utilise 2 pointeurs qui parcourent ces données : esi et edi.
- **ebp, esp** : bas et haut de la pile
- **ebx** : base

Les registres x86

On peut aussi accéder à des sous-parties d'un registre.



Assembleur x86 - Les syntaxes

Syntaxe AT&T

- Utilisée sous Windows et UNIX.
- **Ordre des opérandes** :
  `movl src, dest`
- **Préfixes** :
  `movl $0x1, %eax`
- **Adresses** :
  `movl (%ebx,%ecx,4), %eax`

Syntaxe Intel

- Dominante dans le monde Windows.
- **Ordre des opérandes** :
  `mov dest, src`
- **Pas de préfixes** :
  l'assembleur se débrouille !
- **Adresses** :
  `mov eax, dword [ebx + ecx*4]`

Assembleur x86 - Exemple

```
main:
    mov eax, 1              ; je mets 1 dans eax
    mov ebx, 3              ; je mets 3 dans ebx
    add eax, ebx            ; eax = eax + ebx
    mov ecx, 12
    cmp eax, ecx            ; je compare eax et ecx
                            ; si eax>ecx, ZF=0
                            ; si eax<=ecx, ZF=1
    jle function2           ; je saute en function1 si ZF=1

function1:
    push 0x42               ; je mets 0x42 sur la pile
    ; ...
    pop eax                 ; je mets la valeur sur le haut de la
                            ; pile dans eax
    ; ...

function2:
    push 0x0                ; je mets 0 sur la pile
    ; ...
    pop eax                 ; ...
    ; ...
```

Endianness

Endianisme ou Boutisme

Ordre des octets en mémoire.

- L'entier **0x08048320** sera représenté par :
  - `08 04 83 20` sur une architecture **big endian** (ou *gros-boutiste*).
  - `20 83 04 08` sur une architecture **little endian** (ou *petit-boutiste*).

L'architecture x86 est little endian.

Langages de programmation et compilation

Familles de langages

- Langages **compilés** : C, C++, Pascal, Fortran, ...
- Langages **interprétés** : Python, Perl, Ruby, ...
- Langages **semi-interprétés** : Java, VB, C#, ...

***Note** : on parle ici de famille par rapport à la compilation et pas par rapport au paradigme (comme les langages orientés objet, impératifs, déclaratifs, ...)*

Les différentes familles

Langages compilés

- Un compilateur prend en entrée un fichier texte et produit un fichier binaire (code source vers langage machine).

Langages interprétés

- Le code source est traduit en code machine par un programme spécifique (l'interpréteur).
- Ce programme doit être installé sur chaque machine destinée à exécuter le programme.

Langages semi-interprétés

- Un compilateur traduit le code source en langage assembleur spécifique (le bytecode), indépendant du processeur.
- Ce bytecode est exécuté par un autre programme..

Les différentes familles

Langages compilés

- Le binaire produit est spécifique au CPU et à l'OS.
- La meilleure solution si on cherche la performance.

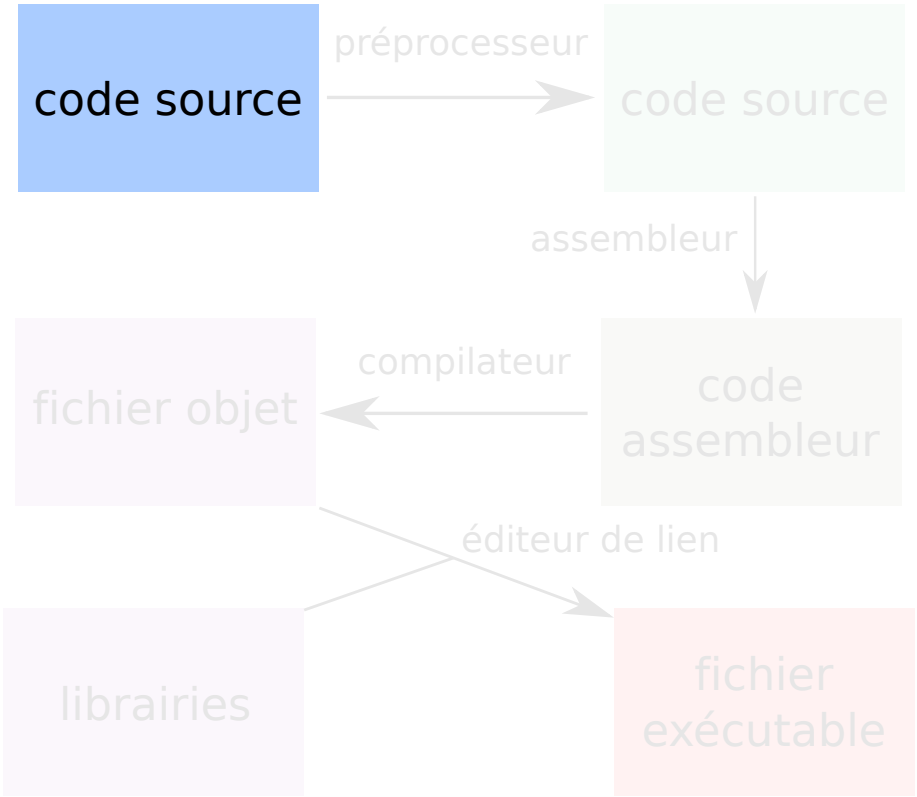
Langages interprétés

- L'interpréteur est spécifique au CPU et à l'OS.
- Le code source est indépendant de la plateforme.
- *"Write once, run everywhere."*

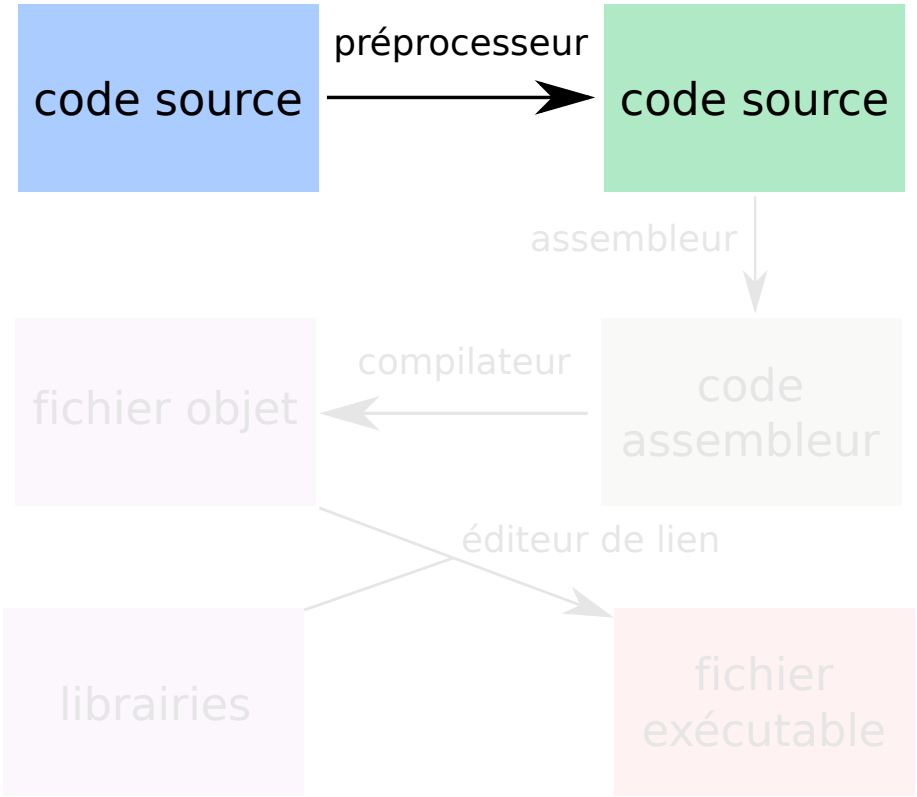
Langages semi-interprétés

- La machine-virtuelle est spécifique au CPU et à l'OS.
- Le compilateur et le code source sont indépendants de la plateforme (i.e ils produisent le même bytecode).

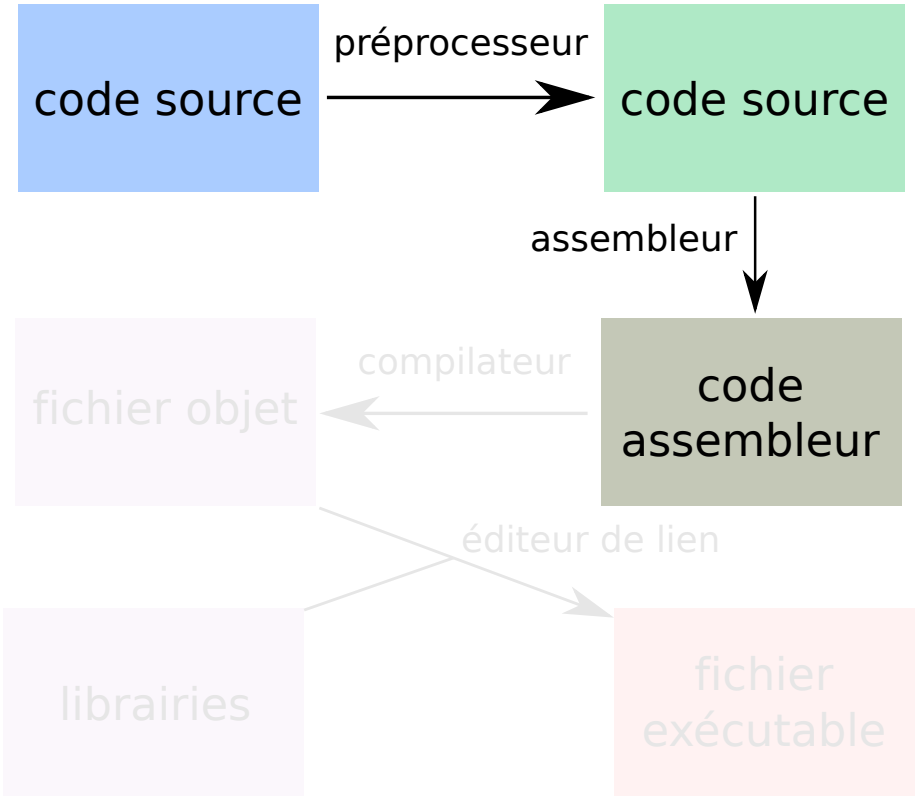
Les étapes de la compilation



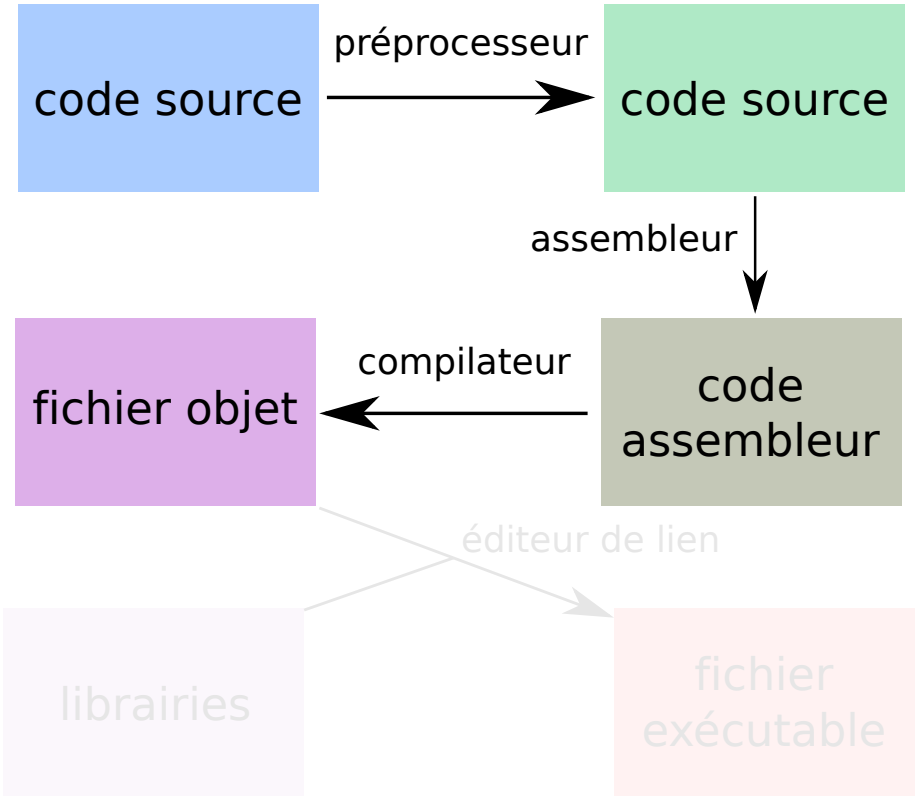
Les étapes de la compilation



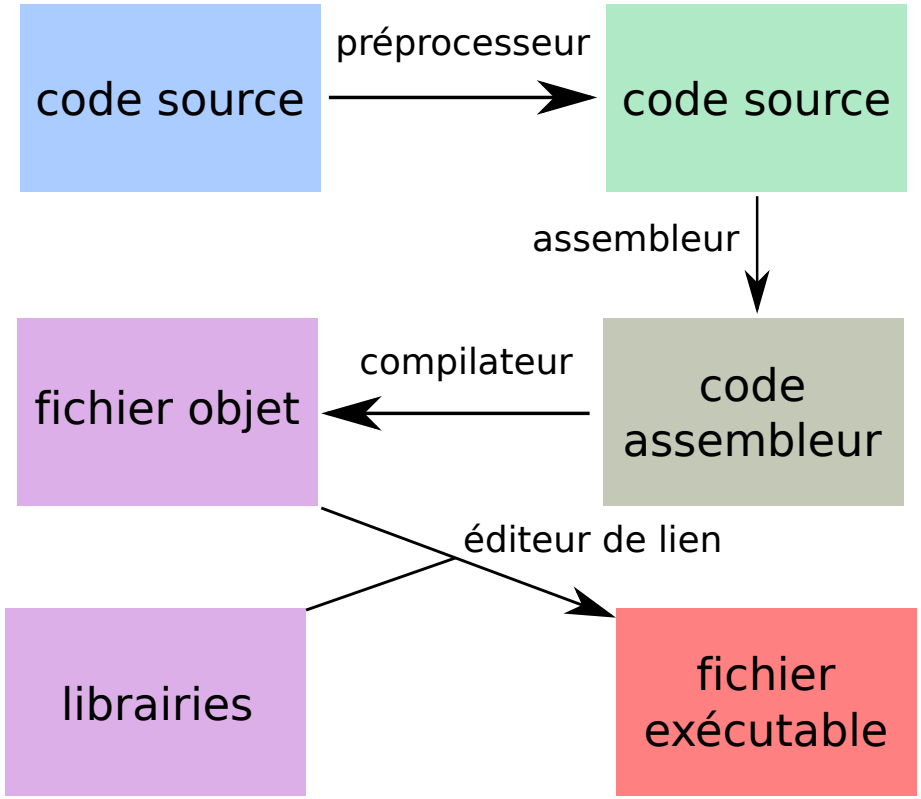
Les étapes de la compilation



Les étapes de la compilation



Les étapes de la compilation



Les fichiers exécutables

Un fichier exécutable est composé de :

- **en-tête** *(header)* : contient les informations sur le fichier
  - point d'entrée *(entrypoint)*
  - imports : fonctions extérieures dont le programme a besoin
  - ...
- **code** : instructions que le processeur va exécuter
- **données** :
  - initialisées (strings à afficher, URLs, ...) et non-initialisées (valeur que l'utilisateur va devoir rentrer, date à laquelle le programme est lancé)
  - globale au programme ou locale (à l'intérieur d'une fonction)

Les étapes de la compilation

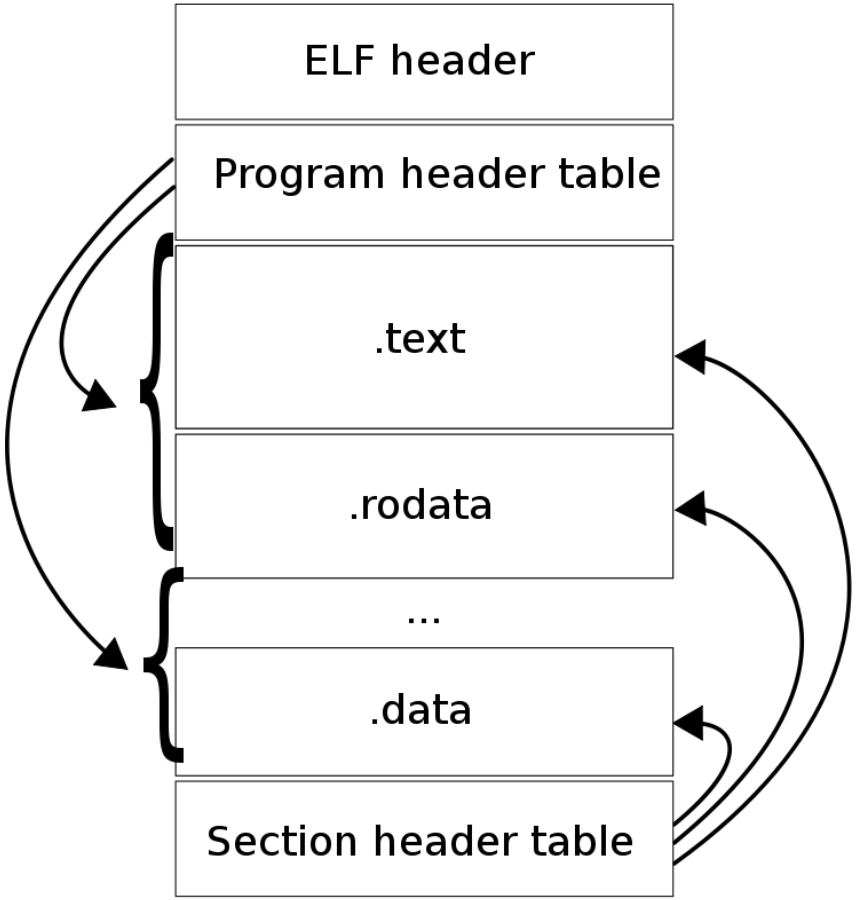
Definition (Executable and Linkable Format (ELF))

Format de fichier pour les **exécutables**, les **librairies** et les **fichiers objets**, utilisé notamment sous Linux.

Petite démonstration, utilisant quelques outils :

- **file** : détermine le type d'un fichier
- **gcc** : GNU Compiler Collection
- **objdump** : manipule les fichiers objets et exécutables
- **hexedit** : éditeur hexadécimal

Le format ELF - sur disque



Le format ELF - sur disque

ELF Header

Donne les **informations** sur le binaire (type, position de la SHT / PHT).

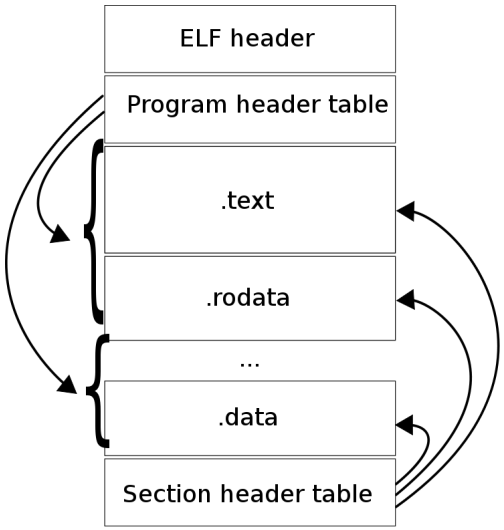
Section Header Table

Décrit les **sections** du binaire :

- *.text* : code exécutable
- *.data, .rodata* : données initialisées
- *.bss* : données non-initialisées

Program Header Table

Décrit l'agencement du programme en mémoire (**segments**).

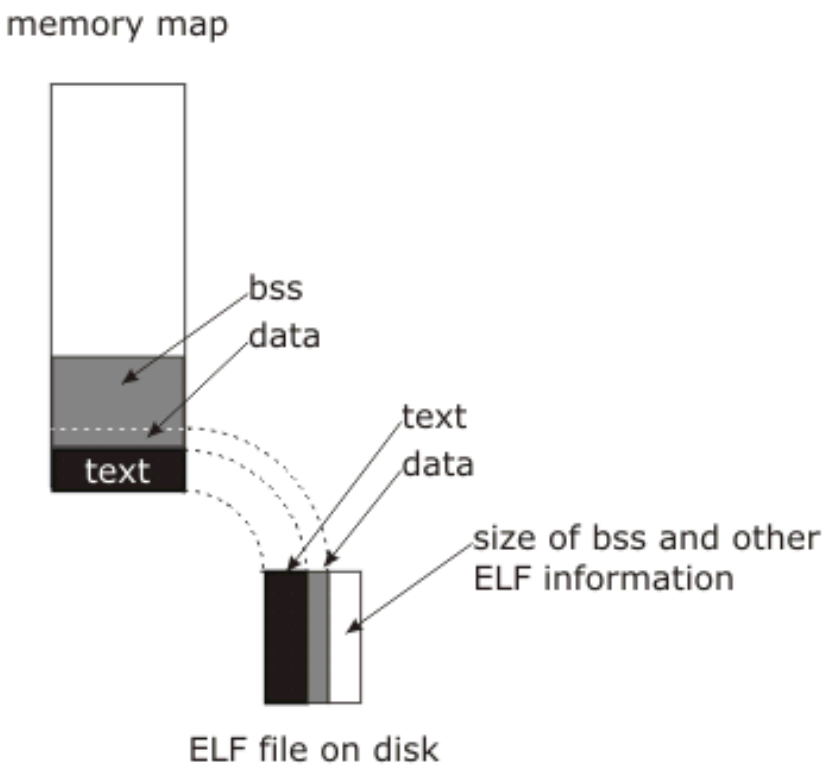


Le format ELF - sur disque

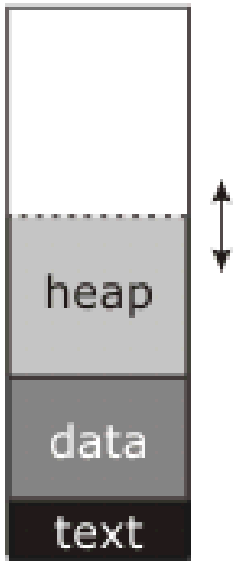
Nouvelle démonstration, nouveaux outils :

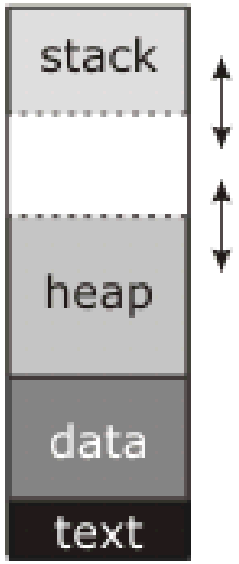
- **strings** : recherche les chaînes de caractères dans un fichier
- **readelf** : donne des informations sur les fichiers ELF

Du disque vers la mémoire

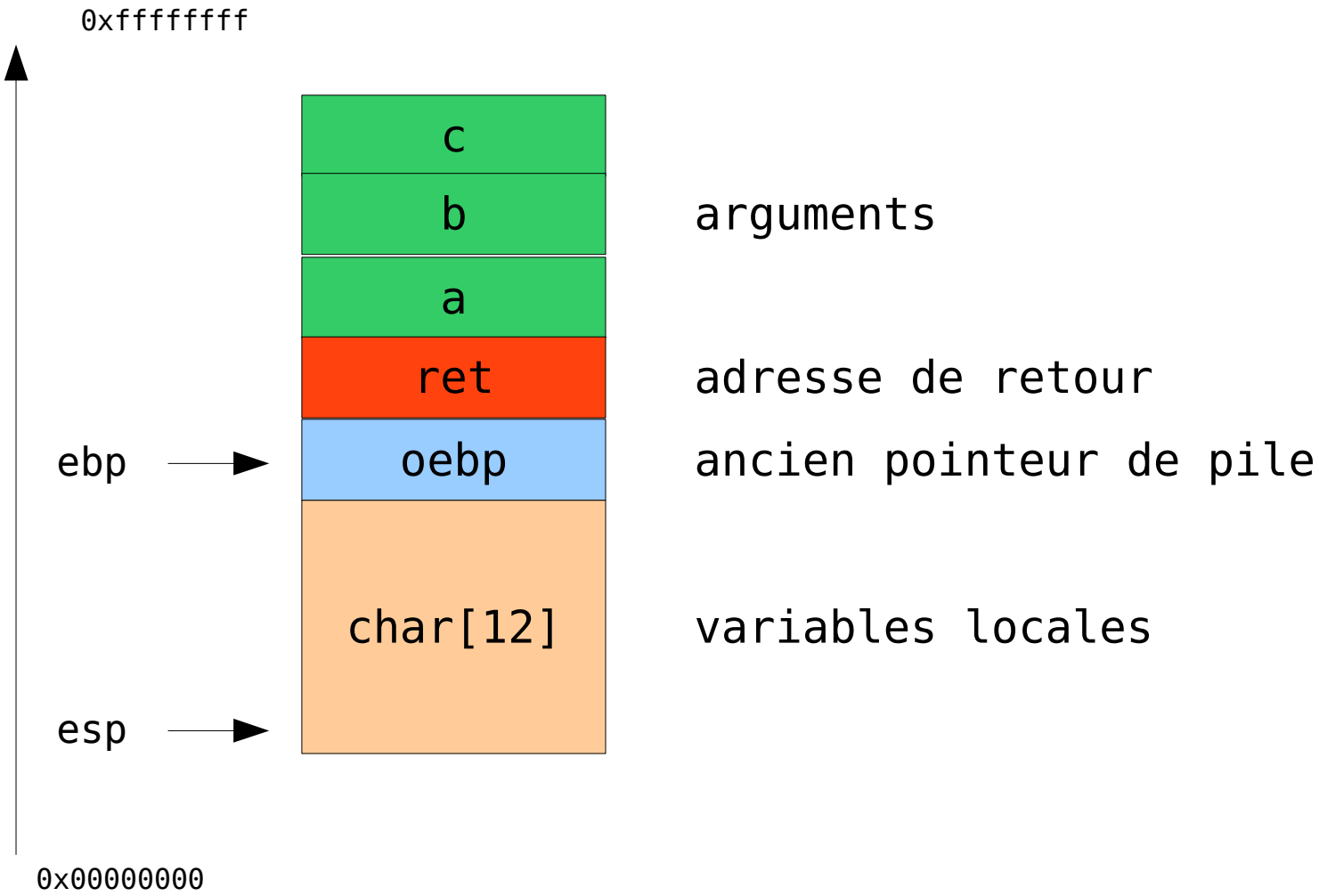


`www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html`





Stackframe


0xffffffff
c
b
a
ret
oebp
char[12]
arguments
adresse de retour
ancien pointeur de pile
variables locales
ebp
esp
0x00000000

Stackframe, appel de fonctions, prologue

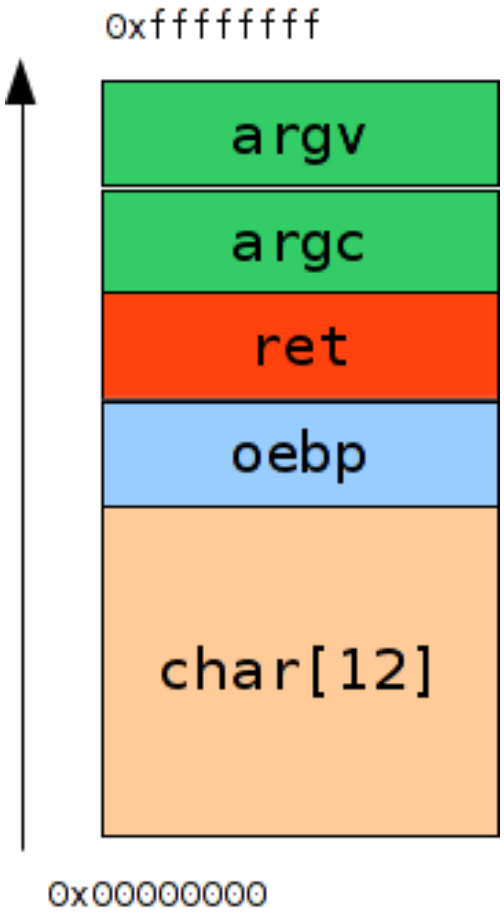
Appel

La fonction appelante pousse les **arguments** de la fonction appelée sur la **pile**. Lors de l'appel, elle sauvegarde **l'adresse de retour** sur le haut de la pile, puis transfère le contrôle à la fonction appelée.

Prologue

La fonction appelée sauvegarde le pointeur de pile (`ebp`) précédent et crée sa nouvelle **stackframe**. Elle alloue de l'espace sur la pile pour ses **variables locales**.

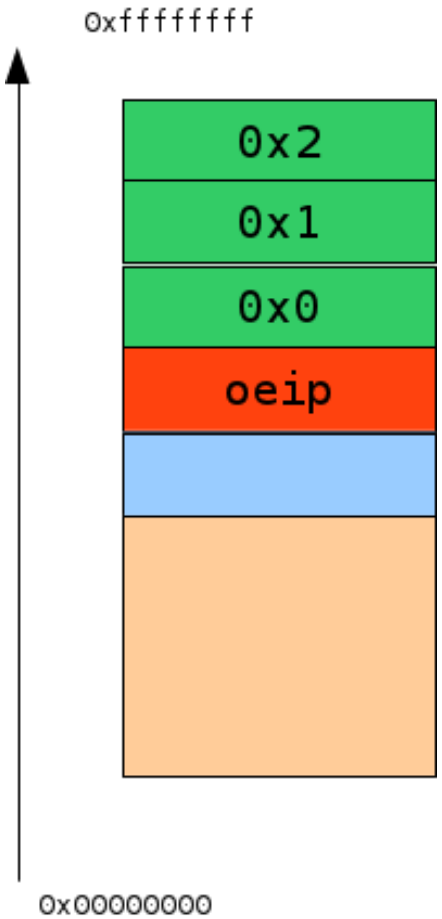
Stackframe - Fonction main




```
#include <string.h>

int main(int argc,
         char **argv)
{
    char mystr[12];
    strcpy(mystr,
           argv[1]);
    return 0;
}
```

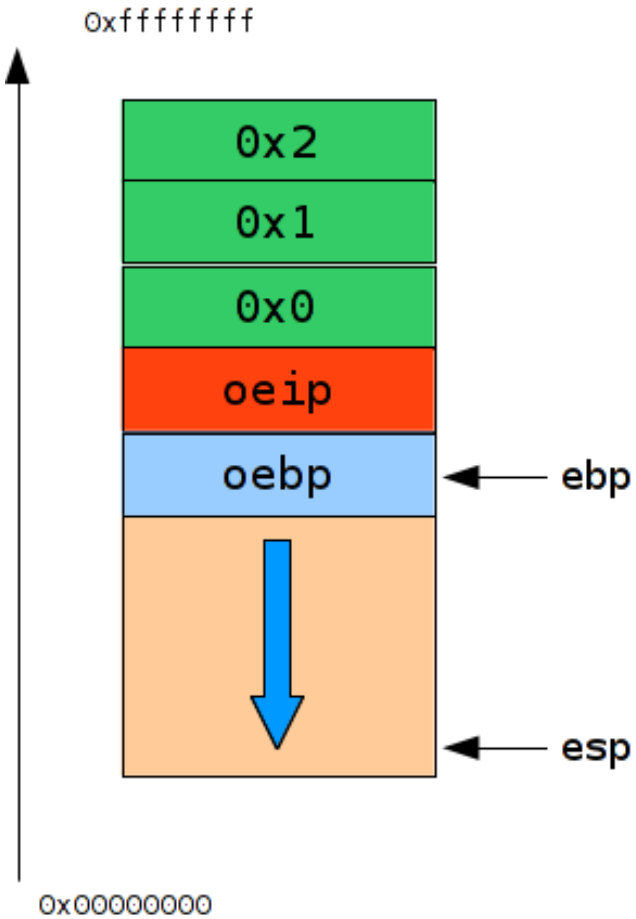
Appel de fonction - Assembleur




```
    push 0x2
    push 0x1
    push 0x0
    call fun1 ; push eip
              ; set eip = fun1

fun1:
    ; do stuff
```

Prologue - Assembleur




```
; prologue
    push    ebp
    mov     ebp, esp
    sub     esp, 0xc
```

Stackframe, épilogue, retour de fonctions

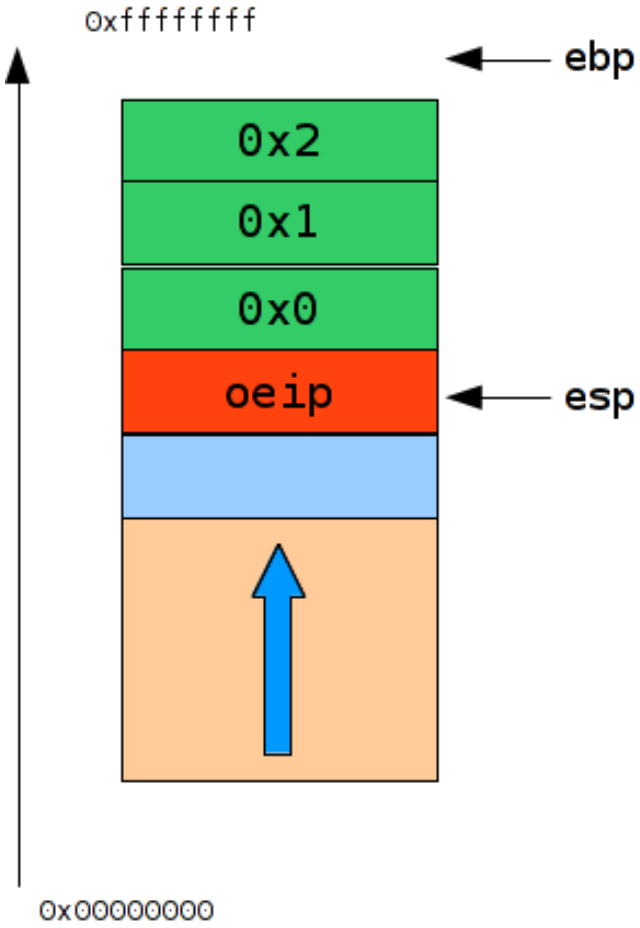
Epilogue

La fonction appelée libère l'espace alloué pour ses variables locales. Elle restore l'ancienne stackframe.

Retour

La fonction appelée transfère le contrôle à l'adresse sur le **haut de la pile** (*i.e* l'adresse en `[esp]`). C'est l'**adresse de retour**.

Epilogue et retour - Assembleur




```
; epilogue
    add     esp, 0xc
    mov     esp, ebp
    pop     ebp
; retour
    ret       ; "pop eip"
```

Appel et retour de fonctions

Quelle est la valeur retournée par ce programme (dans eax) ?

```
BITS 32
GLOBAL _start
EXTERN _exit
SECTION .text

_start:
  xor eax, eax      ; eax = 1
  call fun1         ; appel fun1
  xor eax, eax      ; eax = 0
  push eax          ; eax sur la pile
  call _exit        ; exit(valeur sur la pile)

fun1:
  mov eax, 0x1      ; eax = 1
  ret
```

Appel et retour de fonctions

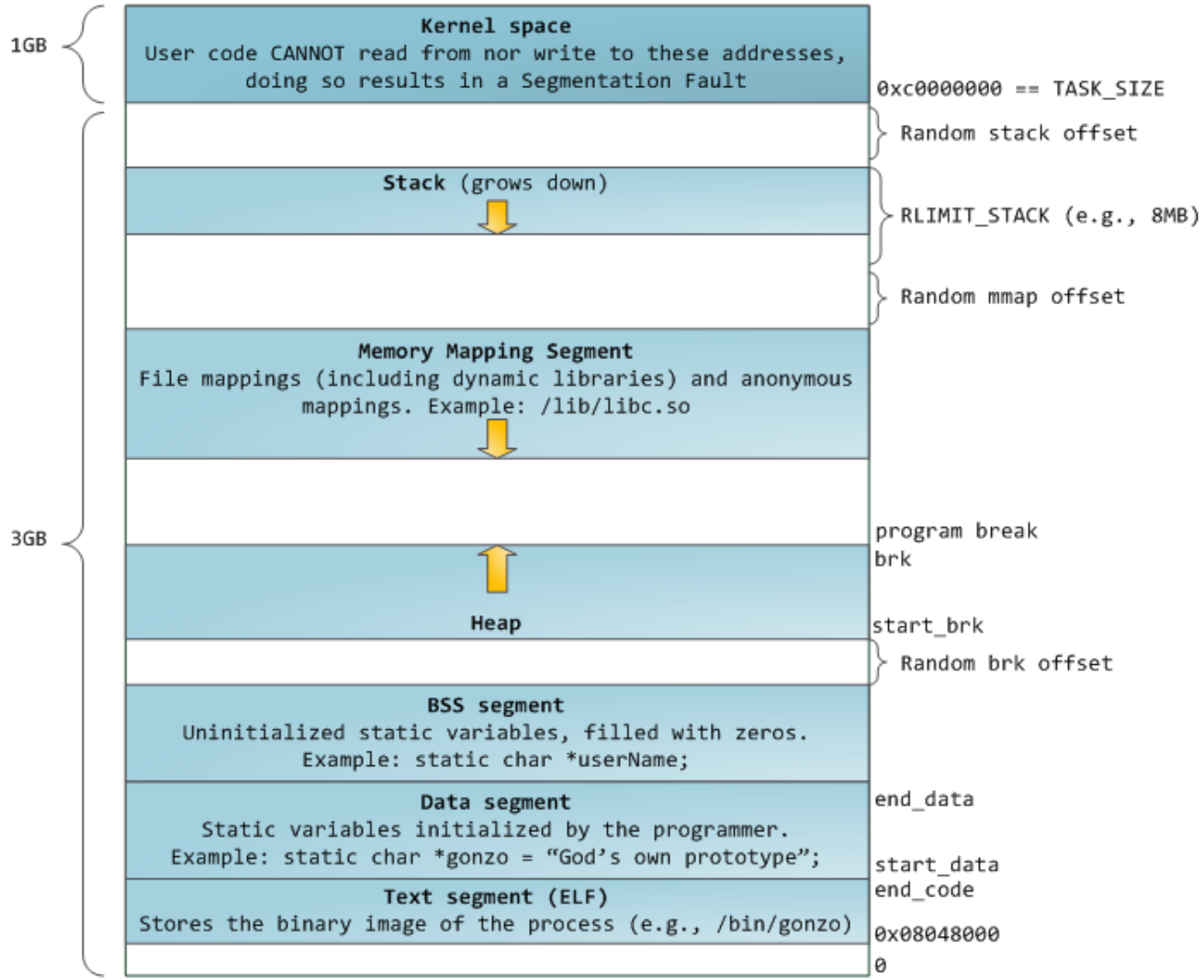
Quelle est la valeur retournée par ce programme (dans eax) ?

```
BITS 32
GLOBAL _start
EXTERN _exit
SECTION .text

_start:
  xor eax, eax          ; eax = 0
  call fun1             ; "push eip"
                        ; set eip=fun1
  xor eax, eax          ; eax = 0
                        ; "31 c0" en hexa
  push eax              ; valeur sur la pile = eax
  call _exit            ; exit avec code retour
                        ;   = valeur sur la stack

fun1:
  mov eax, 0x1          ; eax = 1
  inc DWORD [esp]       ; increment valeur sur la pile
  inc DWORD [esp]       ; ==> [esp] += 2
  ret                   ; retour a l'adresse sur la pile
```

Programme en mémoire - 32 bits



duartes.org/gustavo/blog

Programme en mémoire - 32 bits

```
bf81d000-bf83e000 rw-p 00000000 00:00 0          [stack]
b7833000-b7834000 rw-p 0001c000 08:01 109784     /lib/ld-2.13.so
b7832000-b7833000 r--p 0001b000 08:01 109784     /lib/ld-2.13.so
b7816000-b7832000 r-xp 00000000 08:01 109784     /lib/ld-2.13.so
b7815000-b7816000 r-xp 00000000 00:00 0          [vdso]
b7813000-b7815000 rw-p 00000000 00:00 0
b77e9000-b77ec000 rw-p 00000000 00:00 0
b77e8000-b77e9000 rw-p 00147000 08:01 106097     /lib/libc-2.13.so
b77e6000-b77e8000 r--p 00145000 08:01 106097     /lib/libc-2.13.so
b77e5000-b77e6000 ---p 00145000 08:01 106097     /lib/libc-2.13.so
b76a0000-b77e5000 r-xp 00000000 08:01 106097     /lib/libc-2.13.so
b769f000-b76a0000 rw-p 00000000 00:00 0
b74ec000-b769f000 r--p 00000000 08:01 675954     /usr/lib/locale/locale-
    archive
08def000-08e10000 rw-p 00000000 00:00 0          [heap]
0804e000-08050000 rw-p 00000000 00:00 0
0804d000-0804e000 rw-p 00005000 08:01 303921     /usr/bin/tac
08048000-0804d000 r-xp 00000000 08:01 303921     /usr/bin/tac
```

Test d'attention

Question

Pourquoi quatres allocations, dont une sans permission ?

Réponse

Allocation PROT_NONE
N'utilise pas de mémoire virtuelle
Empêcher l'allocation entre les autres pages ?
Diminution de la fragmentation du tas
Allocation plus rapide
`http://www.greenend.org.uk/rjk/tech/dataseg.html`
`malloc/arena.c` + If consecutive mmap

Digression - L'architecture x86_64

- Deux modes d'exécution principaux :
  - Legacy : identique au x86, les extensions 64 bits sont désactivées
  - Long mode : mode natif. Dans ce mode, les applications 32 bits peuvent quand même s'exécuter non-modifiées (*compatibility mode*).
    - ⋆ Par exemple : jusqu'à Snow Leopard, applications 64 bits mais kernel 32 bits.
- Augmentation de la taille des registres généraux à 64 bits
  - `rax, rbx, ...rbp, rbp, rsi, ...`
- Ajout de 8 nouveaux registres généraux de 64 bits
  - `r8, ...r15`
- Bit NX présent

Digression - L'architecture x86_64

- Possible de référencer la mémoire relativement au compteur programme `rip`
  - Très intéressant pour compiler du *position independent code*
- L'espace d'adressage est de 256 To
- Convention d'appel :
  - Linux x86 : paramètres poussés sur la stack
  - Linux x86_64 :
    - ★ Paramètres de type integer passés dans `rdi, rsi, rdx, rcx, r8, r9`
    - ★ Paramètres de type float/double passés dans `xmm0, ..., xmm7`
    - ★ Le reste est poussé sur la stack

Du binaire au code source

- Le reverse-engineering consiste à retrouver un équivalent fonctionnel du code source à partir du binaire compilé.
- La compilation est une **opération à sens unique**.
  - donnée de 32 bits : pointeur, integer, float, . . . ?
- Equivalence **fonctionnelle**.

Les outils

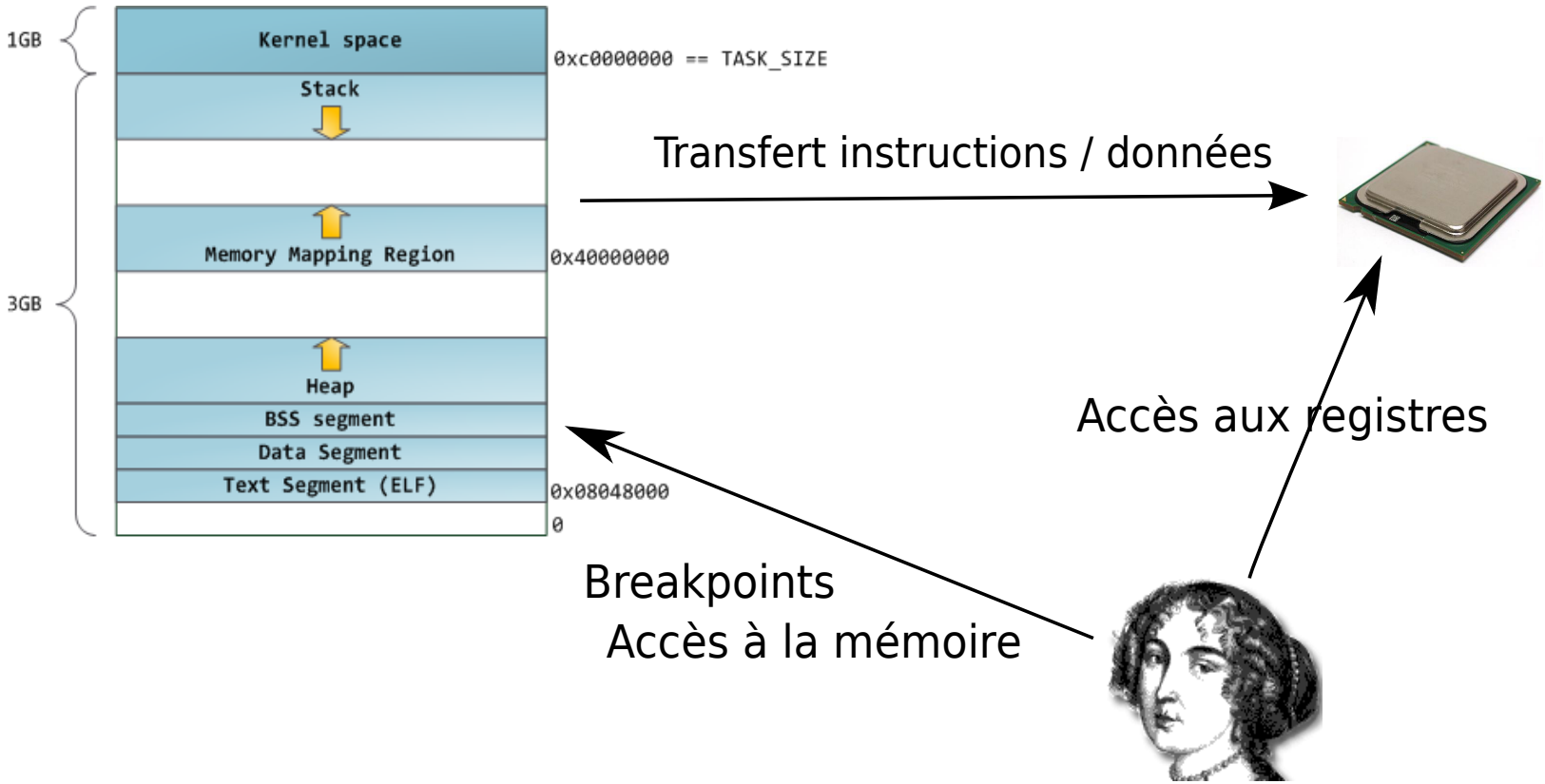
- **Analyse statique** : observer le binaire sur le disque dur
  - *Editeur hexadécimal* : étude des binaires, sans interprétation
  - *Désassembleur* : code binaire vers assembleur
  - *Décompileur* : code binaire vers code source !
- **Analyse dynamique** : observer le binaire “en action”
  - *Débugger* : étude du binaire en mémoire
  - *Outils de monitoring* : étude de l’environnement du binaire (activité réseau, disque, . . .)

Les désassembleurs

- Code machine vers assembleur
- Explorer un code assembleur facilement
- **Control Flow Graph** : reconnaitre les structures classiques
  - `if then else`
  - boucles
  - switches
- **Call Graph** : quelle fonction appelle qui ?

DEMO AVEC IDA

Les debuggers



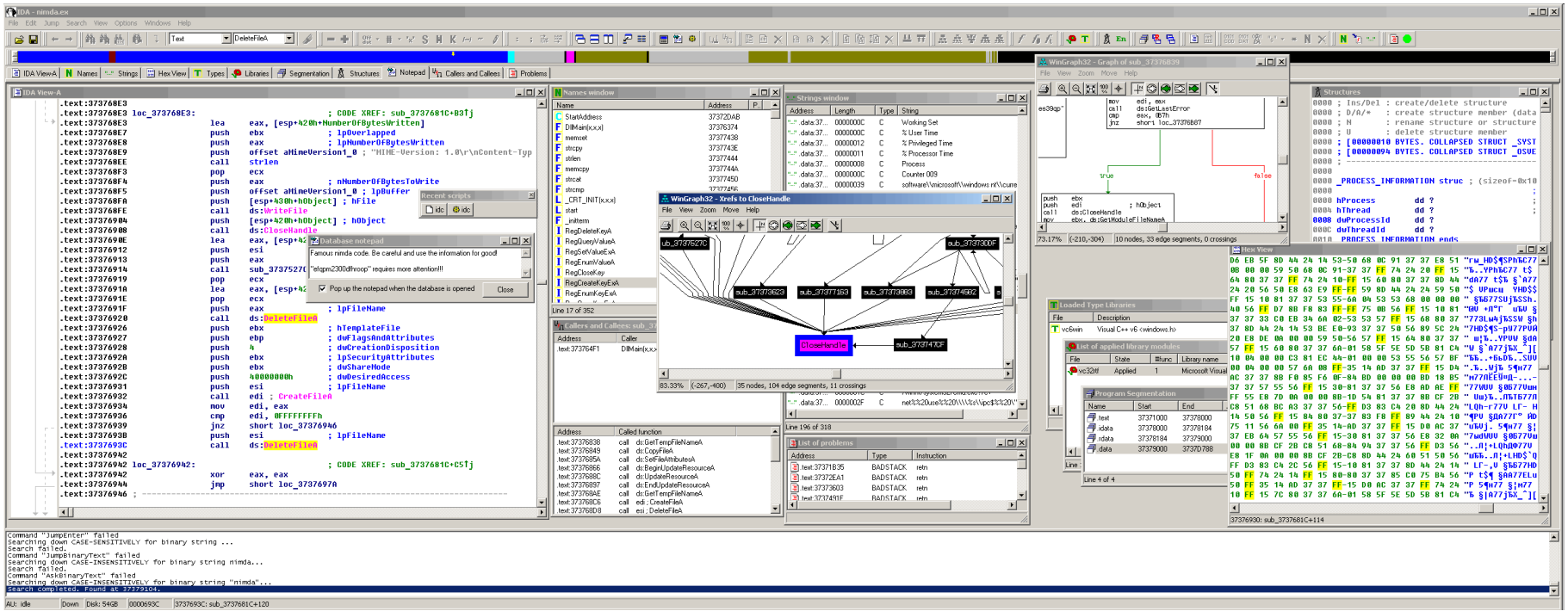
Les debuggers

- Vision dynamique du code lors de son exécution.
- Arrêter l'exécution et étudier un état précis.
- Changer les registres / la mémoire / le code.

```
gdb$ break main
Breakpoint 1 at 0x80483d7
gdb$ run
──────────────────────────────────────────────────────────────────────────────[regs]
  EAX: 0xBFFFF874  EBX: 0xB7FBAFF4  ECX: 0xF25BA99A  EDX: 0x00000001  o d I t s Z a P c
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0xBFFFF7C8  ESP: 0xBFFFF7C8  EIP: 0x080483D7
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
──────────────────────────────────────────────────────────────────────────────[code]
=> 0x80483d7 <main+3>:  and    esp,0xfffffff0
   0x80483da <main+6>:  sub    esp,0x20
   0x80483dd <main+9>:  cmp    DWORD PTR [ebp+0x8],0x2
   0x80483e1 <main+13>: je     0x80483fb <main+39>
   0x80483e3 <main+15>: mov    DWORD PTR [esp],0x8048510
   0x80483ea <main+22>: call   0x80482f8 <puts@plt>
   0x80483ef <main+27>: mov    DWORD PTR [esp],0xffffffff
   0x80483f6 <main+34>: call   0x8048308 <exit@plt>
──────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x080483d7 in main ()
gdb$
```

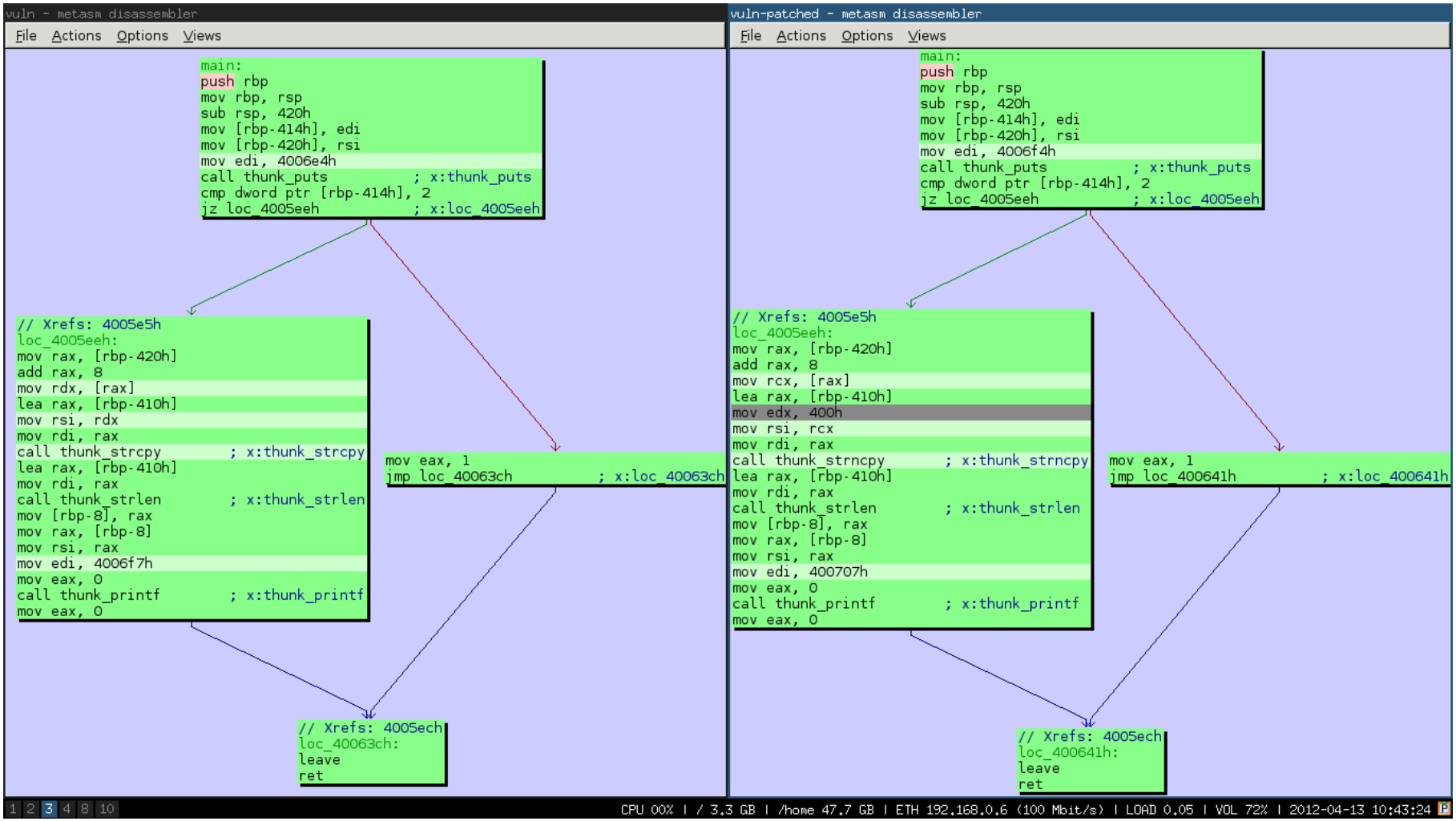
IDA Pro - La Rolls Royce



Pour les fauchés - Metasm

- Framework complet de reverse-engineering écrit en Ruby
- Assembleur, compilateur, linker, désassembleur, décompilateur, débugger
- Nombreuses architectures supportées
- Développé par des français !
- `http://metasm.cr0.org/`

Metasm - bindiff



Hopper - Un peu plus simple

- Framework de reverse-engineering un peu moins complet
- Mais plus simple à utiliser !
- `http://www.hopperapp.com/`

Hopper - GUI


File Edit Find Modify Navigate Scripts Window Help
exemple-pile-bof
Search
Labels
Strings
Tags
_init
__libc_start_main
__gmon_start__
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
register_tm_clones
foo
main
__libc_csu_init
__libc_csu_fini
_fini
_IO_stdin_used
__FRAME_END__
__frame_dummy_init_array_entry
__do_global_dtors_aux_fini_array_e
__JCR_END__
_DYNAMIC
_GLOBAL_OFFSET_TABLE_
__libc_start_main
__gmon_start__
__dso_handle
__TMC_END__
__libc_start_main
__gmon_start__
_ITM_deregisterTMCloneTable
__libc_start_main@@GLIBC_2.2.5
__gmon_start__
_Jv_RegisterClasses
_ITM_registerTMCloneTable
; Section .text
; Range 0x4003c0 - 0x4005b2 (498 bytes)
; File offset 960 (498 bytes)
_start:
deregister_tm_clones:
register_tm_clones:
__do_global_dtors_aux:
; endp
BEGINNING OF PROCEDURE
File Information
Path
Loader ELF
CPU intel/x86_64
Calling Convention System V
Graphic Views
Instruction Encoding
Format
Comment
Colors and Tags
Is Referenced By
Address
Instruction
Remove Reference
Add Reference
Has Reference To
> analysis section .got
> analysis section .got.plt
> analysis section .data
> analysis section .bss
Analysis segment External Symbols
> analysis section External Symbols Section
Background analysis ended
Address 0x4003c0, Segment Segment 2, _start + 0, Section .text, file offset 0x3c0

Hopper - Décompilateur

```
function foo {
    var_24 = LODWORD(arg0);
    var_28 = LODWORD(arg1);
    var_2C = LODWORD(arg2);
    var_10 = 0x41414141;
    var_C = 0x41;
    var_20 = 0x4242424242424242;
    var_18 = 0x4242;
    rax = var_10;
    *rax = 0x4343434343434343;
    *(rax + 0x8) = 0x4343434343434343;
    *(rax + 0x10) = 0x4343434343434343;
    *(int32_t *)(rax + 0x18) = 0x43434343;
    *(int16_t *)(rax + 0x1c) = 0x43;
    return rax;
}
```

Récapitulatif - La pile - Adresse de retour

- L'architecture x86 utilise une **pile**.
- Elle sert à stocker les **variables locales** des fonctions, ainsi qu'à leur passer leurs **arguments**.
- Pour appeler une fonction (en x86), on pousse ses arguments sur la pile.
- Lors du retour d'une fonction, le flot d'exécution se poursuit à l'**adresse de retour**, située au sommet de la pile.



Références

- Cours de Sécurité OS, Télécom ParisTech 2010, Ryad Benadjila
- `http://duartes.org/gustavo/blog/`, Gustavo Duarte
- `http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html`
- `http://hopperapp.com/`, Désassembleur Hopper

Failles Applicatives

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Buffer

- Bloc continu de données du même type
- Langage C : Tableaux
- Allocation des tableaux

Statique	Au chargement du segment de données
Dynamique	Durant l'exécution sur la pile

Organisation mémoire des processus

Régions

- **Text** Instructions et données en lecture seule
- **Data-Bss** Variables statiques
- **Pile** LIFO Format abstrait de données PUSH/POP

Rappel sur la pile

- **PUSH** Permet de pousser une donnée sur la pile
- **POP** Permet de récupérer la dernière donnée de la pile

Intérêt de la pile

Fonctions

La pile permet d'implémenter une hiérarchie de fonctions pour du code haut niveau

Variables

La pile permet d'allouer simplement de la place pour des variables (`esp - imm`)

La pile 1/2

- Un bloc continu de données
- Le haut de la pile est pointé par le registre SP
- Le bas de la pile est à une adresse fixée
- Taille déterminée par le noyau au runtime
- Le CPU implémente les fonctions d'accès à la pile (push/pop)
- Comportement dépendant de l'implémentation

  Intel Grandit vers les adresses basses, SP désigne l'adresse la plus basse de la pile

La pile 2/2

- La valeur de SP change avec les arguments, besoin d'une adresse fixe
- Le registre BP représente le début de la stack frame
- SP et BP sont modifiés lors du prologue et de l'épilogue pour rester cohérents (enter/leave)

Utilisation de la pile

exemple-pile.c

```
void foo(int a, int b, int c)
{
    char tab[5] = "AAAAA";
    char tab2[10] = "BBBBBBBBBB";
}

int main()
{
    foo(1, 2, 3);
}
```

- gcc -S exemple-pile.c -o exemple-pile.s

exemple-pile.s

```
        subl    $12, %esp
        movl    $3, 8(%esp)
        movl    $2, 4(%esp)
        movl    $1, (%esp)
        call    foo
```

Utilisation de la pile

exemple-pile.s

- ```
  foo:
          pushl   %ebp
          movl    %esp, %ebp
          subl    $16, %esp
          movl    $1094795585, -5(%ebp)
          movb    $65, -1(%ebp)
          movl    $1111638594, -15(%ebp)
          movl    $1111638594, -11(%ebp)
          movw    $16962, -7(%ebp)
          leave
          ret
  ```
  ```

Recherche de vulnérabilités

- Une vulnérabilité est un bug amenant un problème de sécurité.
- Comment trouve-t-on ce genre de bug ?

Analyse de code source

- Manuellement : long et fastidieux.
- En s'aidant d'outils automatiques : *Coverity*, *AppScan*, *Valgrind*, *Asan*, ...
- Les outils automatiques renvoient souvent des faux positifs : une partie de l'analyse est toujours manuelle.

Address Sanitizer

- Instrumente le binaire avec des gardes en mémoire.
- Les fonctions sensibles (comme strcpy) sont surveillées.

Utilisation

- gcc -fsanitize=address exemple-pile-bof.c -o exemple-pile-bof
- clang -fsanitize=address exemple-pile-bof.c -o exemple-pile-bof

Address Sanitizer - Exemple

```
dad@himantura ~/Cours/cours_secu_os/code (git)-[master] % ./exemple-pile-bof
=================================================================
==22434==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fffcbb52e15 at pc 0x00000048482c bp 0x7ff
fcbb52d30 sp 0x7fffcbb524e8
WRITE of size 30 at 0x7fffcbb52e15 thread T0
    #0 0x48482b in strcpy (/home/dad/Cours/cours_secu_os/code/exemple-pile-bof+0x48482b)
    #1 0x4b7ae7 in foo (/home/dad/Cours/cours_secu_os/code/exemple-pile-bof+0x4b7ae7)
    #2 0x4b7b77 in main (/home/dad/Cours/cours_secu_os/code/exemple-pile-bof+0x4b7b77)
    #3 0x7f74ef5557ff in __libc_start_main (/usr/lib/libc.so.6+0x207ff)
    #4 0x4b77a8 in _start (/home/dad/Cours/cours_secu_os/code/exemple-pile-bof+0x4b77a8)

Address 0x7fffcbb52e15 is located in stack of thread T0 at offset 85 in frame
    #0 0x4b788f in foo (/home/dad/Cours/cours_secu_os/code/exemple-pile-bof+0x4b788f)

  This frame has 5 object(s):
    [32, 36) ''
    [48, 52) ''
    [64, 68) ''
    [80, 85) 'tab'
    [112, 122) 'tab2' <== Memory access at offset 85 partially underflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow ??:0 strcpy
Shadow bytes around the buggy address:
  0x100079762570: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100079762580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100079762590: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000797625a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000797625b0: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 04 f2 04 f2
=>0x1000797625c0: 04 f2[05]f2 f2 f2 00 02 f3 f3 f3 f3 00 00 00 00
  0x1000797625d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000797625e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000797625f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100079762600: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100079762610: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fuzzing

Fuzzing

Technique consistant à envoyer des entrées invalides à un programme pour déclencher des erreurs.
Elle est employée pour tester une grande variété de logiciels :

- Implémentation de protocoles réseau : ssh, http, protocoles propriétaires
- Parsers : images, texte (html), son (mp3)

Exemples

- `cat /dev/urandom | ncat server 1234`
- `./myprog $(python -c 'print "A"*2048')`

Fuzzing

Avantages

- Ecrire un fuzzer simple est trivial.
- Automatique : de nombreux programmes / librairies existent pour automatiser les tests.

Fuzzing intelligent

- Rester proche des cas valides :
  - Changer quelques octets dans un fichier binaire.
  - Mettre des champs trop longs dans un paquet réseau.

Buffer Overflow

- **Principe** Débordement de l'espace alloué
- **Résultat** Écrasement de la précédente stack frame, de l'adresse de retour (Segmentation Fault)

Buffer Overflow : debugger

Débordement strcpy()

```
void foo(int a, int b, int c)
{
    char tab[5] = "AAAAA";
    char tab2[10] = "BBBBBBBBBB";
    strcpy(tab, "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC");
}

int main()
{
    foo(1, 2, 3);
}
```

Exemples

- **gdb** GNU debugger : très utile pour étudier le déroulement d'un programme
- **lldb** LLVM debugger : plus simple et plus robuste que le précédent

Buffer Overflow : gdb

Debug gdb

```
gdb$
-------------------------------------------------------------------------[regs]
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0xFFFFD358  ESP: 0xFFFFD330  EIP: 0x080483DF
-------------------------------------------------------------------------[code]
=> 0x80483df <foo+59>:  call   0x80482d8 <memcpy@plt>
   0x80483e4 <foo+64>:  leave
   0x80483e5 <foo+65>:  ret
-------------------------------------------------------------------------------
0x080483df in foo ()
gdb$ ni
-------------------------------------------------------------------------[regs]
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0xFFFFD358  ESP: 0xFFFFD330  EIP: 0x080483E4
-------------------------------------------------------------------------[code]
=> 0x80483e4 <foo+64>:  leave
   0x80483e5 <foo+65>:  ret
-------------------------------------------------------------------------------
0x080483e4 in foo ()
gdb$ si
-------------------------------------------------------------------------[regs]
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0x43434343  ESP: 0xFFFFD35C  EIP: 0x080483E5
-------------------------------------------------------------------------[code]
=> 0x80483e5 <foo+65>:  ret
-------------------------------------------------------------------------------
0x080483e5 in foo ()
gdb$ si
-------------------------------------------------------------------------[regs]
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0x43434343  ESP: 0xFFFFD360  EIP: 0x43434343
Cannot access memory at address 0x43434343
0x43434343 in ?? ()
```

Intérêt

- Modification du flot d'exécution
- Exemple :

exemple-pile-ret-modif.c

```c
void foo(int a, int b, int c)
{
    int *retu;
    char tab[5] = "AAAAA";
    char tab2[10] = "BBBBBBBBB";

     /* Compilation little endian
     *  [[BBBBBBBBB][AAAAA][*RET][SEBP][PIE0]
     *                | 5   +  4  +  4   ^
     *                `------------------'   */

    retu = tab + 13;
    (*retu) += 8;
}

int main()
{
    int x;
    x = 0;
    foo(1, 2, 3);
    x = 1;
    printf("%d\n", x);
}
```

Intérêt

- Le programme n'écrit pas 1 !

Retour inattendu

./exemple-pile-ret-modif
0

- Le flot d'instructions peut être modifié simplement
- Possibilité de faire pointer le flot vers du code que l'on a écrit : le shellcode

Shellcode

Definition

Une suite d'instructions assembleur permettant d'exécuter du code arbitraire

- Historiquement, un shell, mais on peut exécuter n'importe quel code
- Exemple :

Shellcode execve

```
xor     eax, eax        ; Logical Exclusive OR
push    eax             ; EAX = 0
push    68732F6Eh       ;
push    69622F2Fh       ; Pousse "//bin/sh"
mov     ebx, esp        ; Enregistre &"//bin/sh"
cdq                     ; EAX -> EDX:EAX (with sign)
push    edx             ; Pousse EDX = 0
push    ebx             ; Pousse &
mov     ecx, esp        ; char *user *
mov     al, 0Bh         ; Numero du syscall
int     80h             ; LINUX - sys_execve
```

Compréhension

- Exécution du shellcode précédent :

Exécution shellcode execve

```
# gcc -o bof bof.c
# ./bof
using address 0xbffff574
bash#
```

- Nécéssite d'avoir le descripteur de fichier stdin ouvert

Idées

Ouvrir un fichier et afficher son contenu
Ouvrir une socket associée à un shell (le descripteur de fichier restera ouvert)

Problèmes

- On ne maitrise plus la suite du programme car la stackframe de la fonction appelante est écrasée
- La taille du shellcode doit être optimisée pour ne pas corrompre l'ensemble du programme
- Se servir de l'espace sûr du buffer pour placer le shellcode et revenir dessus
- Il y a toujours des différences entre les adresses durant le debug et les adresses réelles

Résoudre les problèmes d'adresses

Padding avec des instructions NOP (NOP-sled)
Utiliser l'environnement
Utiliser les librairies chargées

Problèmes

- Pas de caractère nul car il signalerait la fin d'une chaîne de caractères
- Estimer la position de notre shellcode

Pas si simple

Difficile en cas de randomisation de l'espace mémoire du programme (ASLR)
Les systèmes "durcis" surveillent de près les erreurs de segmentation

Exemple de solutions

- Le NOP-sled
- L'utilisation d'espaces non randomisés
- Réutilisation de codes existants, il est plus simple de lancer la fonction system("/bin/sh") à l'aide de la `libc`

Différences avec UNIX

- Les appels systèmes sont gérés par la ntdll qui va les traduire puis lancer une interruption 0x2E
- Ceux-ci sont décrits au travers de la structure noyau référançant les adresses des services systèmes (*SSDT*). Elle peut être indexée par un appel système afin de localiser la fonction en mémoire. Les paramètres sont quand à eux décrits dans la *SSPT*.
- Leurs adresses sont disponibles dans la table `KeServiceDescriptorTable` exportée par le noyau
- Ils sont propriétaires, mais largement disséqués
- Différents pour chaque version de windows !

Shellcoding Windows basé sur les syscalls

Compatible avec une version très précise de windows

Solutions

- On peut utiliser des API documentées qui ne changeront pas avec les versions
- Utiliser le code disponible dans les librairies chargées par le programme
- Par exemple, la librairie `kernel32.dll` est présente dans tous les programmes PE
- Elle contient deux routines qui permettent de récupérer l'adresse d'une fonction, même non exportée

Signatures des fonctions

- **LoadLibrary** : Charge une DLL

Signature

```
HMODULE WINAPI LoadLibrary(
  __in  LPCTSTR lpFileName
);
```

- **GetProcAddress** : Récupère l'adresse d'une fonction dans une DLL

Signature

```
FARPROC WINAPI GetProcAddress(
  __in  HMODULE hModule,
  __in  LPCSTR lpProcName
);
```

Process Environment Block

- Une structure importante de tous les processus Windows est le *PEB*
- Il se trouve à la zone mémoire pointée par `fs: [0x30]` (voir la segmentation), et le *Thread Environment Block* (TEB) en `fs: [0x0]`
- Il contient par exemple la localisation du tas, des *SEH*, de la liste chaînée des DLL chargées (*kernel32.dll*)

Récupération adresse kernel32

```
xor ebx, ebx              // clear ebx
mov ebx, fs:[ 0x30 ]      // get a pointer to the PEB
mov ebx, [ ebx + 0x0C ]// get PEB->Ldr
mov ebx, [ ebx + 0x14 ]// get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov ebx, [ ebx ]          // get the next entry (2nd entry)
mov ebx, [ ebx ]          // get the next entry (3rd entry)
mov ebx, [ ebx + 0x10 ]// get the 3rd entries base address (kernel32.dll)
```

Heap overflow

- **Principe** : Utilisé plus d'espace que celui alloué pour utiliser les metadatas de l'entrée suivante dans la liste doublement chaînée.
- Chaque entrée est stoquée comme metadata + chunk (.ptr et .size), et sont souvent proches en mémoire
- On peut complètement réécrire les metadatas pour exécuter du code arbitraire lors d'un appel à `free()` par exemple.

Définition structure header

```
struct malloc_chunk {
  size_t               prev_foot;  /* Size of previous chunk (if free).  */
  size_t               head;       /* Size and inuse bits. */
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
};
```

Exploitation - Petits chunks

Méthode

On peut réécrire l'adresse d'une fonction dans la *Global Offset Table* du programme.

Définition unlink

```
/* Unlink a chunk from a smallbin  */
#define unlink_small_chunk(M, P, S) {\
    mchunkptr F = P->fd;\
    mchunkptr B = P->bk;\
    F->bk = B;\
    B->fd = F;\
}
```

Exemple

Heap Overflow Classique

```
int main(void)
{
    char* overflow = malloc(768);
    char* p = malloc(64);
    scanf("%s", overflow);    // Ecrasement
    free(overflow);
    free(p);
    return 0;
}
```

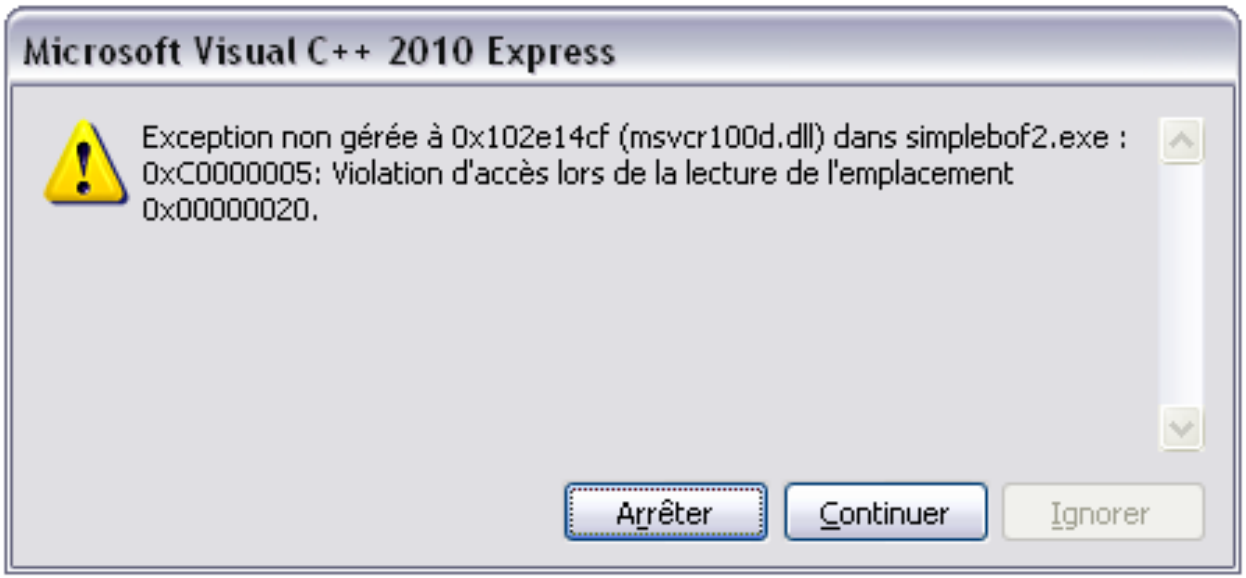
Crash

```
% perl -e 'print "A"x2500' | ./exemple-heap-overflow
*** glibc detected *** ./exemple-heap-overflow: free():
invalid next size (normal): 0x0000000001cd6010 ***
```

Format Strings

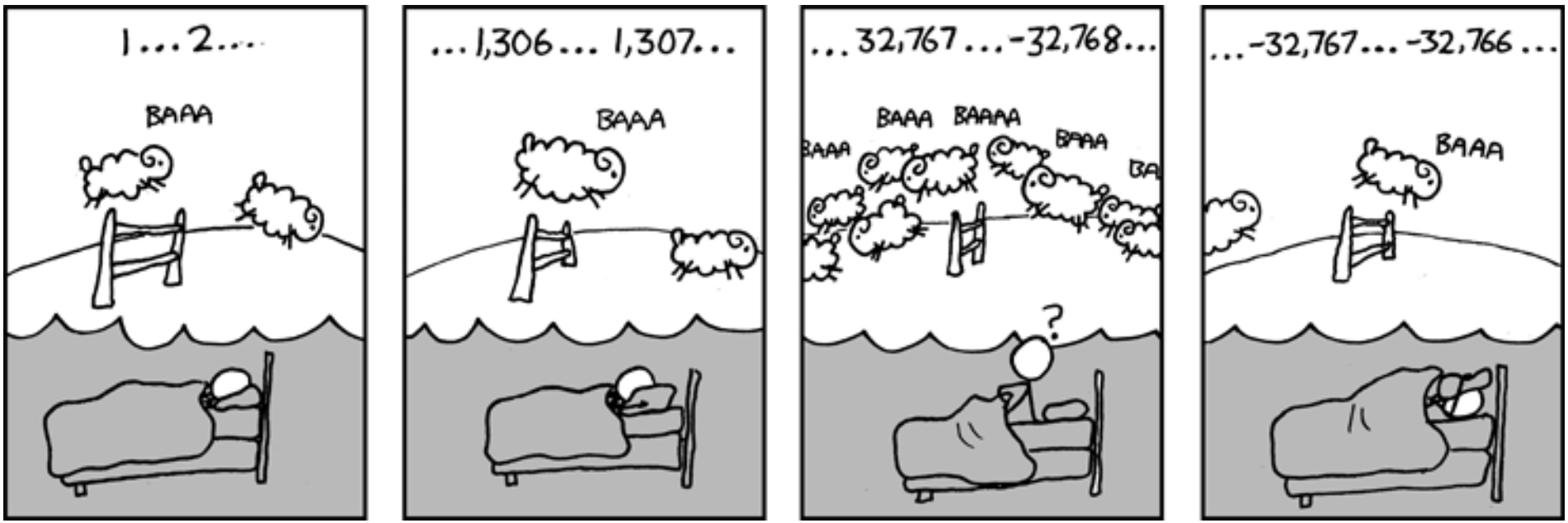
- **Principe** : Profiter d'une mauvaise utilisation du paramètre de format dans les fonctions `printf`.
- On peut récupérer simplement les valeurs sur la pile avec `%p`, puis allez lire et écrire n'importe où avec `%s` et `%n`.
- Exemple :

```
int bof(char *str)
{
    printf(str);
    return 0;
}
int main()
{
    int x, y;
    bof("%s.%s.%s.%s.%s");
    return 0;
}
```



Integer overflow

- **Principe** : Il s'agit de profiter d'une mauvaise représentation mémoire d'un entier
- Par exemple, un débordement d'entier intervient lorsque le compteur d'une voiture passe de 999 999 à 0
- Ces erreurs de représentations permettent généralement d'outrepasser les vérifications de sécurité et d'aboutir à des race conditions



Structured Exception Handler

- Grâce au gestionnaire d'exceptions Windows, les erreurs comme les divisions par zéros ou les fautes de pages peuvent être gérées simplement avec les *SEH*
- Cependant les handlers sont poussés sur la pile comme élément d'une liste chaînée
- On peut donc déborder sur le *SEH* pour le réécrire, il suffit ensuite de déclencher une faute pour pouvoir exécuter n'importe quel code arbitraire.

Problème

Certains systèmes utilisent les **SafeSEH** et le **SEHOP** pour prévenir la réécriture des *SEH* simplement

Structured Exception Handler

- Exemple de code avec gestion des exceptions, beaucoup de A pour déclencher une exception immédiate

Bof avec exception

```
int bof(char *str)
{
    char buf[10];
    __try
    {
        strcpy(buf, str);
    }
    __except(1)
    {
        printf("Perdu\n");
    }
    return 0;
}
int main()
{
    int x, y;
    /* en pratique, beaucoup plus de A */
    bof("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
    return 0;
}
```

SEH - Écrasement

- Une chaîne d'exception est mise en place

Bof avec exception

```
int bof(char *str)
{
    char buf[10];
    __try
    {
        strcpy(buf, str);
    }
    __except(1)
    {
        printf("Perdu\n");
    }
    return 0;
}
int main()
{
    int x, y;
    /* en pratique, beaucoup plus de A */
    bof("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
    return 0;
}
```

SEH - Écrasement

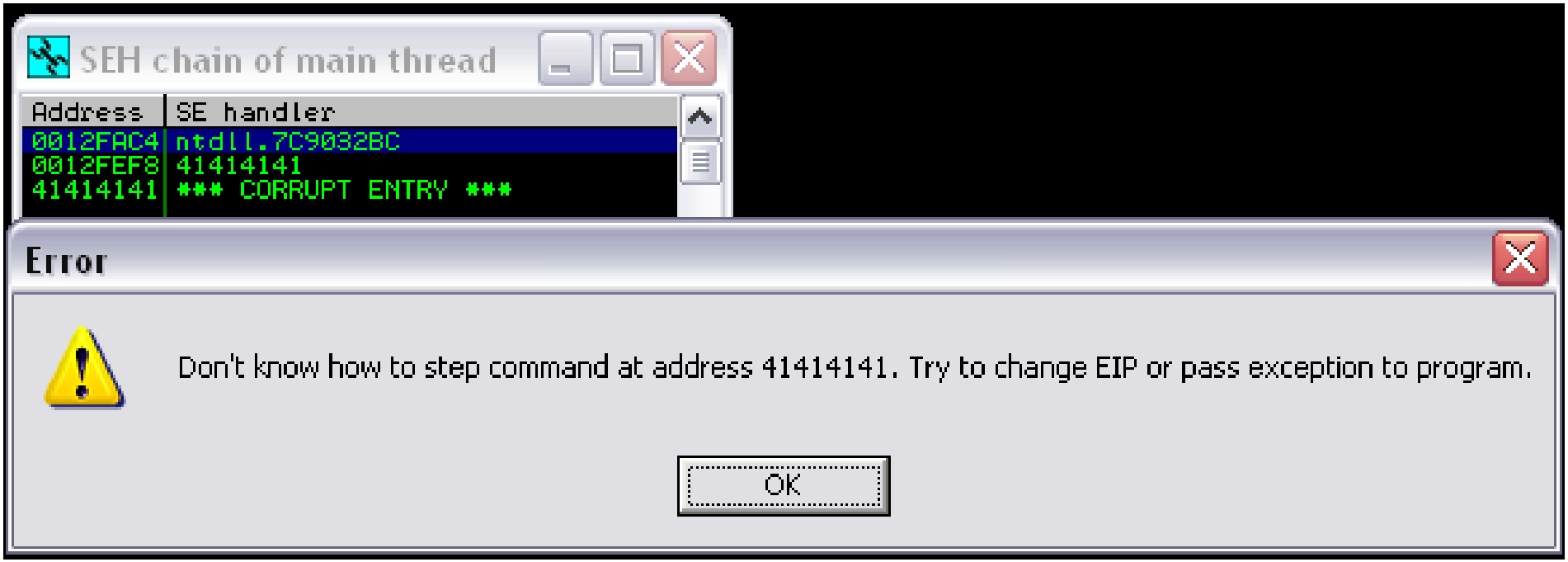
- Chaque exception s'ajoute à la fin de la chaîne quand il le faut, et s'enlève si tout c'est bien passé

Écrasement avec SEH

```
0012FEE0   00000000  ....
0012FEE4   00000000  ....
0012FEE8   00000004  ...
0012FEEC   FE9DC470  pÃĎxÃ¿ Cookie
0012FEF0   0012FE90  xÃ¿   sEBP
0012FEF4   00000000  ....
0012FEF8   0012FFA8  ÂÍÃ£   Pointer to next SEH record
0012FEFC   00411041  AA.  SE handler
0012FF00   FECE69E0  ÃăiÃŪÃ¿
0012FF04   00000000  ....

0012FEE0   41414141  AAAA
0012FEE4   41414141  AAAA
0012FEE8   41414141  AAAA
0012FEEC   41414141  AAAA
0012FEF0   41414141  AAAA
0012FEF4   41414141  AAAA
0012FEF8   41414141  AAAA  Pointer to next SEH record
0012FEFC   41414141  AAAA  SE handler
0012FF00   41414141  AAAA
0012FF04   41414141  AAAA
```

SEH - Démonstration



- On a réussi à casser la chaîne en introduisant un pointeur vers notre exception

Return to libc

- On a déjà évoqué le concept précédemment, réutiliser le code existant pour exécuter directement des routines contenues dans la `libc`
- Il est plus simple de pousser *"/bin/sh"* puis de faire un call `system@libc` que de construire la pile avec les arguments de execve et de faire l'appel système ! Cela prend aussi moins de place
- Des variantes de cette technique existent, *ret2code*, *ret2reg*, *ret2got* ou encore *ret2ret* !
- Il s'agit d'une technique qui permet d'outrepasser des restrictions système, en appelant `mprotect` pour rendre notre pile exécutable par exemple
- Il faut cependant connaître les adresses exactes des libraires, ce qui la rend peu portable

Conseils

- Toujours utiliser les **versions sécurisées des fonctions** copiant des entrées non contrôlées, `strncpy_s`, `snprintf_s`...
- Faire appel à des programmes qui implémentent leurs gestionnnaires de mémoire pour détecter les fuites mémoires, écritures invalides ou bien les comportements douteux.
  - Valgrind
  - Fortify
- Protéger la pile grâce aux **canaries**, qui vont vérifier que l'adresse de retour n'a pas été modifiée
- Utiliser les outils de durcissement système, comme les patch **GrSsec** pour noyaux linux et les options noyau pour rendre les piles non exécutables (**NX**)
- Sous windows, il faut compiler les programmes avec les options SEH **/SAFESEH et /GS**

Récapitulatif - Exploitation

- Le but est de **détourner le flot d'exécution** normal d'un programme.
- L'origine est souvent un **bug** (débordement de buffer).
- Il faut alors réussir à faire exécuter quelque chose d'intéressant au programme (et pas de le faire planter) : **shellcode**, **payload**.
- Il existe beaucoup de techniques d'attaque...
- ...et de nombreuses contre-mesures !



Références

- `ftp://g.oswego.edu/pub/misc/malloc.c`
- man gcc, man clang, grep sanitize
- `https://sploitfun.wordpress.com/2015/06/26/linux-x86-exploit-development-tutorial-series/` Merci Hydraze
- Fuddly `https://github.com/k0retux/fuddly`

Protection des OS

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Problèmes

- Injection et exécution de code arbitraire
- Différences entre Windows et Unix
- Présentation non exhaustive des techniques les plus utilisées

  Pourquoi Description de l'origine des buffers overflows

  Comment Description de techniques de durcissement du système

Origine des buffer overflows

- Les systèmes peuvent adopter plusieurs type de mémoires

  Flat model L'adressage mémoire est vu comme un bloc contigu linéaire. Tout le code, la pile et les données (.data) sont contenus dans un même espace d'adressage.

  Segmented La mémoire est vue comme un ensemble de segments (accessibles via les regitres du processeur `CS`, `DS`, `ES`, `SS`, `FS`, `GS`)

  Real mode Type supporté pour des raisons de compatibilité, l'espace d'adressage linéaire est un tableau de segments

Origine des buffer overflows

- Le mode *Flat model* est le plus répandu pour des raisons de performances, mais il permet d'accéder aux adresses mémoire de la même façon quelque soit les segments.
- L'injection de code se fait via les segments `DS` et `SS`
- L'exécution de code se fait via le segment `CS`
- Ces deux segments fournissent la même traduction d'*adresse logique* en *adresse linéaire* puis *physique*
- On peut donc exécuter des données en les référançant par la même *adresse virtuelle (linéaire)* !

Origine des buffer overflows

- Les segments permettent de n'exécuter du code qu'à partir du segment `CS`, et que le code des segments de data ne soient pas exécutable
- Il est donc possible d'avoir des buffer overflows en *flat memory model* et sans protection au niveau des pages

Protections

Comment les constructeurs et les développeurs ont ils remédiés à ces problèmes ?

Niveaux de protections

Plusieurs niveaux de contrôle possibles

- Directement à l'exécution via l'ajout de routines de contrôle
- Dans l'OS pour gérer directement la mémoire de façon plus ou moins transparente

Niveau Compilateur

- Il faut protéger la pile, par exemple
  - Comparaison de la valeur d'une variable sur la pile avec une variable connue
  - Une valeur aléatoire est calculée et placée avant l'adresse de retour
  - Avant d'exécuter le RET, un `__stkcheck(valRandom)` est exécutée
  - Si un tampon est débordé et que la pile n'est plus cohérente une exception est remontée et le programme stoppé

Problèmes

La valeur de la précédente stack frame peut être écrasée pour modifier le flot d'exécution
Les *format strings* ne sont pas impactées

Niveau OS

- L'OS est une zone de contrôle privilégiée permettant de définir finement les droits des segments, pages et les fonctions.
- Plusieurs idées ayant différents noms suivant les systèmes :
  - Unix
    - OpenWall
    - ASCII Armor
    - Bit NX
    - Exec-shield et W^X
    - Pax
  - Windows
    - SafeSEH
    - ASLR

Compilateur - Unix

- Sous Unix, le canary est inclus par défaut depuis gcc4 (stack-protector)
- gcc fourni une option `-D_FORTIFY_SOURCE=2` permettant de remplacer des fonctions dangereuses (`printf`, `strcpy`...) par des versions plus sûre
- Illustration canary

OpenWall

- Tout premier patch Unix visant à combiner des mécanismes noyau et processeur pour empêcher l'exécution de code sur la pile
- Le segment `SS` devient non exécutable
- Impose des restrictions sur `/tmp` et `/proc` pour éviter les fuites d'informations
- Détruit les segments partagés non utilisés

Impact

Le code exécutable ne peut pas être sur la pile

Inconvénients

Les retours à la libc outrepassent purement et simplement cette protection
Des protections trop fortes peuvent modifier complètement le fonctionnement du programme (Oracle)

ASCII Armor

- Une zone "ASCII Armor" est mise en place entre 0x00000000 et 0x01010100
- L'ensemble du code exécutable est déplacé dans cette zone
- Il devient très difficile de placer l'adresse des fonctions

Pourquoi ?

Les fonctions comme `strcpy` considèrent le caractère nul comme la fin d'un string
Mettre un caractère nul arrête la copie du shellcode $\Rightarrow$ il n'est plus cohérent

Problèmes

L'espace protégé n'est pas très grand et peut ne pas suffir, la protection devient alors moins forte

Bit de non exécution des pages

- Souvent appelé bit NX (Non eXecute)
- Cela permet d'empêcher strictement l'exécution de code sur certaines pages
- Il permet d'ajouter un flag d'exécution aux pages, de façon hardware ou software
- Les constructeurs l'ont implémentés depuis les séries Intel Pentium 4 et AMD Athlon 64
- Leurs présence ne suffit pas, il faut des mécanismes OS pour les utiliser

Exec-shield

- Développé par RedHat, il introduit des changements profonds dans le coeur du code noyau
- Support du bit NX sur la majorité des architectures
- Emule le bit NX grâce à la limite du segment de code sur les architectures IA32

PaX

- Fournit le bit NX et les fonctionnalités NX sur tous les systèmes
- Implémentation beaucoup plus complète que Exec-Shield, mais qui peut provoquer des disfonctionnements dans les applications légitimes
- Utilisé dans toutes les distributions "durcies" (Gentoo Hardened)
- Plusieurs techniques disponibles :
  - PAGEEXEC et SEGMEXEC
  - ASLR
  - Hook de la fonction mprotect() pour un contrôle plus fin des droits d'accès à la modification des pages



PaX - PAGEEXEC

- Permet l'utilisation ou l'émulation du bit NX
- Le bit Superviseur est surchargé pour représenté le bit NX
- Un accès à une page qui n'a pas été mise en cache dans le *TLB* (Table Lookaside Buffer) provoque une faute de protection
- La MMU informe l'OS de la faute
  - Si c'est une faute *ITLB* (exécution), le processus est stoppé
  - Si c'est une faute *DTLB* (read/write), l'exécution continue normallement

Remarque

La mémoire n'est pas divisée, chaque processus possède 3GB de mémoire virtuelle (IA32)
Très lent

PaX - SEGMEXEC

- La fonctionnalité du bit NX est émulée en sépérant l'espace d'adressage en deux moitiés mises en mirroir (`do_mmap()`)
- Un premier descripteur de segment de données couvre l'espace 0 - 1.5 GB d'adresses linéaires
- Un second descripteur de segment de code couvre l'espace 1.5 - 3 GB
- La récupération d'une instruction est traduite sur la séparation de l'espace d'adressage
- Si le code n'appartient pas à la bonne moitié, le programme s'arrête

Problème

Les performances sont limitées car l'espace disponible n'est que de **1.5GB pour le code ou les données**

PaX - mprotect()

- PaX assure que la RAM n'est en aucun cas en écriture et exécutable (`PROT_WRITE` et `PROT_EXEC`)
- Pour modifier cela, la fonction `mprotect()` permet de spécifier les permissions d'une zone mémoire et peut donc être appellée lors d'une exploitation
- PaX injecte donc sa version de `mprotect()` avec un return Permission Denied si les flags `PROT_WRITE` et `PROT_EXEC` sont passés en même temps

Remarques

Empêche l'exécution de code sur les pages contenant des données non contrôlées
Incompatibilités logicielle : Java (car ils ne savent pas se servir de mprotect...)

W^X

- OpenBSD a sorti W^X pour maitriser les permissions des segments
- Le signal trampoline n'est plus au dessus de la pile
- Le segment de pile est marqué comme non exécutable
- La PLT et la GOT n'ont plus de code exécutable, ld.so doit être légèrement modifié
- Si la granularité de l'exécution de page n'est pas disponible, on en revient à définir un ensemble d'exécution
- Les zones exécutables sont séparés des zones non exécutables en deux parties distinctes

Address Space Layout Randomization

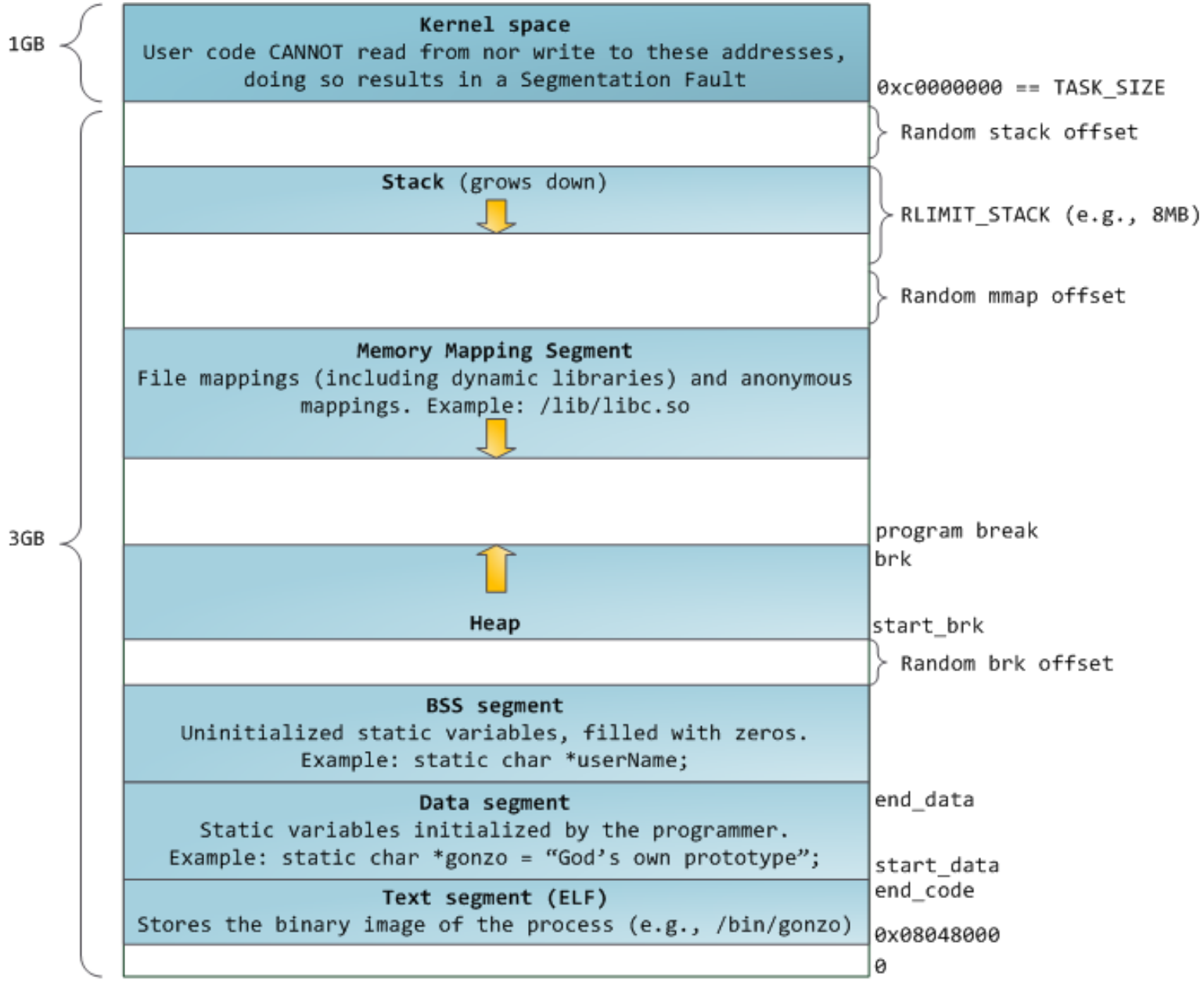
- L'ASLR permet d'introduire de l'aléa dans les adresses utilisées par un processus

RANDMMAP	Randomise la mémoire gérée par mmap() (librairies, tas, pile de thread, mémoires partagées)
RANDEXEC	Randomise la mémoire gérée par brk() et les segments de data, bss, et code de l'exécutable
RANDUSTACK	Randomise la pile utilisateur
RANDKSTACK	Randomise la pile noyau

Remarques

- Attaque sur l'entropie possible
- RANDEXEC peut corrompre l'exécution de certains programmes qui utilisent des adresses en dur (non relogeables)
- La majorité des programmes est compilé sans ASLR ! (*checksec.sh*)

Address Space Layout Randomization



Vérification automatique

Un script a été mis au point afin de vérifier que des protections sont présentes sur un système donné, checksec.sh

Exemple checksec.sh

```
% checksec.sh --proc-all
* System-wide ASLR (kernel.randomize_va_space): On (Setting: 2)

  Description - Make the addresses of mmap base, heap, stack and VDSO page randomized.
  This, among other things, implies that shared libraries will be loaded to random
  addresses. Also for PIE-linked binaries, the location of code start is randomized.

  See the kernel file 'Documentation/sysctl/kernel.txt' for more details.

* Does the CPU support NX: Yes

         COMMAND    PID RELRO           STACK CANARY          NX/PaX        PIE
         awesome   1262 No RELRO        No canary found       NX enabled    No PIE
    dbus-launch    1273 No RELRO        No canary found       NX enabled    No PIE
    dbus-daemon    1274 Partial RELRO   No canary found       NX enabled    PIE enabled
          xterm    1321 No RELRO        No canary found       NX enabled    No PIE
            zsh    1322 No RELRO        No canary found       NX enabled    No PIE
            mpd   14669 No RELRO        No canary found       NX enabled    No PIE
            vim   20918 No RELRO        No canary found       NX enabled    No PIE
            ssh    2317 No RELRO        Canary found          NX enabled    No PIE
          mupdf    2477 No RELRO        No canary found       NX enabled    No PIE
        firefox    2571 No RELRO        No canary found       NX enabled    No PIE
       gconfd-2    2590 No RELRO        No canary found       NX enabled    No PIE
```

Compilateur - Windows

- Sous Windows, le canary est inclus par défaut le flag /**GS**
- Protection de la gestion des exceptions /**SafeSEH** et **SEHOP**
- Encodage des pointeurs
- Modifications des fonctions sujettes à des débordements de tampons

Compilateur - Windows

- Illustration canary `security_check_cookie`

Bof compilé avec /GS

```
int bof(char *str)
{
    char buf[10];
    strcpy(buf, str);
    return 0;
}
int main()
{
    int x, y;
    bof("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
    return 0;
}
```

Canary Windows - Initialisation

- La valeur du canary est calculée au début du programme

Entrée du programme

```
CPU Disasm
Address   Hex dump          Command                Comments
004113E2  |.  55            PUSH EBP
004113E3  |.  8BEC          MOV EBP,ESP
004113E5  |.  E8 3EFCFFFF   CALL 00411028          ; [__security_init_cookie
004113EA  |.  E8 11000000   CALL __tmainCRTStartup ; [__tmainCRTStartup
004113EF  |.  5D            POP EBP
004113F0  \.  C3            RETN
```

Canary Windows - Initialisation

- Calculé à partir de plusieurs paramètres

Calcul du cookie (__security_init_cookie)

```
CPU Disasm
Address   Hex dump          Command
00411B05  |.  FF15 68714100 CALL DWORD PTR DS:[<&KERNEL32.GetSystemTimeAsFil
...
00411B1A  |.  FF15 6C714100 CALL DWORD PTR DS:[<&KERNEL32.GetCurrentProcessI
...
00411B26  |.  FF15 70714100 CALL DWORD PTR DS:[<&KERNEL32.GetCurrentThreadId
...
00411B32  |.  FF15 74714100 CALL DWORD PTR DS:[<&KERNEL32.GetTickCount>]
...
00411B8B  |.  890D 14604100 MOV DWORD PTR DS:[__security_cookie],ECX ; Canary = 0xB5C33723
```

Canary Windows

Code de la fonction bof

```
CPU Disasm
Address   Hex dump        Command                                  Comments
00411660 /$ \55           PUSH EBP                                 ; INT simplebof2.bof(str)
00411661 |. 8BEC           MOV EBP,ESP
00411663 |. 83EC 50        SUB ESP,50
00411666 |. A1 14604100    MOV EAX,DWORD PTR DS:[__security_cookie]
                                                                   ; cookie = 0xB5C33723
0041166B |. 33C5           XOR EAX,EBP                              ; EAX ^= EBP
0041166D |. 8945 FC        MOV DWORD PTR SS:[LOCAL.1],EAX           ; Pousse EAX sur
                                                                   ; le premier argument
                                                                   ; disponible, en 0x0012FF04
00411670 |. 53             PUSH EBX
00411671 |. 56             PUSH ESI
00411672 |. 57             PUSH EDI
00411673 |. 8B45 08        MOV EAX,DWORD PTR SS:[ARG.1]
00411676 |. 50             PUSH EAX
00411677 |. 8D4D F0        LEA ECX,[LOCAL.4]
0041167A |. 51             PUSH ECX
0041167B |. E8 ACFAFFFF    CALL 0041112C                            ; strcpy(0x12FEF8, &str)
00411680 |. 83C4 08        ADD ESP,8
00411683 |. 33C0           XOR EAX,EAX
00411685 |. 5F             POP EDI
00411686 |. 5E             POP ESI
00411687 |. 5B             POP EBX
00411688 |. 8B4D FC        MOV ECX,DWORD PTR SS:[LOCAL.1]           ; Recupere la valeur
                                                                   ; du cookie^EBP
                                                                   ; ECX = 0x41414141
0041168B |. 33CD           XOR ECX,EBP                              ; Calcule le cookie
                                                                   ; ECX = 0x4153BE49
0041168D |. E8 7DF9FFFF    CALL 0041100F                            ; [__security_check_cookie
00411692 |. 8BE5           MOV ESP,EBP
00411694 |. 5D             POP EBP
00411695 \. C3             RETN
```

Evolution de la pile

Pile

- CPU Stack

```
Address   Value       ASCII  Comments
0012FEB8  /10360668   hxx6    Alloue 0x50 octets sur la pile
0012FEF8   00000000           /
0012FEFC   00000000           | Buffer pour la chaine de caractÃÍres
0012FF00   00000000           \
0012FF04  |B5D1C82B   +ÃĹÃŚÂţ  Valeur du cookie^EBP
0012FF08  /0012FF68   hÃ£     Push EBP
0012FF0C  \004112A3   Âč      RET vers main+13
0012FF10  /0041496C   lIA    Pointeur vers str = "AAA..."

CPU Stack
Address   Value       ASCII  Comments
0012FEF8   41414141   AAAA
0012FEFC   41414141   AAAA
0012FF00   41414141   AAAA
0012FF04   41414141   AAAA   Cookie^EBP ecrase
0012FF08   41414141   AAAA   EBP ecrase
0012FF0C   41414141   AAAA   Ret ecrase
```

Arrêt du programme

- Comparaison des valeurs, et arrêt du programme

Vérification du cookie

```
CPU Disasm
Address   Hex dump          Command
; simplebof2.__security_check_cookie(void)
00411EF0  /$ \3B0D 14604100 CMP ECX,DWORD PTR DS:[__security_cookie]
; Compare simplement le cookie a la valeur calculee au lancement
00411EF6  |.  75 02         JNE SHORT 00411EFA
; Erreur! On jump vers __report_gsfailure
00411EF8  \.  F3:C3         REP RETN
```

report_gs_failure

```
CPU Disasm
Address   Hex dump          Command
00411F10  |> \8BFF          MOV EDI,EDI
...
00411FE4  |.  FF15 80714100 CALL DWORD PTR DS:[<&KERNEL32.SetUnhandledExcept
...
00411FEF  |.  FF15 5C714100 CALL DWORD PTR DS:[<&KERNEL32.UnhandledException
...
00412014  |.  FF15 64714100 CALL DWORD PTR DS:[<&KERNEL32.TerminateProcess>]
```

Protection SEH

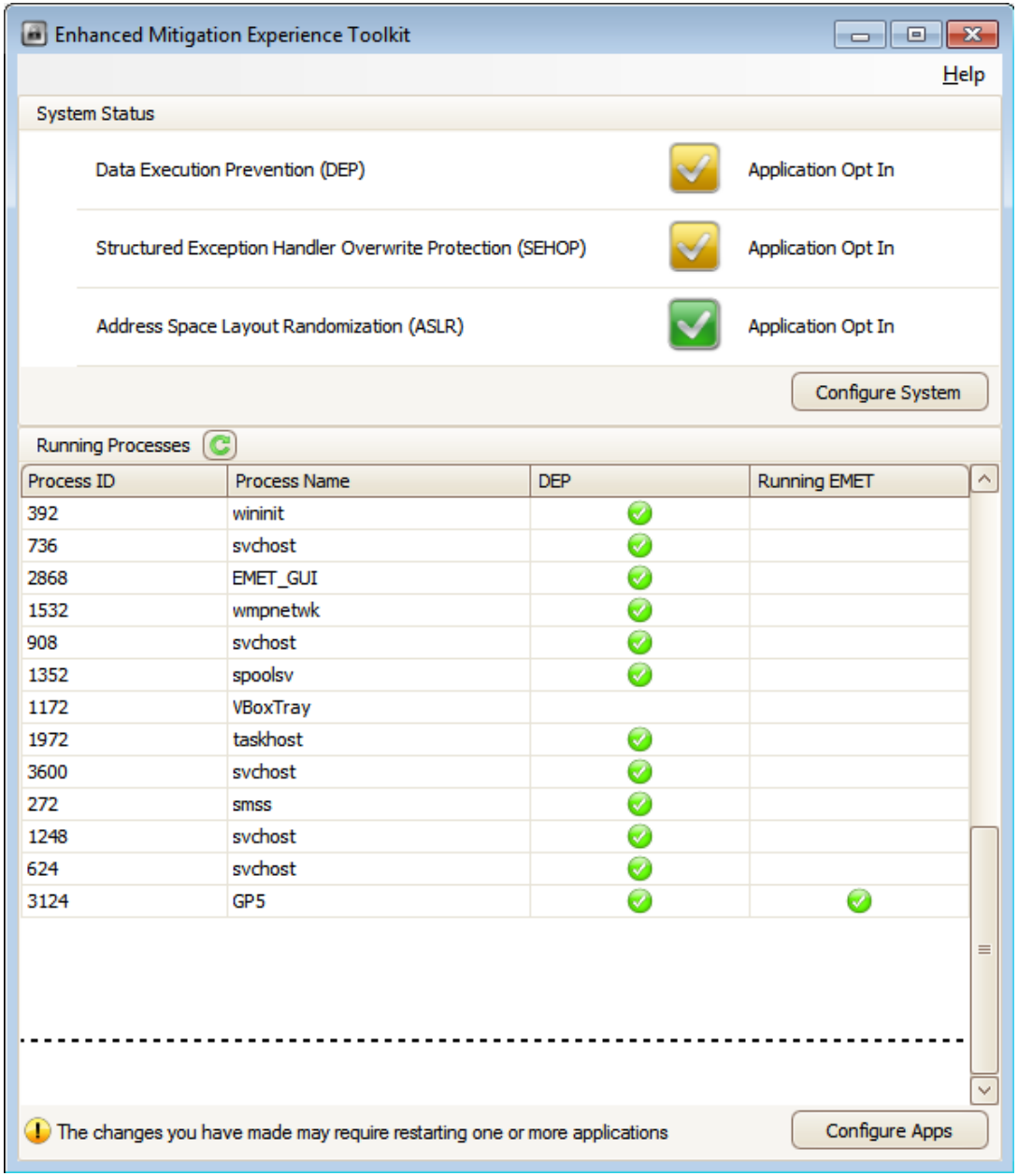
Protections

SafeSEH : Les handlers sont comparés à une liste statique, bypass possible si une DLL n'est pas compilée avec le flag /SafeSEH
Structure Exception Handler Overwrite Protection (SEHOP) : Un cookie est mis en place á la fin de la chaîne, pas de technique de bypass connue

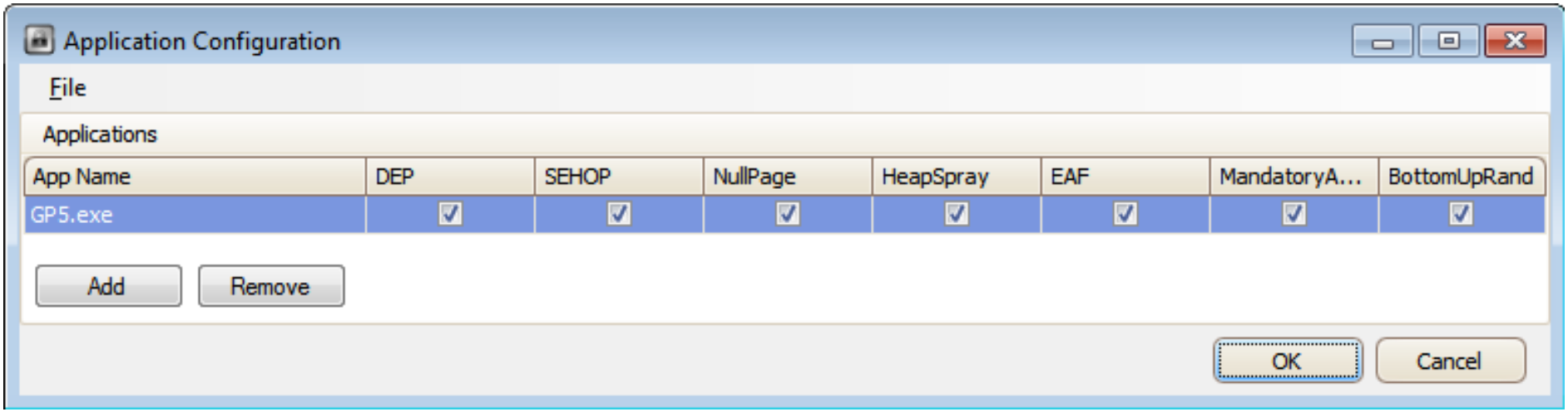
Address Space Layout Randomization

- L'outil *Enhanced Mitigation Experience Toolkit* (EMET) de Microsoft permet de :
  - Gérer dynamiquement le DEP
  - Mettre en place le SEHOP
  - Prévient l'allocation de page nulles et de données répétées sur le tas (Heap Spray)
  - Filtrer la table des adresses d'exportations (EAF)
  - Forcer l'ASLR

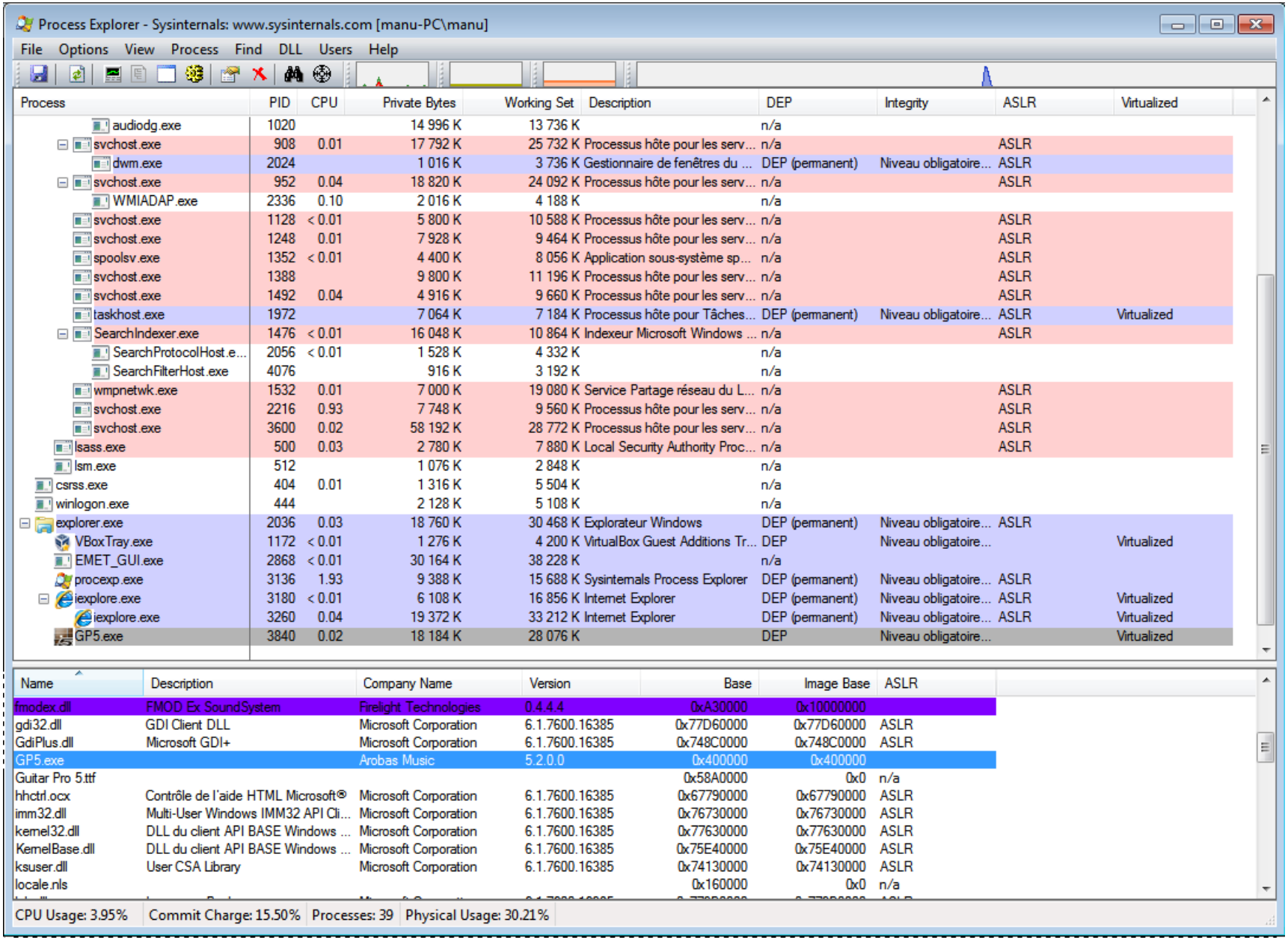
Address Space Layout Randomization



Address Space Layout Randomization



Address Space Layout Randomization - GP5



Sécurité OS Windows

- Windows 7 est un des OS les plus compliqué à exploiter, par défaut :
  - **ASLR** sur l'ensemble des librairies et programmes (mais certaines versions antérieures de l'OS non)
  - **Bit NX** via le mécanisme de Data Execution Prevention (DEP)
  - Les Structured Exception Handler (SEH) se voient ajoutées des couches de protection, les **SafeSEH** et le **SEHOP**

Récapitulatif - Méthodes de protection

- Plusieurs solutions existes pour empêcher l'exploitation de vulnérabilités :
  - Au niveau de la **compilation**
  - Au niveau de l'**OS**
- Ces deux méthodes sont complémentaires.



Références

- http://www.openbsd.org/papers/ven05-deraadt/mgp00009.html
- http://www.tenouk.com/cncplusplusbufferoverflow.html
- http://pax.grsecurity.net/docs
- http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html
- http://tk-blog.blogspot.com/search/label/checksec.sh
- http://msdn.microsoft.com/en-us/library/aa290051%28v=vs.71%29.aspx
- http://support.microsoft.com/kb/2639308/en-us

Conclusion

Telecom Sud Paris

Aurélien Wailly

Amazon
Dublin

Tenable?
Paris

30 Mai 2022

Ouf !

- La sécurité est un sujet vaste !
- Qu'a-t-on appris ?
  - La sécurité des OS utilise très souvent des fonctionnalités du hardware.
  - La gestion de la mémoire est un point capital : stabilité, isolation.
  - Systèmes multi-utilisateurs : problématiques et implémentations UNIX / Windows.
  - Comment fonctionne un binaire, outils d'analyse.
  - Méthodes d'exploitation de failles.
  - Comment rendre un OS plus résistant aux attaques.

Mais on est loin d'avoir fini !

- Sécurité réseau : protocoles, attaques.
- Malware et botnets
- Cryptographie : pour les matheux !
- Sécurité Web :
  - Souvent le point faible dans un système...
  - Injection SQL, XSS, ...
- Social engineering... ;-)

Pour aller plus loin

- Linux :
  - `http://duartes.org/gustavo/blog/`
  - *Understanding the Linux Kernel*, Daniel P. Bovet, Marco Cesati
  - *The Linux Programming Interface*, Michael Kerrisk
- Windows :
  - *Windows Internals 5th Edition*, Mark E. Russinovich, David Solomon
- Exploitation :
  - *Smashing the stack for fun and profit*, Aleph One
  - `http://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/`
- Assembleur :
  - `http://www.intel.com/products/processor/manuals/`
  - `http://sandpile.org/`

Pour s'entrainer

- NewbieContest : `http://newbiecontest.org/`
- OverTheWire (anciennement PullThePlug) : `http://www.overthewire.org/wargames/`
- Intruded.net (anciennement DiEvo) : `http://intruded.net/wglist.html`
- SmashTheStack.org : `http://smashthestack.org/wargames.html`
- Starlfeet Academy Hackits : `http://isatcis.com/`
- The Python Challenge : `http://www.pythonchallenge.com/`
- Try2Hack : `http://www.try2hack.nl/`
- Hack This Site : `http://www.hackthissite.org/`
- 0x41414141 : `http://0x41414141.com/`
- wg.nx.to : `http://edx.ath.cx:1111/`
- Hackbbs : `http://hackbbs.org/`
- RPISEC Training : `http://rpisec.net/news/show/55`
- AMENRA Wargames : `sshlevel1@ivan.stalkr.net-p13248pass:level1`

Merci pour votre attention - Des questions ?

