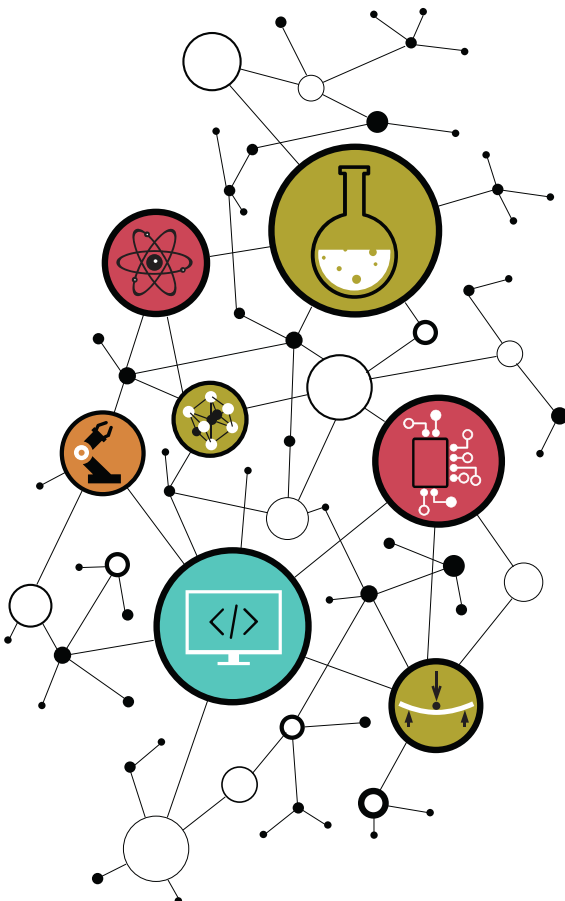


PARTIE 9

HACKING



I. Les bidouilleurs

Le **hacking** est un terme aux significations multiples ayant évolué avec les années et les disciplines concernées. À l'origine il est utilisé par des radio-amateurs qui « *hackaient* » (au sens bricoler, bidouiller¹⁸) les systèmes afin de les améliorer. Il a ensuite été repris par des étudiants du MIT (*Massachusetts Institute of Technology*) pour qui l'étude des ordinateurs et réseaux informatiques se faisait au travers du démontage des matériels et désassemblage des logiciels, sous forme de jeux et de défis. Aujourd'hui, on parlerait de rétro-ingénierie (ou *retro-engineering*) même si ce terme apporte une notion moins artisanale à l'activité d'analyse des systèmes.

De nos jours le terme « *hacker* » est quelque peu galvaudé et aurait (du point de vue du grand public) tendance à désigner un pirate informatique au sens *black-hat* du terme. Un **black-hat** est une personne malveillante, agissant dans l'illégalité et s'introduisant dans un système d'informations (ordinateur, serveur, réseau, ...) afin d'en tirer un bénéfice (dégradations des services, obtentions d'informations sensibles, demandes de rançons, ...). Par opposition on trouvera le **white-hat** qui est un hacker travaillant en toute légalité, recherchant une vulnérabilité ou une faille de sorte à la rendre publique et voire proposer un correctif à l'éditeur. Entre les deux le **grey-hat** serait plutôt un *hacker* avec une éthique mais agissant dans l'illégalité. Il peut se rapprocher du *white-hat* au sens où il recherche des vulnérabilités et en informe l'éditeur concerné, ou il peut désigner le *hacktiviste* prônant une certaine idéologie (par exemple *Anonymous* militant pour la liberté d'expression et le respect de la vie privée).



Dans ce chapitre, nous allons nous introduire au monde du *hacking* en insérant un code malveillant dans un programme existant. Ce code est un **shellcode** : il aura pour objectif d'ouvrir un *shell* (un console), outil indispensable pour ensuite avoir un accès quasi-illimité à l'ensemble de la machine infectée. Ce *shellcode* sera inséré sur la pile afin de ne pas être détecté dans le code binaire du fichier exécutable.

Bien évidemment ceci n'est qu'une brève démonstration d'un monde bien plus vaste. De nombreuses autres solutions de *hacking* existent, comme la *ROP-chain* (*Return-Oriented Programming*) ou l'exploitation de vulnérabilités (*exploit*). Pour ce dernier cas, le site CVE (*Common Vulnerabilities and Exposures*)¹⁹ rassemble toutes les vulnérabilités publiques pour tout un tas d'application.

¹⁸ D'ailleurs le terme français-québécois associé à « *hacker* » est « bidouilleur ».

¹⁹ <https://cve.mitre.org/index.html>

II. Appels système

II.1. Fonction `execve()`

Vous n'avez peut-être pas réalisé qu'il est possible de lancer un fichier exécutable depuis un autre exécutable, mais vous l'avez fait à plusieurs occasions : en lançant un de vos programmes depuis le `bash` (qui est effectivement un programme exécutable), ou dans `gdb` et `valgrind` par exemple. Avec le programme qui suit, vous allez découvrir une des fonctions qui permet de lancer depuis votre programme un autre exécutable : `execve()`²⁰. Nous pourrions alors analyser le code assembleur généré afin de le réutiliser avec un objectif de *hacking*.

```
int main(void)
{
    char *argv[] = { "/bin/sh" , NULL };

    execve( argv[0], argv, NULL );

    exit(EXIT_FAILURE);
}
```

La fonction `execve()` permet de lancer n'importe quel exécutable depuis le processus courant. Pour cela il faut lui passer en paramètres :

1. l'adresse de la chaîne de caractères comprenant le chemin complet de l'exécutable ;
2. l'adresse du tableau contenant la liste des arguments (dont le nom de l'exécutable, puis la liste complète des arguments jusqu'à la valeur `NULL`) ;
3. un troisième argument (également un tableau) qui ne nous intéresse pas ici.

Dans l'exemple ci-dessus, le programme compilé démarrera puis lancera l'exécutable `/bin/sh`, qui est un *shell*. Au final vous allez donc lancer un *shell* dans un *shell*, ce qui n'a pas grand intérêt. En revanche, ouvrir un terminal depuis un logiciel frauduleux permet au *hacker* d'accéder à une grande partie du système cible, ce qui revient à avoir les clés de la machine !

Placez-vous dans le répertoire `disco/hack/`.

Compilez le fichier `shell.c` en ajoutant les options de *debug* (`-g`). L'option `-static` permettra d'embarquer le code binaire de la fonction `execve()` dans notre exécutable afin de l'étudier.

```
gcc -g -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-
protection=none -static shell.c -o bin/shell

./bin/shell
```

Que se passe-t-il à l'exécution du programme ?

²⁰ <https://www.man7.org/linux/man-pages/man2/execve.2.html>

Analysons le désassemblage du fichier exécutable binaire.

```
objdump -S bin/shell > misc/shell_disassembly.md
```

Dans le fichier `misc/shell_disassembly.md`, cherchez la fonction `<main>` et tracez le contenu de la pile jusqu'à l'exécution de `callq execve` inclus.

Quelles sont les deux valeurs stockées dans le contexte de la fonction `<main>`? À quoi correspondent-elles dans le programme écrit en C?

Vers quelle donnée pointe la première case du tableau `argv[0]`? Confirmez avec ceci :

```
objdump -s -j .rodata bin/shell | head
```



Que contiennent les registres `rdx`, `rsi` et `rdi`?

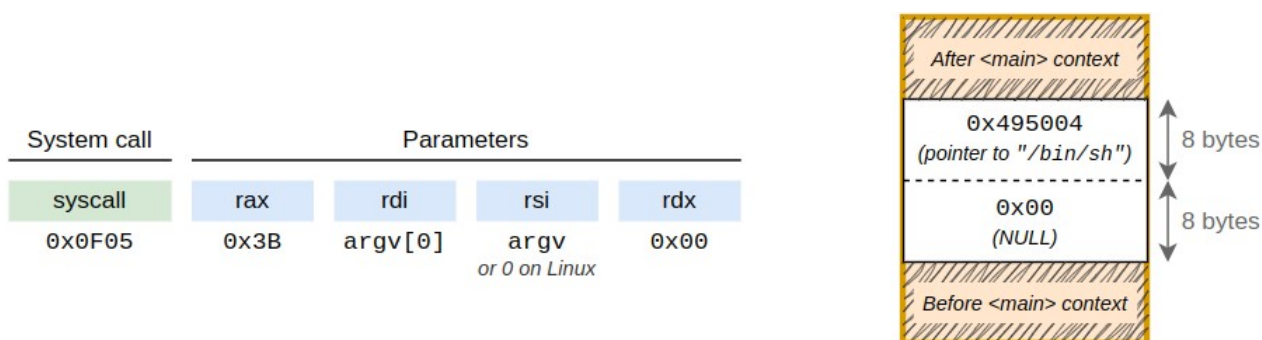
II.2. Instruction syscall

Cherchez dans le fichier `misc/shell_disassembly.md` la fonction `execve`.

Quel est le code binaire de l'instruction `syscall` ?

Quelle est la valeur du registre `rax` juste avant le `syscall` ?

L'instruction `syscall`²¹ (en x86_64) réalise un appel système : la main est alors donnée à l'OS qui effectue une action précise, définie par la valeur contenue dans le registre `rax`²². Dans notre cas, vous avez relevé que le registre `rax` contient la valeur `0x3b` (ou `59` en décimal) au moment de l'appel à `syscall`, ce qui correspond à l'appel système `sys_execve`. Avec les instructions placées dans le `main` avant l'appel à `syscall`, les registres `rdi`, `rsi` et `rdx` contiennent les arguments nécessaires à l'exécution de ce programme, comme indiqué en début de cet exercice.



Avec nos connaissances, il nous est possible d'écrire directement cet appel système (avec ses paramètres) en assembleur. Il « suffit » de placer la valeur `0x3b` dans le registre `rax`, de placer l'adresse de la chaîne de caractère `"/bin/sh"` dans le registre `rdi`, et de mettre les registres `rsi` et `rdx` à `0`. C'est le but de l'exercice page suivante.

²¹ Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C) ; p. 982.

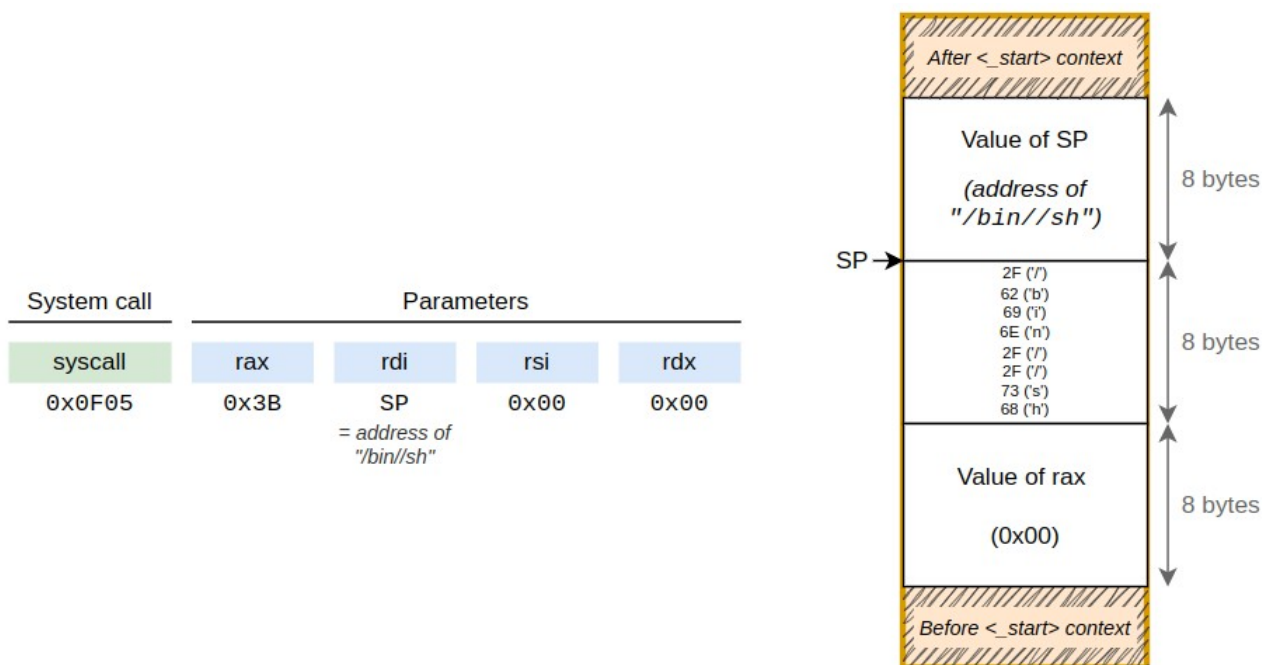
²² https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

II.3. Extraction du shellcode

Observez le contenu du fichier `shell_asm.s`.

```
_start:
    pushq    %rax
    movq     $0x68732f2f6e69622f, %rbx
    pushq    %rbx
    xorq     %rdx,%rdx
    xorq     %rsi,%rsi
    pushq    %rsp
    popq     %rdi
    movb     $0x3b, %al
    syscall
```

Vous devriez observer qu'il mène au résultat de la figure ci-dessous. Notez que cette fois-ci la chaîne de caractère `"/bin/sh"` n'est pas allouée statiquement dans la section `.rodata` de l'exécutable (contrairement au cas du fichier `shell.c`), mais qu'elle est stockée sur le segment de pile, dans le contexte de la fonction `_start`. Ainsi son adresse relative est connue puisqu'il s'agit de l'adresse désignée par le *Stack Pointer*.



Compilez et exécutez le programme pour confirmer son fonctionnement.

```
as shell_asm.s -o obj/shell_asm.o
ld obj/shell_asm.o -o bin/shell_asm
./bin/shell_asm
```

Observez l'implémentation binaire du programme et extrayez le *shellcode* (recopiez l'intégralité du code binaire du programme).

```
objdump -S obj/shell_asm.o
```

III. Exécution d'un shellcode sur la pile

L'exercice précédent nous a permis d'identifier un code binaire opératoire relativement léger (24 octets) permettant de lancer un shell à partir d'un exécutable. Dans l'esprit du *hacking*, nous chercherons à réutiliser ce **shellcode** (littéralement un *code qui lance un shell*) en exploitant une vulnérabilité du système. Ainsi le programme pourra lancer un *shell* accessible au hacker alors même que le logiciel d'origine ne le permettait pas.

Ouvrez le fichier `shellcode.c` et analysez le contenu du tableau `shellcode[]`. Que constatez-vous ? Qu'est censé faire ce programme ?

Compilez et exécutez le programme.

```
gcc shellcode.c -o bin/shellcode
./bin/shellcode
```

Le programme peut-il s'exécuter ?

Observez l'en-tête de programme du fichier ELF exécutable de sortie, et précisez les droits (`rwX`) sur le futur segment de pile qui sera associé à notre programme à l'exécution ?

```
objdump -fp bin/shellcode
```

Durant l'édition des liens, le *linker* a la capacité de préparer les droits associés aux futurs segments mémoire de l'application, notamment le segment de pile. Ce travail est réalisé à la construction du fichier ELF (en-tête du programme). Compilez à nouveau le fichier `shellcode.c` et rendant la pile exécutable, vérifiez les droits sur le segment de pile et exécutez le programme.

```
gcc -fno-stack-protector -z execstack shellcode.c -o bin/shellcode
objdump -fp bin/shellcode
./bin/shellcode
```

Le programme peut-il s'exécuter ?

Que fait le programme ?

Compilez et analysez le programme assembleur. Analysez ensuite l'exécution du programme en ouvrant une session de *debug* avec **gdb**. Décrivez le fonctionnement du programme en proposant un schéma commenté !

```
gcc -S -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o misc/shellcode.s

gcc -g -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o bin/shellcode

gdb ./bin/shellcode
(gdb) la a
(gdb) b main
(gdb) r
(gdb) s
(gdb) ni
...
(gdb) ni
$                               ← Ici, le shellcode est en cours d'exécution
$ exit
(gdb) q
```

Après analyse du script assembleur, représentez ci-contre le contenu de la pile avant exécution de l'instruction `call %rdx` implémentant `return(0)` de la fonction `main`. S'aider des outils et des commandes suivantes.

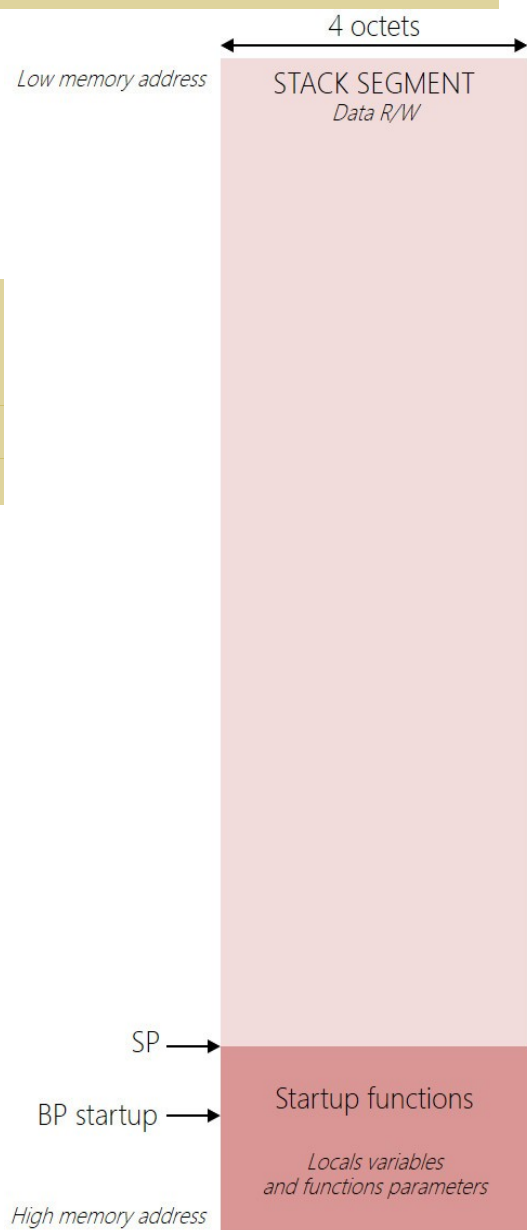
```
gcc -c -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o obj/shellcode.o

objdump -S ./obj/shellcode.o

objdump -s ./obj/shellcode.o
```

À ce stade, vous avez réussi à récupérer le code binaire d'un programme permettant de lancer un *shellcode* (premier exercice). De nombreux exemples de *shellcodes* sont disponibles sur Internet²³. Puis vous avez stocké le code binaire de ce *shellcode* sur la pile de sorte à l'exécuter par la suite.

Avec l'exercice suivant, vous verrez comment insérer ce *shellcode* dans un programme existant afin de le détourner de son fonctionnement normal.



²³ <http://shell-storm.org/shellcode/>

IV. Exploitation de vulnérabilité

Nous allons dans cette dernière partie donner quelques briques et techniques permettant l'ouverture au développement d'un *exploit* (exploitation d'une vulnérabilité). Une technique classique consiste à remplacer l'adresse de retour d'une fonction par celle d'un code malveillant (*shellcode*) caché sur la pile. Rappelons que l'adresse de retour d'une fonction est sauvée par défaut sur la pile durant l'appel de la fonction (*call*). Ainsi, lorsque la fonction courante souhaitera se terminer en exécutant l'instruction *ret*, qui dépile l'adresse de retour de la fonction appelante depuis la pile pour la placer dans le registre CPU d'instruction *rip*, la fonction la remplacera par l'adresse du code malveillant à exécuter. Voilà ci-dessous une écriture en pseudo-code des instructions *call* et *ret*.

CALL <i>function_address</i>	<=>	$RSP \leftarrow RSP - 8$ $*(RSP) \leftarrow RIP$ $RIP \leftarrow function_address$
RET	<=>	$tmp \leftarrow *(RSP)$ $RSP \leftarrow RSP + 8$ $RIP \leftarrow tmp$

Ouvrez le fichier *disco/hack/exploit.c* et analysez-le. Pour l'instant on considère qu'il s'agit d'une application classique, dans laquelle on a ajouté le *shellcode*. Compilez et exécutez le fichier.

```
gcc -fno-stack-protector -z execstack exploit.c -o bin/exploit
./bin/exploit
```

A-t-on accès à un shell (autrement dit, le *shellcode* s'exécute-t-il) ? Pourquoi ?

Pour que le *shellcode* s'exécute, il faut qu'à un certain moment le *Instruction Pointer* pointe vers celui-ci. Pour cela, nous allons falsifier dans la pile l'adresse de retour de la fonction ayant appelé le *main()*. L'instruction *ret* se chargera d'affecter l'adresse frauduleuse dans le *Instruction Pointer*.

Modifiez le premier commentaire *TODO* dans le fichier *disco/hack/exploit.c* par une ligne de code permettant de remplacer l'adresse de retour de la fonction appelante par l'adresse du *shellcode* sur la pile. Compilez et validez la bonne exécution du programme.

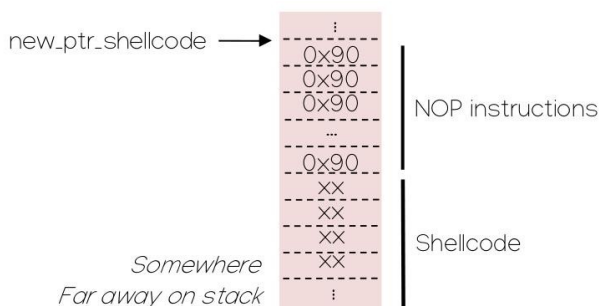
Avez-vous désormais accès à un shell ? Si oui, mission accomplie.

Différents mécanismes de sécurité (logiciels ou matériels) peuvent mettre à mal l'exécution de *shellcodes*, car ceux-ci sont (plutôt) connus et sont donc détectables. Pour éviter qu'un *exploit* soit détecté il existe différentes techniques de camouflage :

- *obfuscation* : le *shellcode* est encodé ou chiffré avant d'être inséré dans le firmware, mais il faut également intégrer une routine de décodage ou déchiffrement ;
- *padding, garbage insertion* : ajout d'instructions **NOP** (*No Operation*) ou de code mort (code sans impact) dans le *shellcode* afin de brouiller la lisibilité de celui-ci
- ...

La méthode que nous allons implémenter ici consiste à insérer le *shellcode* sur la pile, mais suffisamment loin du bas de pile. Les outils automatiques ne font généralement qu'analyser le code et les données proches des informations déjà présentes, ce qui permet d'éviter au *shellcode* d'être détecté. Par contre, des outils d'analyse heuristique pourront remarquer un accès soudain à une zone mémoire encore jamais utilisée, ce qui peut paraître suspect et provoquer une analyse.

Modifiez le second commentaire **TODO** afin de déplacer le *shellcode* plus loin sur la pile. Nous placerons des instructions **NOP** (*opcode* binaire **0x90**) avant le *shellcode*. Cette technique est couramment utilisée afin de faciliter la recherche d'un *shellcode* sur la pile par un *exploit*. Notamment lorsque l'adresse exacte du code malveillant n'est pas précisément connue.



Compilez et testez la solution. Si vous avez accès au shell, alors mission accomplie : vous venez d'insérer un code malveillant dans un code source, tout en le camouflant un minimum \o/

*Note : Les solutions sont cachées dans le répertoire **hack/misc/** !*

V. White-hat

Cet exercice de travaux pratiques rassemble un tas de compétences, dont toutes les notions importantes attendues dans cet enseignement. Si vous avez pu réaliser cet exercice par vos propres moyens, alors vous pouvez être fier.

Vous pouvez aller encore plus loin en cherchant à développer un exploit permettant d'ouvrir une console *root* sur votre machine personnelle (à partir d'un programme *non-root*). Si vous y arrivez, chapeau ! Auquel cas votre exploit pourra être intégré dans cette trame de TP pour les futurs élèves (avec tout le crédit qui va avec, et peut-être même des cafés jusqu'à la fin d'année) !

Notamment depuis l'arrivée d'internet, les systèmes numériques d'information se sont grandement complexifiés, mais également durcis. Bien des vulnérabilités ont déjà été explorées et exploitées, et les solutions déjà déployées. Mais comme une recherche inextinguible de trésor, bien des failles restent encore à trouver. Que ce soit au niveau matériel (étage de prédiction avec les failles *Meltdown* et *Spectre*, *NX bit* ou *No eXecute bit* dans la table de translation d'adresses utilisée par la MMU, ...) ou au niveau *kernel* Linux avec par exemple une gestion aléatoire des *mapping* mémoire de certains segments applicatifs (pile, tas, bibliothèques partagées et vDSO, ...), plusieurs contre-mesures sont déjà à l'œuvre afin de contraindre le travail des *Black Hats* dans leur volonté de pénétrer les systèmes.

Prenons l'exemple de la fonction `execve`. Comme précisé dans la documentation officielle²⁴, cette fonction travaille par remplacement de segments mémoire de la fonction appelante (`.text`, `.bss`, `.data` et pile). De même, il existe un jeu de conditions possibles et documentées permettant à l'appelant d'hériter des privilèges d'exécution de l'appelé²⁵. Je vous laisse donc mûrir et entrouvrir le spectre du possible, si l'applicatif appelé et le système de fichiers sous-jacent n'a pas été pensé contre !

Et pour les bouineurs et bouineuses, le site RootMe²⁶ donne accès à de nombreux exercices de *hacking*. Les défis relevés sur ce site font même l'objet d'un portfolio qui peut être analysé dans le cadre d'entretiens d'embauches !

²⁴ <https://www.man7.org/linux/man-pages/man2/execve.2.html>

²⁵ <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=fs%2Fexec.c>

²⁶ <https://www.root-me.org/>

