



Introduction à l'exploitation de binaires (pwn)

Fièrement offert par mouthon

Cycle de conférences: pwn

- ??/??/2024: Introduction, stack based (mouthon)
- ??/??/2024: Heap exploitation (voydstack)
- ??/??/2024: Linux kernel (Dvorhack)



RootMe
- Hacking platform -

~~flexing~~ whoami

- Root-Me : QA depuis 2023
- Double diplôme X / TU Munich
- Thèse de Master : Rowhammer et obfuscation
- CTF @ h4tum & ECSC Team France



Mathéo Vergnolle (mouthon)

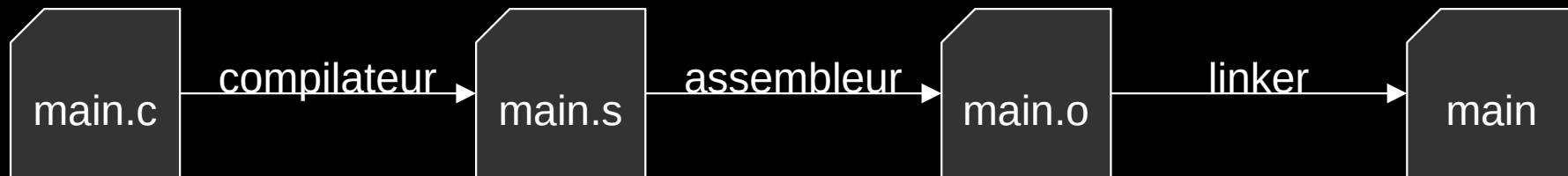
Motivation

- Transformer les segfaults en shell
- RCE (service remote), LPE (kernel, binaire suid)
- Problèmes intéressants et stimulants
- Avoir des CVE plus classes que le web ou ne pas avoir de CVE du tout
- Briller en société parce qu'on est un h4ck3r

Au programme

- Pré-requis : savoir programmer en C, bases en assembleur x86_64 et en reverse
- pwn **userland**, sous **Linux** (ELF), arch **x86_64**
- Exploitation de dépassement de tampon sur la stack (“stack buffer overflow”), de 1996 au ROP
- Concepts généraux et **mise en pratique**

X86_64 ASM 101



- Registres = variables de l'assembleur
- Paramètres des fonctions/syscalls dans `rdi`, `rsi`, `rdx`, `rcx/r10`, `r8`, `r9`, puis pile
- Retour des fonctions/syscalls et numéro de syscall dans `rax`
- `rip` = instruction pointer, `rsp` = stack pointer
- Exemple

Organisation de la mémoire

Code (.text) →
Données (.rodata, .data, .bss) {
Tas ("heap") →

Memory mapping region

Pile ("stack") →

```
cat /proc/self/maps
55ec49157000-55ec49159000 r--p 00000000 fe:00 4981162 /usr/bin/cat
55ec49159000-55ec4915d000 r-xp 00002000 fe:00 4981162 /usr/bin/cat
55ec4915d000-55ec4915f000 r--p 00006000 fe:00 4981162 /usr/bin/cat
55ec4915f000-55ec49160000 r--p 00007000 fe:00 4981162 /usr/bin/cat
55ec49160000-55ec49161000 rw-p 00008000 fe:00 4981162 /usr/bin/cat
55ec4a524000-55ec4a545000 rw-p 00000000 00:00 0 [heap]
7f3d5ec00000-7f3d5eec0000 r--p 00000000 fe:00 4995412 /usr/lib/locale/locale-archive
7f3d5f063000-7f3d5f066000 rw-p 00000000 00:00 0
7f3d5f066000-7f3d5f08a000 r--p 00000000 fe:00 4992832 /usr/lib/libc.so.6
7f3d5f08a000-7f3d5f1f6000 r-xp 00024000 fe:00 4992832 /usr/lib/libc.so.6
7f3d5f1f6000-7f3d5f244000 r--p 00190000 fe:00 4992832 /usr/lib/libc.so.6
7f3d5f244000-7f3d5f248000 r--p 001dd000 fe:00 4992832 /usr/lib/libc.so.6
7f3d5f248000-7f3d5f24a000 rw-p 001e1000 fe:00 4992832 /usr/lib/libc.so.6
7f3d5f24a000-7f3d5f254000 rw-p 00000000 00:00 0
7f3d5f25f000-7f3d5f2a1000 rw-p 00000000 00:00 0
7f3d5f2a1000-7f3d5f2a2000 r--p 00000000 fe:00 4992107 /usr/lib/ld-linux-x86-64.so.2
7f3d5f2a2000-7f3d5f2c9000 r-xp 00001000 fe:00 4992107 /usr/lib/ld-linux-x86-64.so.2
7f3d5f2c9000-7f3d5f2d3000 r--p 00028000 fe:00 4992107 /usr/lib/ld-linux-x86-64.so.2
7f3d5f2d3000-7f3d5f2d5000 r--p 00032000 fe:00 4992107 /usr/lib/ld-linux-x86-64.so.2
7f3d5f2d5000-7f3d5f2d7000 rw-p 00034000 fe:00 4992107 /usr/lib/ld-linux-x86-64.so.2
7ffeb4523000-7ffeb4545000 rw-p 00000000 00:00 0 [stack]
7ffeb455a000-7ffeb455e000 r--p 00000000 00:00 0 [vvar]
7ffeb455e000-7ffeb4560000 r-xp 00000000 00:00 0 [vdso]
fffffffff6000000-fffffffff6010000 --xp 00000000 00:00 0 [vsyscall]
```

La pile ("stack")

```
1  int truc(int x) {  
2      int y = 8;  
3      return x + y;  
4  }  
5  
6  int bar() {  
7      int x = 3;  
8      int y = 5;  
9      y = truc(x);  
10     return y;  
11 }  
12  
13 void foo() {  
14     truc(x: 2);  
15     bar();  
16 }  
17  
18 int main() {  
19     int result = bar();  
20     return result;  
21 }
```

Adresses
basses

Adresses
hautes

Stack frame de <i>truc</i>	<i>y</i>
	Ret addr de <i>truc</i>
Stack frame de <i>bar</i>	<i>y</i>
	<i>x</i>
	Ret addr de <i>bar</i>
Stack frame de <i>main</i>	<i>result</i>
	Ret addr de <i>main</i>

Pwning like 1996

.o0 Phrack 49 0o.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack' [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind.

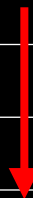
```
void foo(unsigned int size) {  
    char buf[16];  
    read(fd: fileno(stream: stdin), buf, nbytes: size);  
    puts(s: buf);  
    if (buf[0] == '2') {  
        read(fd: fileno(stream: stdin), buf, nbytes: size);  
        puts(s: buf);  
    }  
}
```

buf[0:8]

buf[8:16]

...

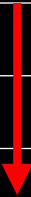
Return address



ret2shellcode

- On écrit du code machine sur la stack, puis on saute dessus
- Plus d'infos:
https://www.youtube.com/watch?v=zxhQevqX_7w
(conf de Voydstack sur le shellcoding)

Stack canary

buf[0:8]	
buf[8:16]	
Canary (?????)	
Return address	

- Random, différent à chaque run
- Commence par 0x00
- Valeur vérifiée à la fin de la fonction
- → on doit le leak

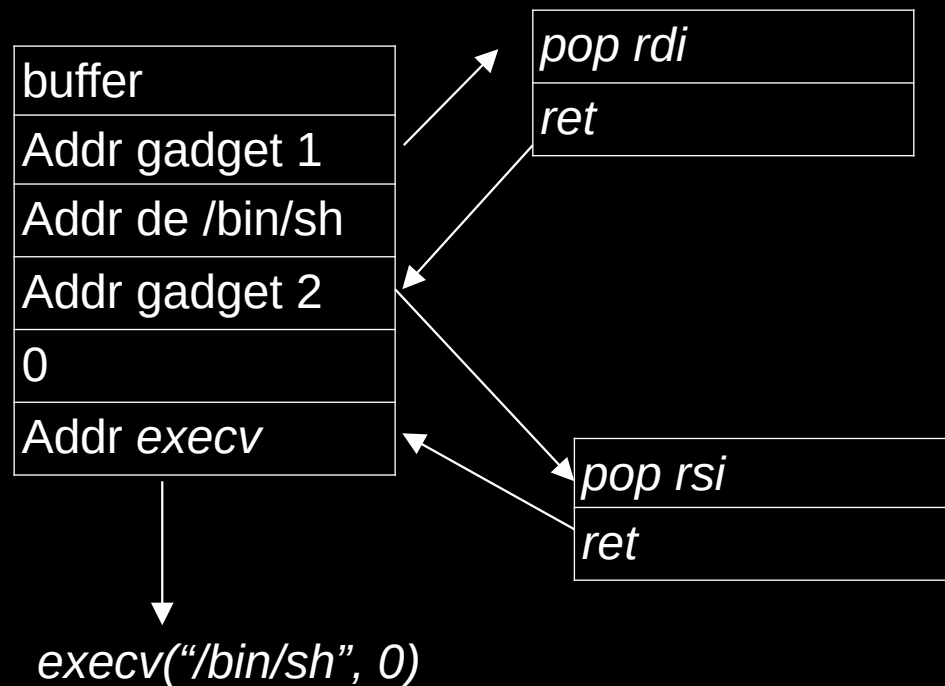


W^X et ASLR

- W^X / DEP : la mémoire est écrivable ou exécutable, mais pas les deux
 - On ne peut donc pas injecter notre propre code, doit réutiliser du code existant (ex: one gadget)
- ASLR (+ PIE) : les adresses des blocs de mémoire sont aléatoires
 - On doit leak les adresses, ou utiliser des adresses relatives

Return-Oriented Programming

- “gadgets” : des petits bouts de code à la fin des fonctions qu’on peut chaîner
- Turing complete



Et maintenant ?

- Protections plus avancées: shadow stack, CFI/CET, canaries plus sophistiqués (cf. OpenBSD)
- Attaques plus avancées: JOP, SROP, ret2dl_resolve, blind ROP, PIROP...
- <https://www.root-me.org/fr/Challenges/App-Systeme/>
- Questions ?