Buffer Overflow Vulnerabilities, Attacks, and Defenses

Software Security

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

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

- ***Reminder:*** Lab 02 due Thursday (2/6) @ 3pm
- ***Question:*** Buffer Overflow CTF ***or*** Format String Vulns.
- **REU** (Research Experiences for Undergraduates)
  - **@ MSU:** Cybersecurity Algorithms - http://www.bobcatsoftwarefactory.com/nsf-reu-2020
  - **@ NYU:** HW and SW Security for Mobile Devices and Networks (see email sent earlier today)

Goals & Learning Objectives

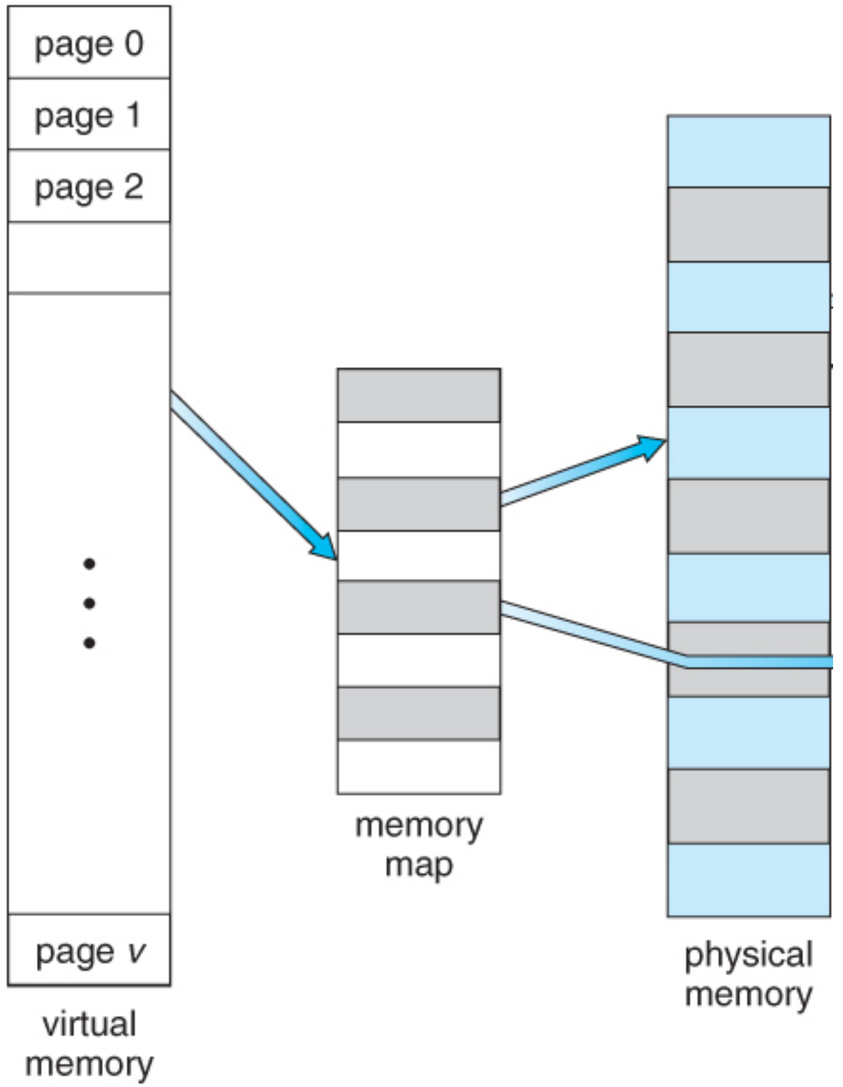
- Buffer Overflow Vulnerabilities, Attacks, and Defenses
  - Review the layout of a program in memory
  - Understanding the stack layout
  - Vulnerable code
  - Challenges in exploiting buffer overflow vulnerabilities
  - Countermeasures

Foundations: Memory Layout, Stacks, and Function Invocation

Virtual Addresses vs. Physical Addresses

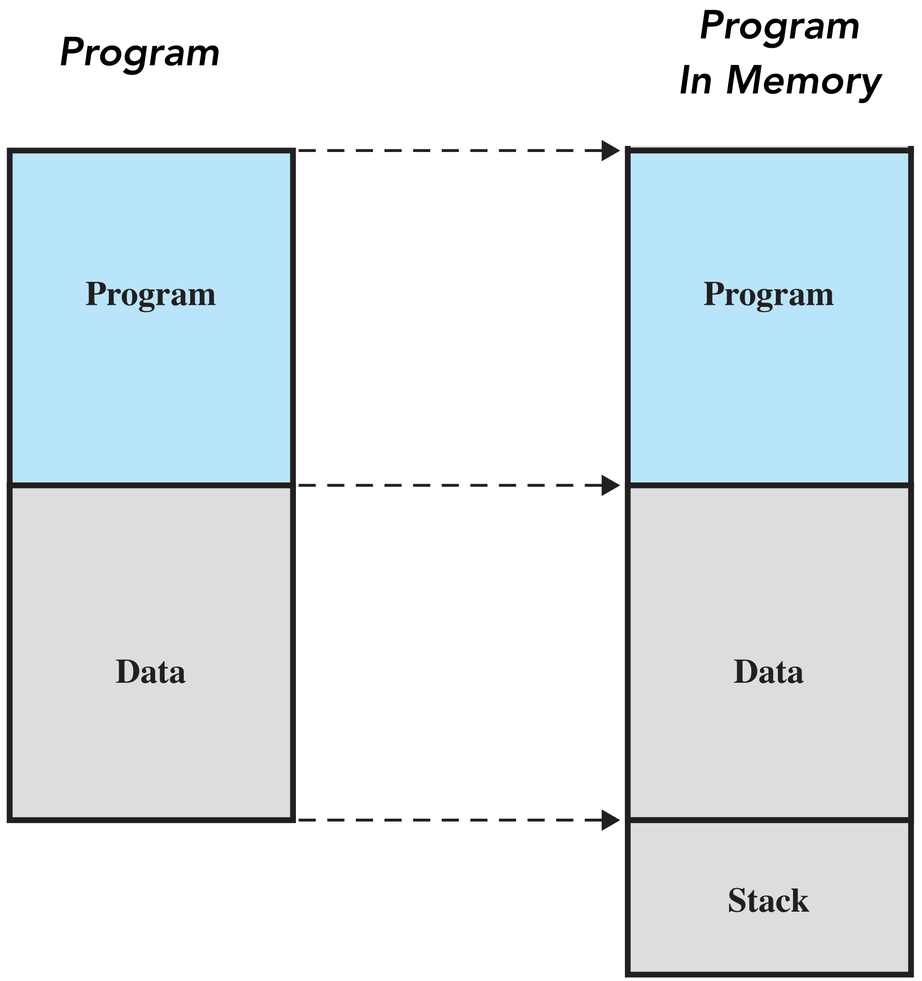
- Programs do not access physical memory directly
- Process memory is mapped from a virtual address space to the physical address space
- ➡ ***Virtual Memory***
- OS maintains a table (memory map) for each process

Why?



Loading a Program Into Memory

A standard layout for programs makes ***loading*** and ***managing*** programs easy!

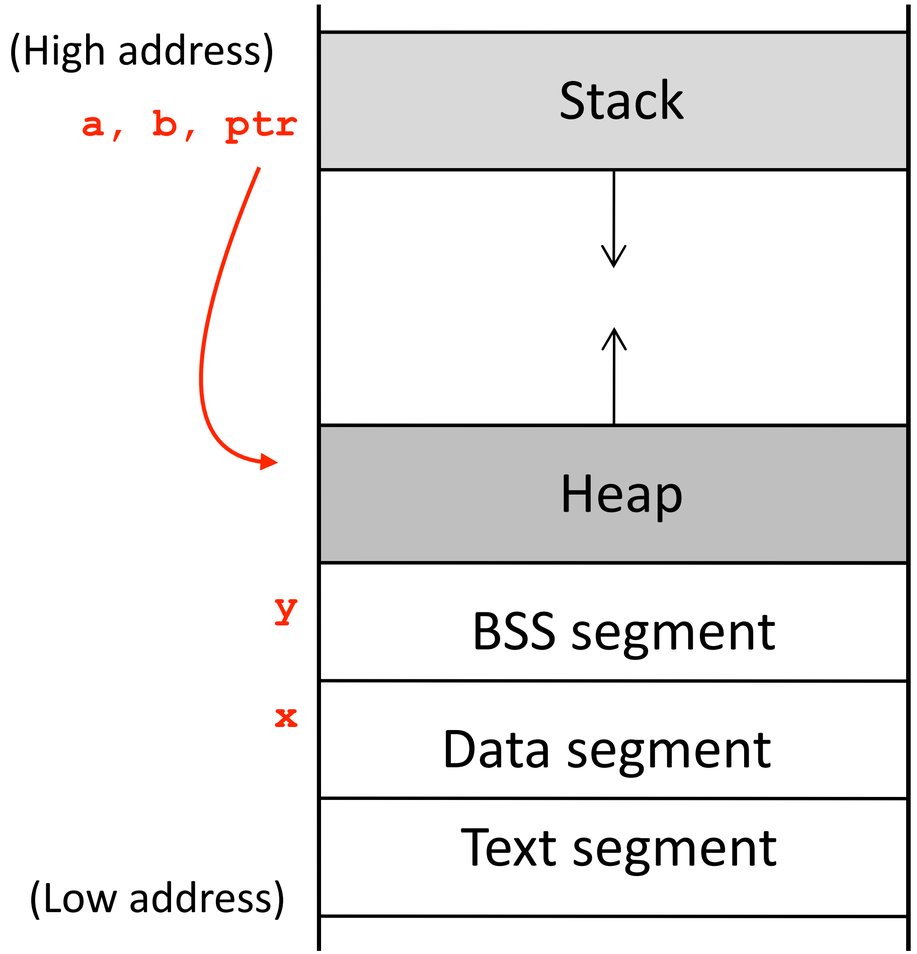


The Memory Layout of a Running Program

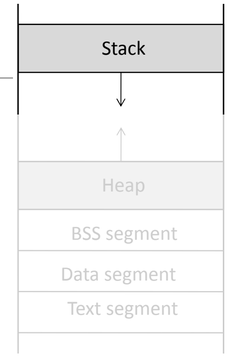
Activity!

Where in memory do the variables go for this program?!

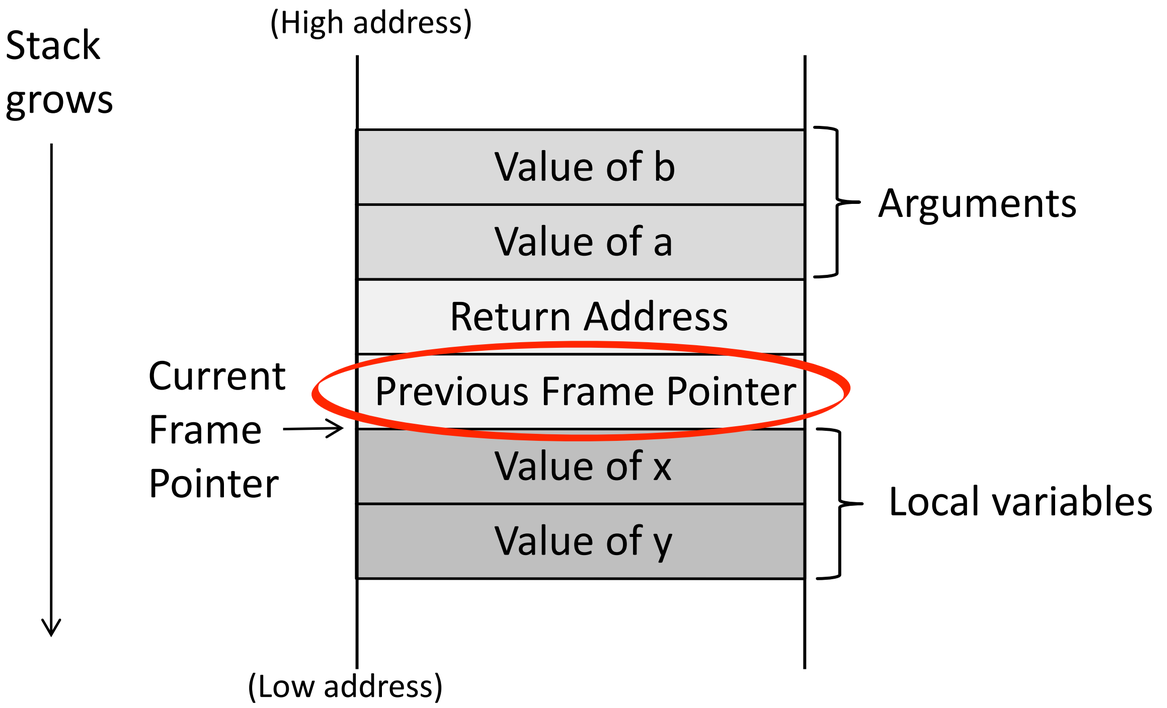
```
int x = 100;
int main()
{
   int a = 2;
   float b = 2.5;
   static int y;

   int *ptr = (int *) malloc(2*sizeof(int));
   ptr[0] = 5;
   ptr[1] = 6;

   free(ptr);
   return 1;
}
```



Stack Layout: Function Arguments, Local Variables, …




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

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



Solution: Frame Pointer (ebp)!

```
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
```

PROBLEM!

The compiler doesn't know where the stack frame will be in memory…

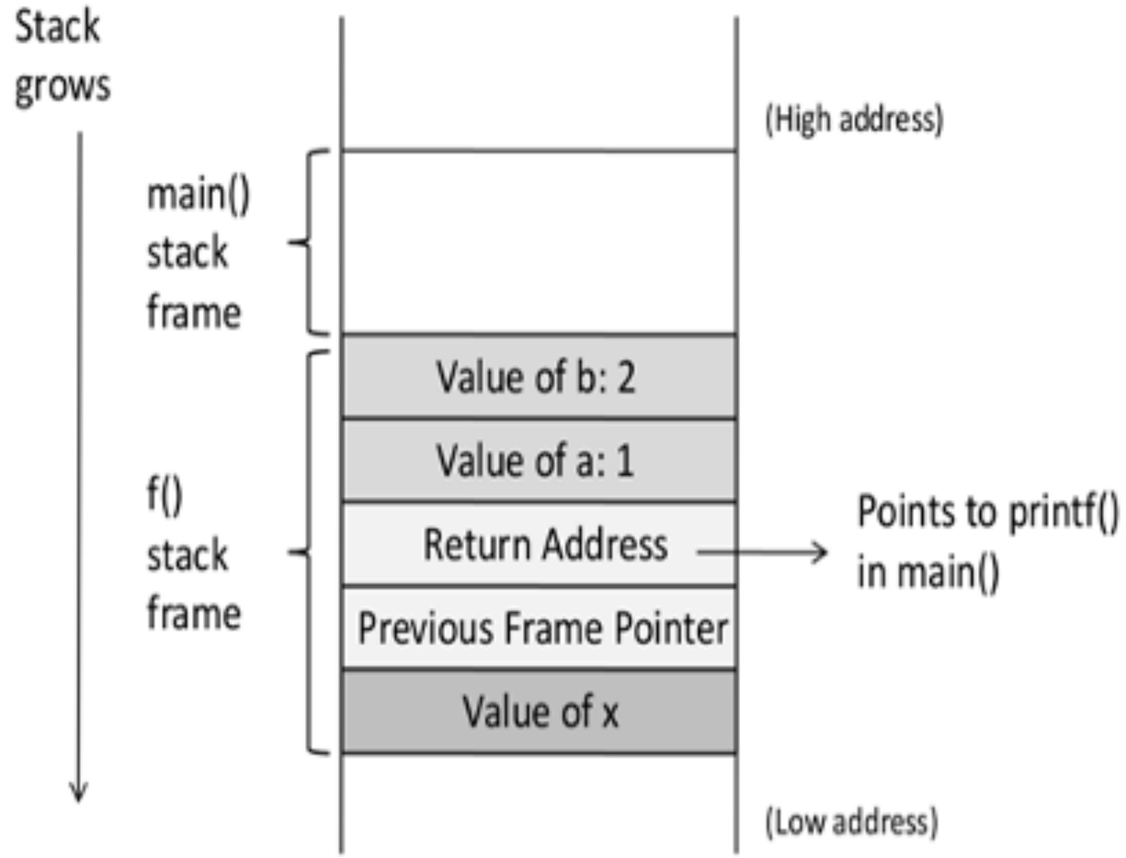
How do we know the address of function arguments/local variables?

Example: A Function Call Stack

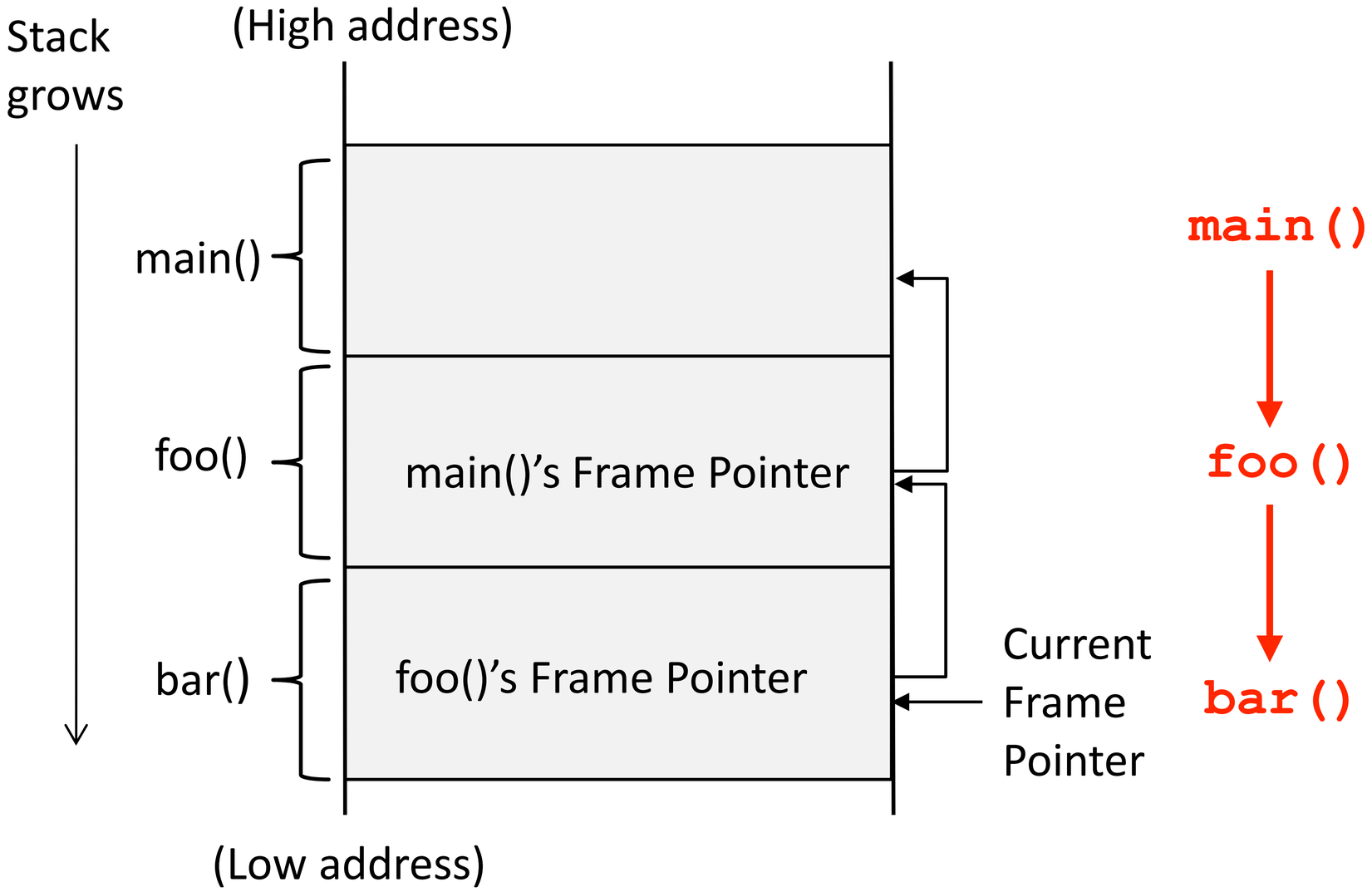
Activity!

What will foo's stack frame look like?

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

void main()
{
  f(1,2);
  printf("hello world");
}
```



Stack Layout and a Function Call Chain



Note

Frame pointer (ebp) points to current frame; ebp is different for each stack frame. Need to save previous frame pointer

Now, let's consider an example vulnerable program…

A Vulnerable Program

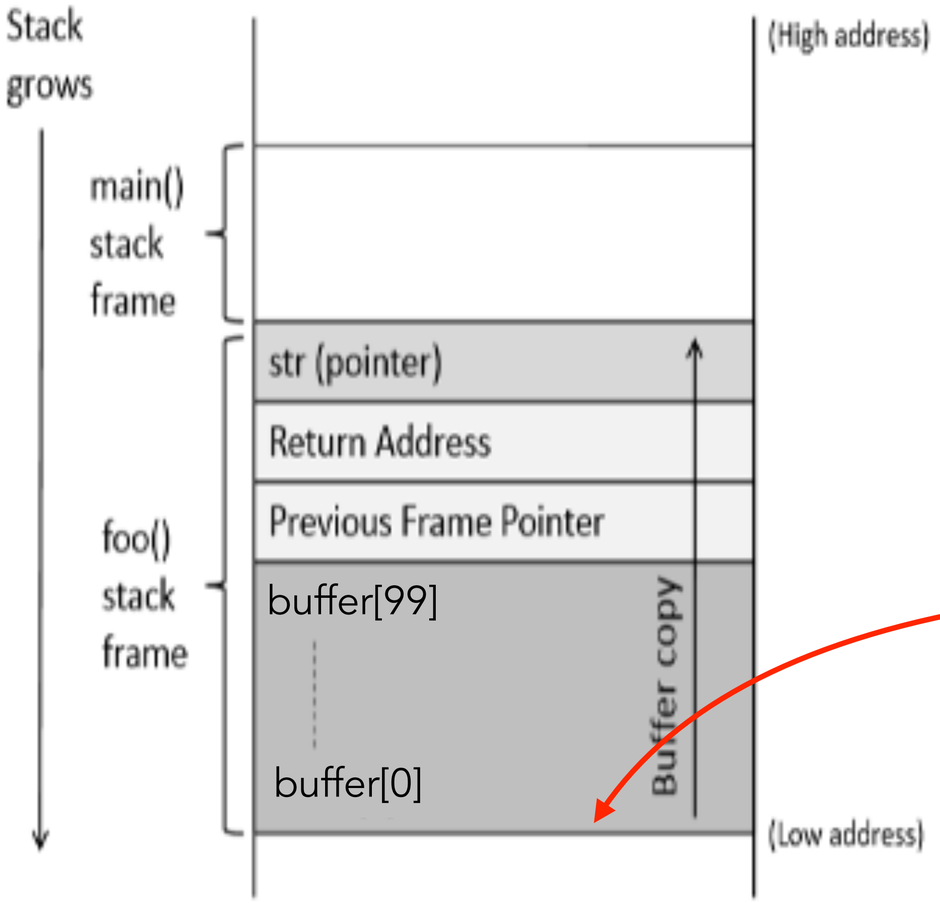
- Reading 300 bytes of data from `badfile`.
- Storing the file contents into a `str` variable of size 400 bytes.
- Calling `foo` function with `str` as an argument.

Note: *The `badfile` is created by the user and hence the contents are controlled by the user.*

Call chain: `main()` -> `foo()` -> `strcpy()`

```
/* 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;
}
```

A Vulnerable Program *(cont.)*



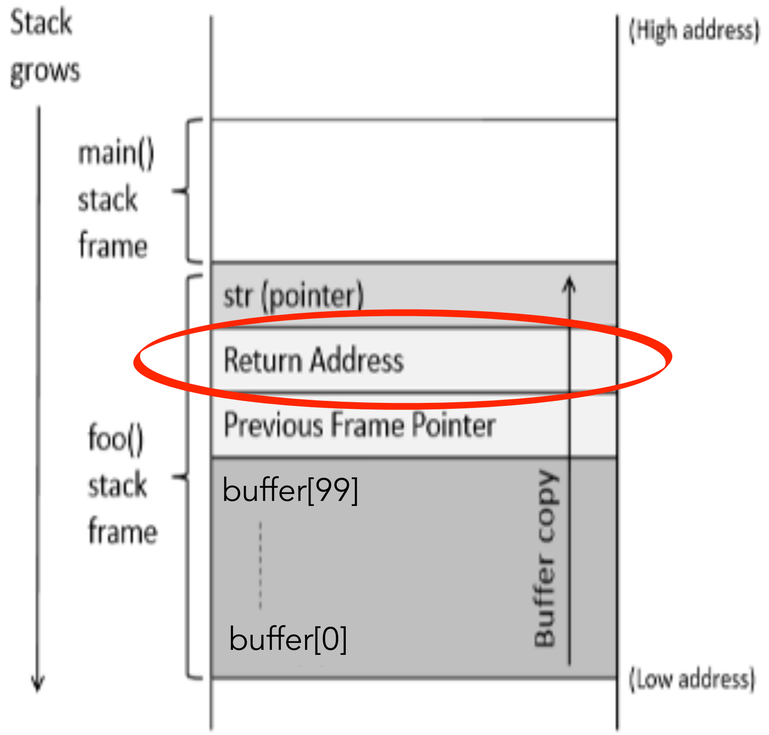

```c
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

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

    return 1;
}
```

Consequences of a Buffer Overflow

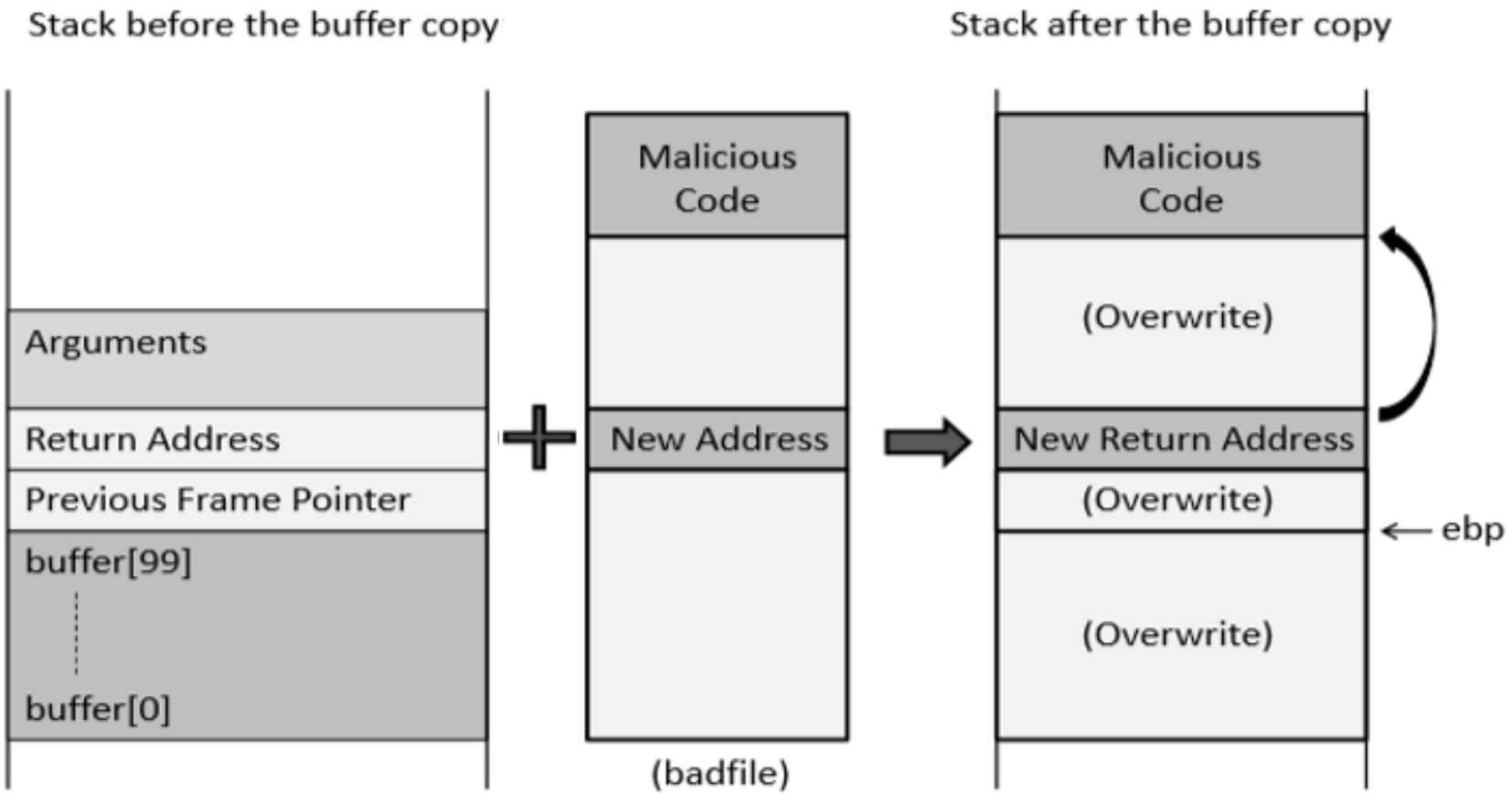
What could really go wrong with a buffer overflow…?

Overwriting the ***return address*** with some bad data (random address) can lead to:

- Non-existent address *>>> access unmapped page;* ***crash!***
- Access violation *>>> access mapped (protected) page (e.g., kernel memory);* ***crash!***
- Invalid instruction *>>> contents of return address is not a valid instruction;* ***crash!***
- ***Execution of attacker's code!***
  >>> contents of return address is a valid instruction; program will continue running w/ different logic!



How to Run Malicious Code



OK, so HOW do we carry out the buffer overflow attack?!?!?!!?!??!

First… Environment Setup

- Turn off **address randomization** (countermeasure)

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

- Compile a set-uid root version of `stack.c` with **executable stack permitted** + **no stack guard**

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

Don't worry, we will look at these countermeasures later, and look at how these can be defeated……

Creating the Malicious Input (`badfile`)

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