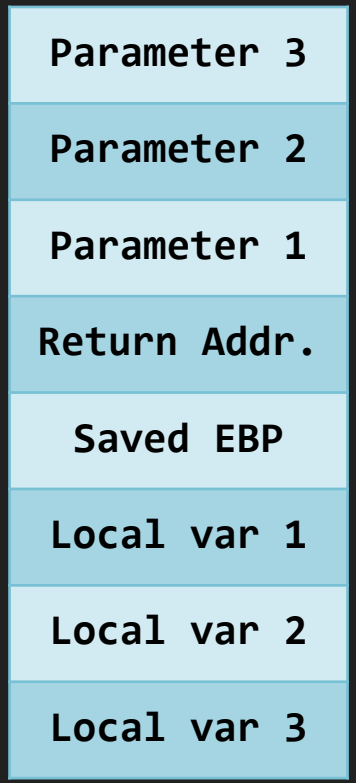
Buffer Overflows, ret2libc, ROP

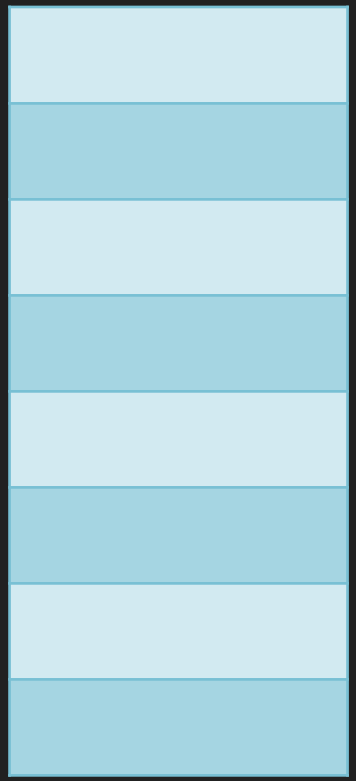
MST - 2/9

Review: the stack in C



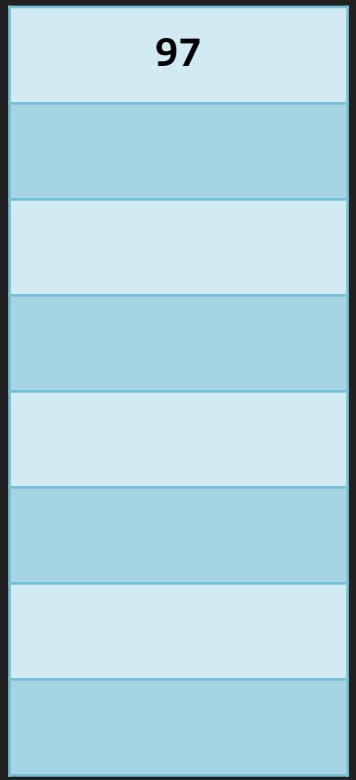

```
int tripsum(int a, int b, int c) {
    int x = a + 1;
    x += b + 1;
    x += c + 1;
    return x;
}

// <later in the program>
tripsum(3, 15, 97);
```

Review: the stack in C



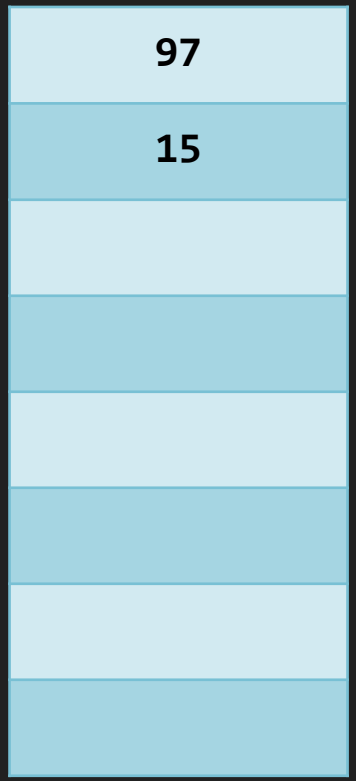
```
push 97 <---
push 15
push 3
call tripsum
add esp, 12
```

Review: the stack in C




```
push 97
push 15 <---
push 3
call tripsum
add esp, 12
```

Review: the stack in C

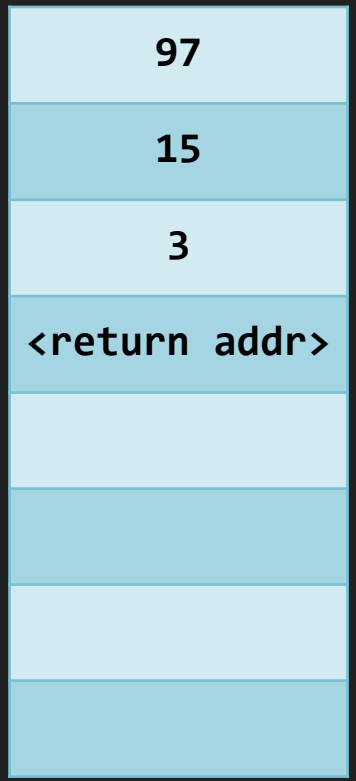



```
push 97
push 15
push 3 <---
call tripsum
add esp, 12
```

Review: the stack in C

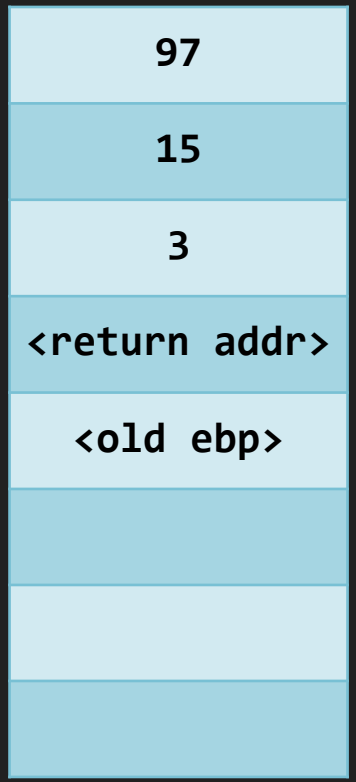

```
push 97
push 15
push 3
call tripsum <---
add esp, 12
```

Review: the stack in C



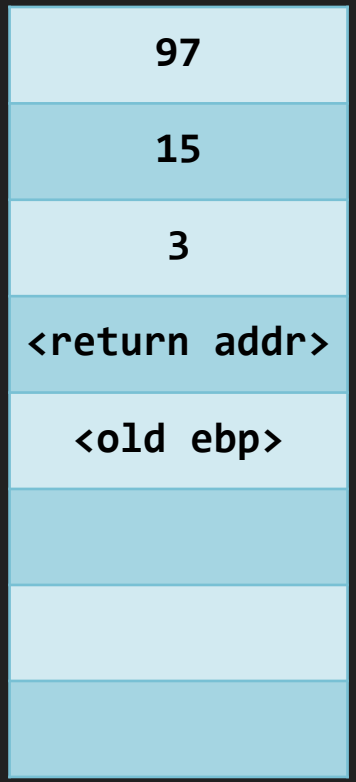

```
push ebp <---
mov ebp, esp
sub esp, 4
; actual operations here
mov esp, ebp
pop ebp
ret
```

Review: the stack in C



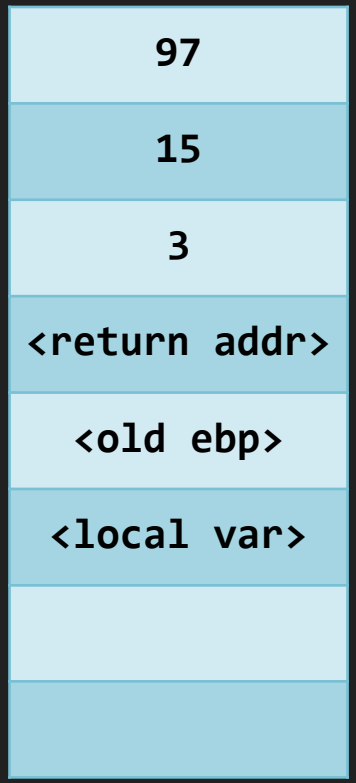

```
push ebp
mov ebp, esp <---
sub esp, 4
; actual operations here
mov esp, ebp
pop ebp
ret
```

Review: the stack in C



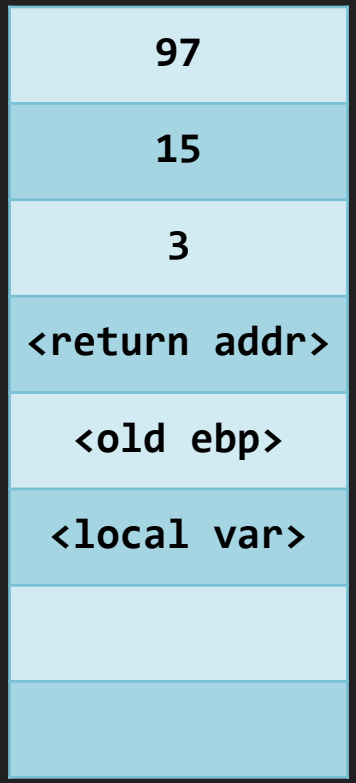

```
push ebp
mov ebp, esp
sub esp, 4 <---
; actual operations here
mov esp, ebp
pop ebp
ret
```

Review: the stack in C



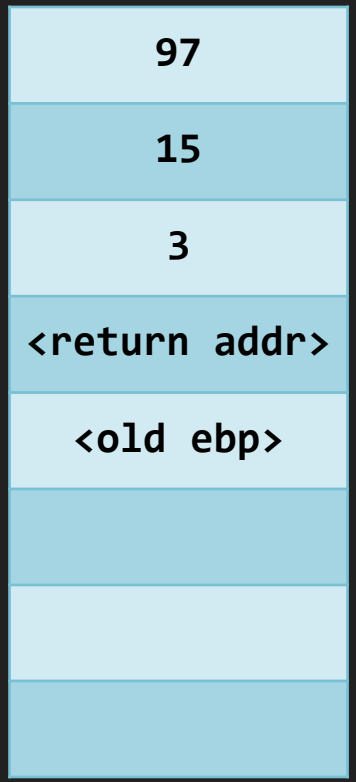

```
push ebp
mov ebp, esp
sub esp, 4
; actual operations here <---
mov esp, ebp
pop ebp
ret
```

Review: the stack in C



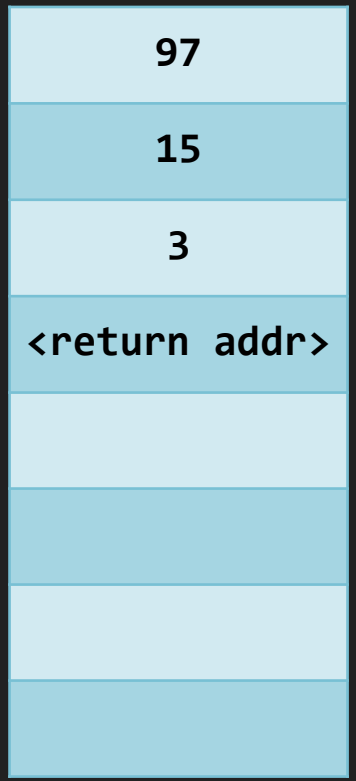

```
push ebp
mov ebp, esp
sub esp, 4
; actual operations here
mov esp, ebp <---
pop ebp
ret
```

Review: the stack in C




```
push ebp
mov ebp, esp
sub esp, 4
; actual operations here
mov esp, ebp
pop ebp <---
ret
```

Review: the stack in C

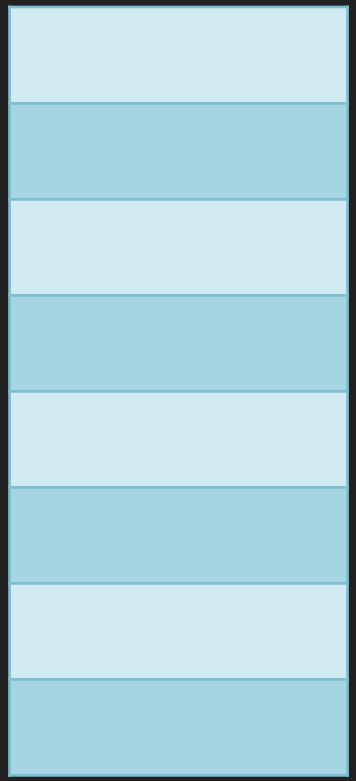



```
push ebp
mov ebp, esp
sub esp, 4
; actual operations here
mov esp, ebp
pop ebp
ret <---
```

Review: the stack in C


```
push 97
push 15
push 3
call tripsum
add esp, 12 <---
```

Review: the stack in C

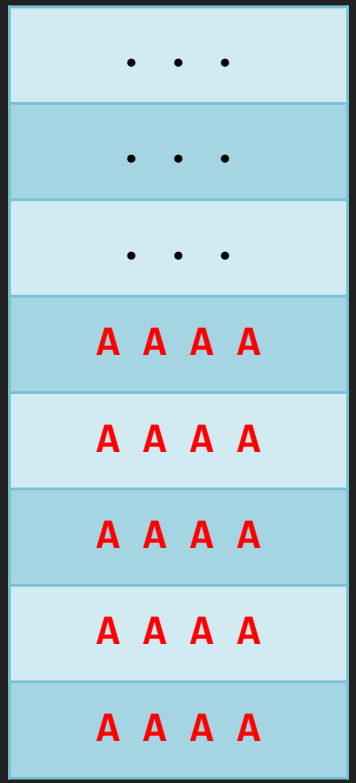


```
push 97
push 15
push 3
call tripsum
add esp, 12
```

What happens if we write too much data?


```
int main() {
   char buf[12];
   printf("Your name: ");
   gets(buf);
   printf("Hello, %s\n", buf);
   return 0;
}

$ ./my_name Cyrus
```

What happens if we write too much data?




```
int main() {
   char buf[12];
   printf("Your name: ");
   gets(buf);
   printf("Hello, %s\n", buf);
   return 0;
}

$ ./my_name AAAAAAAAAAAAA...
```

Memory Corruption

- Data is manipulated by user input in an unintended way
- Important constructs can be overwritten
- Usually leads to crashes… (SEGFAULT, anyone?)
- …but we can use it for our purposes!
- Basis for pretty much all low-level vulnerabilities we'll look at

What can we do with a buffer overflow?

- We control the **return address**, so we control **execution flow**
- What should we set it to?
  - **Shellcode** that we have written
  - Existing functions
  - Something else?

Shellcode

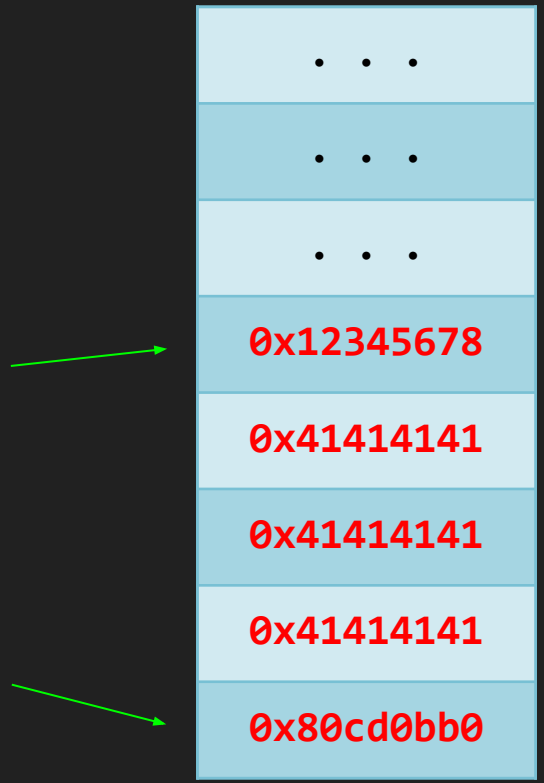
- Short payload code that gives us control of the program or system)
  - Launch a shell
  - Open a port
  - Give us critical system data
  - You name it!

```
xor    %eax, %eax
push   %eax
push   $0x68732f2f ; “hs//”
push   $0x6e69622f ; “nib/”
mov    %esp, %ebx
push   %eax
push   %ebx
mov    %esp, %ecx
mov    $0xb, %al
int    $0x80

"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x
2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\
xb0\x0b\xcd\x80"
```

What can we write?

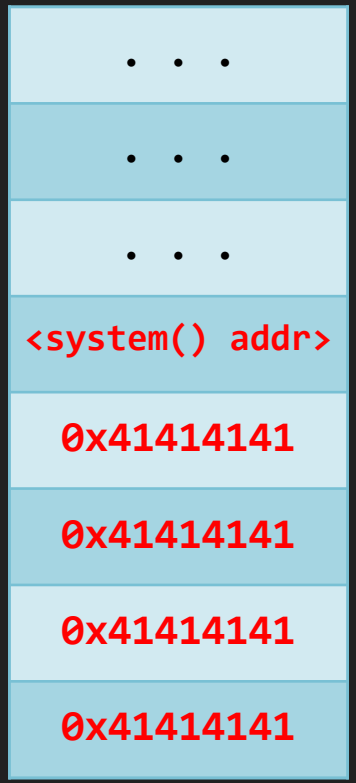
1. Write our shellcode to the buffer
2. Pad it with “A”s so it’s long enough
3. Replace the return address with the start of the buffer
4. Execute our shellcode!

What can we write?



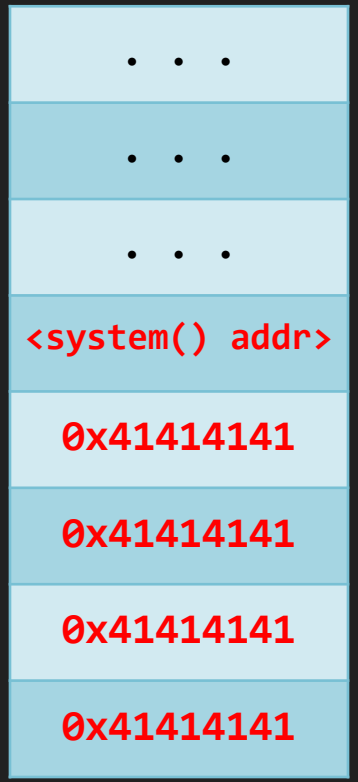
1. Write our shellcode to the buffer
2. Pad it with “A”s so it’s long enough
3. Replace the return address with the start of the buffer
4. Execute our shellcode!

“Return to libc” (ret2libc)

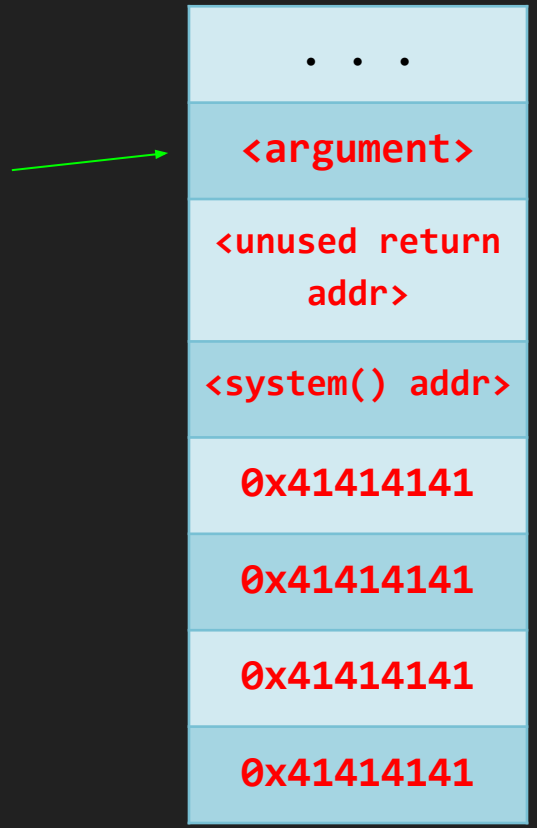


- We can replace the return address with **any function**, including library functions!
- What if we put the address of `system()`?

Calling `system()` with arguments



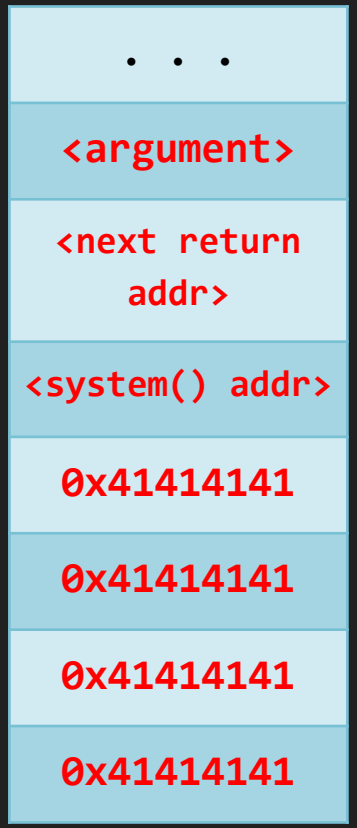
Calling `system()` with arguments



Where else can we jump?

- We can jump to **any valid address** and continue execution
- **Return Oriented Programming** - jumping to arbitrary locations to properly execute code
- We put multiple return addresses on the stack by creating fake stack frames
- **ROP Gadgets** are little snippets of instructions that we can use to build up a larger **ROP chain**

ROP & Chaining Functions



- We need a ROP gadget to create the stack frame successfully
- Recall how stack frames are laid out - what makes it difficult to create several in a row?

ROP Gadget

```
80483d9:        89 e5                   mov    %esp,%ebp
80483db:        83 ec 14                sub    $0x14,%esp
80483de:        68 24 a0 04 08          push   $0x804a024
80483e3:        ff d0                   call   *%eax
80483e5:        83 c4 10                add    $0x10,%esp
80483e8:        c9                      leave
80483e9:        f3 c3                   repz ret
```

ROP Gadget

```
80483d9:        89 e5                   mov    %esp,%ebp
80483db:        83 ec 14                sub    $0x14,%esp
80483de:        68 24 a0 04 08          push   $0x804a024
80483e3:        ff d0                   call   *%eax
80483e5:        83 c4 10                add    $0x10,%esp
80483e8:        c9                      leave
80483e9:        f3 c3                   repz ret
```

Full ROP example

Homework

Homework

- Exploit the given program using a ROP attack!
- Chain two function calls together

Tips

- gdb is useful for stepping through the program and following execution
- You will probably want to write a python script (or something) that outputs the final input data, rather than manually writing it
- **Don’t hesitate to ask questions! (Slack, email, etc)**

Homework grading

- Submit your assembly code file to cm7bv@virginia.edu with the subject “MST Assignment 4 - <YOUR_UVA_ID>”
  - eg: “MST Assignment 4 - cm7bv”
- Also, include a brief (1-paragraph) description of what you did and how it went

Useful Resources

- Buffer Overflow tutorial (https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/)
- Useful ROP tutorial (http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html)