Lecture Outline

- Notion of binding time
- Object lifetime and storage management

An aside: Stack Smashing 101
- Slides courtesy of RPISEC/MBE

Scoping
- Static scoping
- Dynamic scoping
Stack Frames

- In x86-64 RBP is fp and RSP is sp. Define the stack frame for the currently executing function
  - local variables
  - pointer to previous frame
  - return address

```c
void foo() {
    long long x = 0x1337;
    char str[16];
    strcpy(str, "ABCDEFGH0123456");
}
```

Note: for 64bit, each 'slot' is 8 bytes
What is corruption?

- So what happens if a programmer makes a simple mistake:

  ```
  char foo[64];
  int money = 0;
  gets(foo);
  ```
gets()?

NAME
gets - get a string from standard input (DEPRECATED)

SYNOPSIS
#include <stdio.h>
char *gets(char *s);

DESCRIPTION
Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ("\0"). No check for buffer overrun is performed (see BUGS below).

- DO NOT EVER USE
  - scanf("%s", ...) as well

• So what happens if we give this program a bunch of A’s?
  With gets we can give as many A’s as we want!
Stack Smashing 101

main() has a stack frame
- Contains local variables
- Pointer to previous frame
- Return address

Lower Memory

Not supposed to touch

Higher Memory

Start of char foo[64]

End of foo

money = 0

Base pointer

RETURN ADDRESS
Stack Smashing 101

As gets() continues to read input, we fill up the 64 bytes allocated for buffer foo.
Stack Smashing 101

As gets() continues to read input, we fill up the 64 bytes allocated for foo.

Go far enough, it corrupts important data!

Not supposed to touch

0x41
0x7fff01020304
0x40051f

Start of char foo[64]
End of foo
Base pointer
RETURN ADDRESS

09/09/2019
Stack Smashing 101

- We can give ourselves money
- If we want to set money to 0x1337beef we need to know:
  - Most x86 machines are little endian (little byte goes first)
  - Meaning the byte order for numbers is "backwards" in memory
  - 0x01020304 would be

```
var_30 = qword ptr -30h
var_28 = qword ptr -28h
var_14 = dword ptr -14h

; __unwind {
    push rbp
    mov rbp, rsp
    push rbx
    sub rbp, 28h
    mov [rbp+var_28], rdi
    mov [rbp+var_30], rsi
    mov rax, [rbp+var_28]
    mov eax, [rax+128h]
    test eax, eax
    jnz short loc_30FB
    mov rax, [rbp+var_28]
    mov edx, [rax+8Ch]
    mov rax, [rbp+var_28]
    mov rdx, [rax+14h]
    mov rax, [rbp+var_30]
    mov eax, [rax+10h]
    mov ebx, ecx
    sub ebx, eax
    mov eax, ebx
    add edx, eax
    mov edx, [rax+var_28]
    mov [rax+8Ch], edx
    ...
    ...
    ; CODE XREF: ZSt4cout
    call _ZSt4cout
    mov rdx, cs:_ZSt4endl
    mov rsi, rdx
    mov rdi, rax
    call _ZSt4endl
    mov rdx, cs:_ZSt4endl
    mov rsi, rdx
    mov rdi, rax
    call _ZSt4endl
    ...
    ...
    ...
```

<table>
<thead>
<tr>
<th>0x04</th>
<th>0x03</th>
<th>0x02</th>
<th>0x01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stack Smashing 201

- What else can we corrupt?
- What happens if you corrupt further? When does it segfault?
  - What was that about a return address?
Stack Smashing 201

When `func()` is called, runtime stores the return address on the stack (i.e., the address of the instruction that immediately follows call `func` in `main`)

```c
int func() {
    puts("Hello World");
    return 17;
}

int main() {
    int res = func();
    return 0;
}
```
Stack Smashing 201

Before the call:

```
0x40051a <main+13>:    call  0x4004f6 <func>
0x40051f <main+18>:   mov   DWORD PTR [rbp-0x4],eax
0x400522 <main+21>:   mov   eax,0x0
0x400527 <main+26>:   leave
0x400528 <main+27>:   ret
```

No argument

```
---------------------------stack--------------------
0000 0x7fffffffffe0f0    -->  0x7fffffffffe1e0    -->  0x1
0008 0x7fffffffffe0f8    -->  0x0
0016 0x7fffffffffe100    -->  0x400530  (__libc_csu_init):
```

```c
var_30 = qword ptr -30h
var_28 = qword ptr -28h
var_14 = dword ptr -14h

; __unwind {
    push rbp
    mov  rbp, rsp
    push rbx
    sub  rsp, 28h
    mov  [rbp+var_28], rdi
    mov  [rbp+var_30], rsi
    mov  rax, [rbp+var_28]
    mov  eax, [rax+128h]
    test eax, eax
    jnz  short loc_30FB
    mov  rax, [rbp+var_28]
    mov  edx, [rax+0Ch]
    mov  rax, [rbp+var_28]
    mov  ecx, [rax+14h]
    mov  rax, [rbp+var_30]
    mov  eax, [rax+10h]
    mov  ebx, ecx
    sub  ebx, eax
    mov  eax, ebx
    add  edx, eax
    mov  rax, [rbp+var_28]
    mov  [rax+0Ch], edx
}

loc_30FB:
    mov  rax, [rbp+var_28]
    mov  eax, [rax+9Ch]
    test eax, eax
    jns  loc_31C4
    mov  rax, [rbp+var_28]
    add  rax, 18h
    mov  rsi, rax
    mov  rax, cs:_ZSt4cout_ptr
    mov  rdi, rax
    call _ZSt11char_traitsIcESai Chuck
    lea  rsi, aIsDead    ;" is dead!"
    mov  rdi, rax
    call _ZSt11char_traitsIcEERSt13��อะไร changes
    mov  rdx, cs:_ZSt4endlIcSt11char_traits
    mov  rdi, rdx
    mov  rdi, rax
    call _ZNStlsEPFRSoS_E ; std::ostream
    mov  rax, [rbp+var_28]
    mov  eax, [rax+8]
    test eax, eax
    jnz  short loc_3180
    lea  rsi, aWannaCheatYes1 ; wanna cheat?
    mov  rax, cs:_ZSt4cout_ptr
    mov  rdi, rax
    call _ZSt11char_traitsIcEERSt13��อะไร changes
    lea  rax, [rbp+var_14]
```
Stack Smashing 201

Before the call:

```
=> 0x40051a <main+13>: call 0x4004f6 <func>
0x40051f <main+18>: mov DWORD PTR [rbp-0x4],eax
0x400522 <main+21>: mov eax,0x0
0x400527 <main+26>: leave
0x400528 <main+27>: ret

No argument
```

After the call:

```
=> 0x4004f6 <func>: push rbp
0x4004fa <func+1>: mov rbp, rsp
0x4004fa <func+4>: sub rsp, 28h
0x400501 <func+11>: mov [rbp+var_28], rdi
0x400506 <func+16>: mov [rbp+var_28], rsi
0x40051f <main+18>: mov rax, [rbp+var_28]
0x400522 <main+21>: mov eax, [rax+128h]
0x400527 <main+26>: test eax, eax
0x400528 <main+27>: jnz short [rbp-0x3FB]
0x400530 <main+30>: mov edx, [rax+0x8Ch]
0x400535 <main+35>: mov eax, [rax+var_28]

0x400550 <main+50>: mov rax, [rbp+var_28]
0x400555 <main+55>: lea eax, [rax+0x14h]
```

Return address points back to where it left off in main.
Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: \texttt{pop \ rip}\n
About to return:
Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: `pop rip`

About to return:

Returned back to main:

```
0x40051f <main+18>: mov DWORD PTR [rax+0x4], eax
0x400522 <main+21>: mov eax,0x0
0x400527 <main+26>: leave
0x400528 <main+27>: ret
0x400529: nop DWORD PTR [rax+0x0]
```

```
loc_30FB:
```
Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: \texttt{pop \ rip}

About to return:

Returned back to main.

What if we change this???

? ???
Stack Smashing 201

**Without corruption:**

- At the end of the function, it **returns**
- **0x40051f** is popped off the stack and stored in rip
- Control goes to that address

We want to change this
Stack Smashing 201

Corrupted:
- At the end of the function, it returns
- 0x4141414141414141 is popped off the stack and stored in rip
- Control goes to that address
- but it's invalid memory...

Segmentation fault

09/09/2019