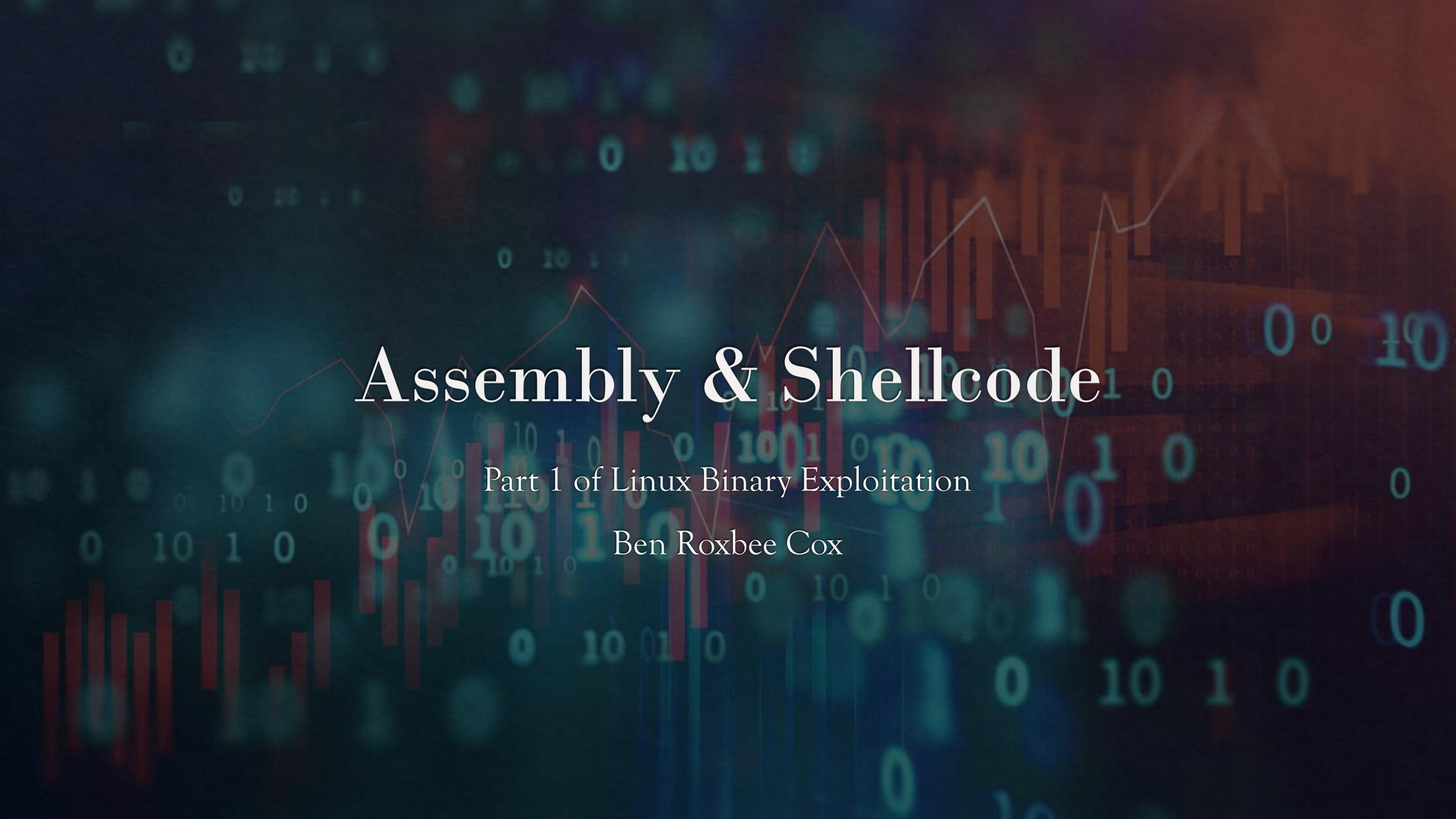
While you wait

- Start Kali/Parrot
- Install pwndbg
  - git clone https://github.com/pwndbg/pwndbg
  - cd pwndbg
  - ./setup.sh



Assembly & Shellcode

Part 1 of Linux Binary Exploitation

Ben Roxbee Cox

What is Binary Exploitation?

- Also known as pwning
- Taking advantage of a bug or misconfiguration in an executable
- Competitions like pwn2own, DefCon qualifiers
- Some binary exploitation concepts change depending on system architecture
- Knowledge required for (basic) binary exploitation
  - Assembly
  - Some knowledge of memory
  - Scripting
- Tools required
  - gdb (preferable with GEF/Pwn-dbg)
  - Python3
  - Pwntools
  - Reverse engineering framework

4 Week Plan

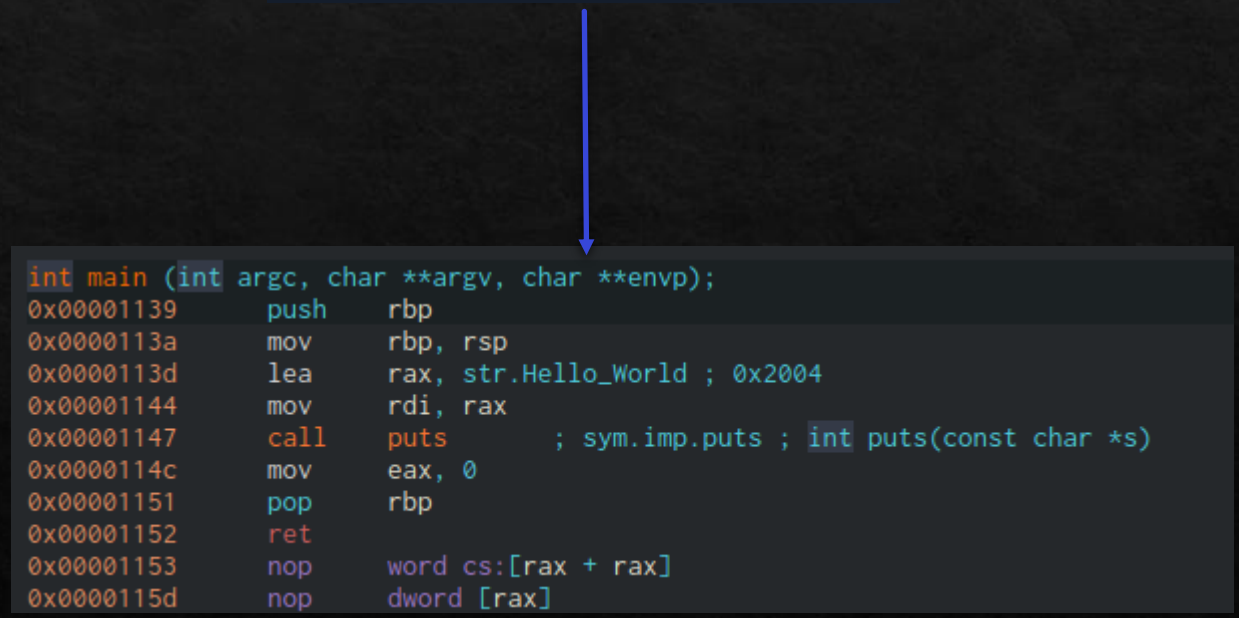
- Week 1: Assembly & Shellcoding
  - Writing our own assembly, writing some shellcode. Getting used to debugging tools
- Week 2: Reverse Engineering
  - Learning some basic reversing techniques, getting used to reversing frameworks
- Week 3: Stack smashing
  - Basic program exploitation. How to exploit programs that have little/no protections
- Week 4: Return Oriented Programming
  - Exploiting programs with some modern protections enabled

Session Plan

- Intro to assembly
- Intro to Memory
- Assembly Activity
- Intro to system calls
- Shellcoding Activity

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


int main(){
    puts("Hello World");
}
```

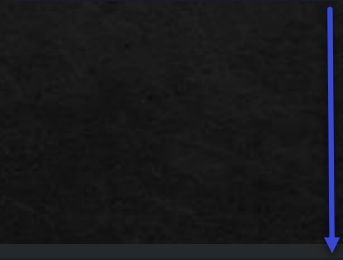


What is assembly & shellcode

- We usually program in high level languages
  - Python
  - C
  - PHP
- Assembly is a low level language
- In binary exploitation it is sometimes possible to write malicious assembly (shellcode) and get the program to execute it

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


int main(){
    puts("Hello World");
}
```



```
int main (int argc, char **argv, char **envp);
0x00001139      push    rbp
0x0000113a      mov     rbp, rsp
0x0000113d      lea     rax, str.Hello_World ; 0x2004
0x00001144      mov     rdi, rax
0x00001147      call    puts        ; sym.imp.puts ; int puts(const char *s)
0x0000114c      mov     eax, 0
0x00001151      pop     rbp
0x00001152      ret
0x00001153      nop     word cs:[rax + rax]
0x0000115d      nop     dword [rax]
```

Registers

- 8 bit (byte)
- 16 bit (word)
- 32 bit (dword)
- 64 bit (qword)

0xFFFFFFFFFFFFFFFF

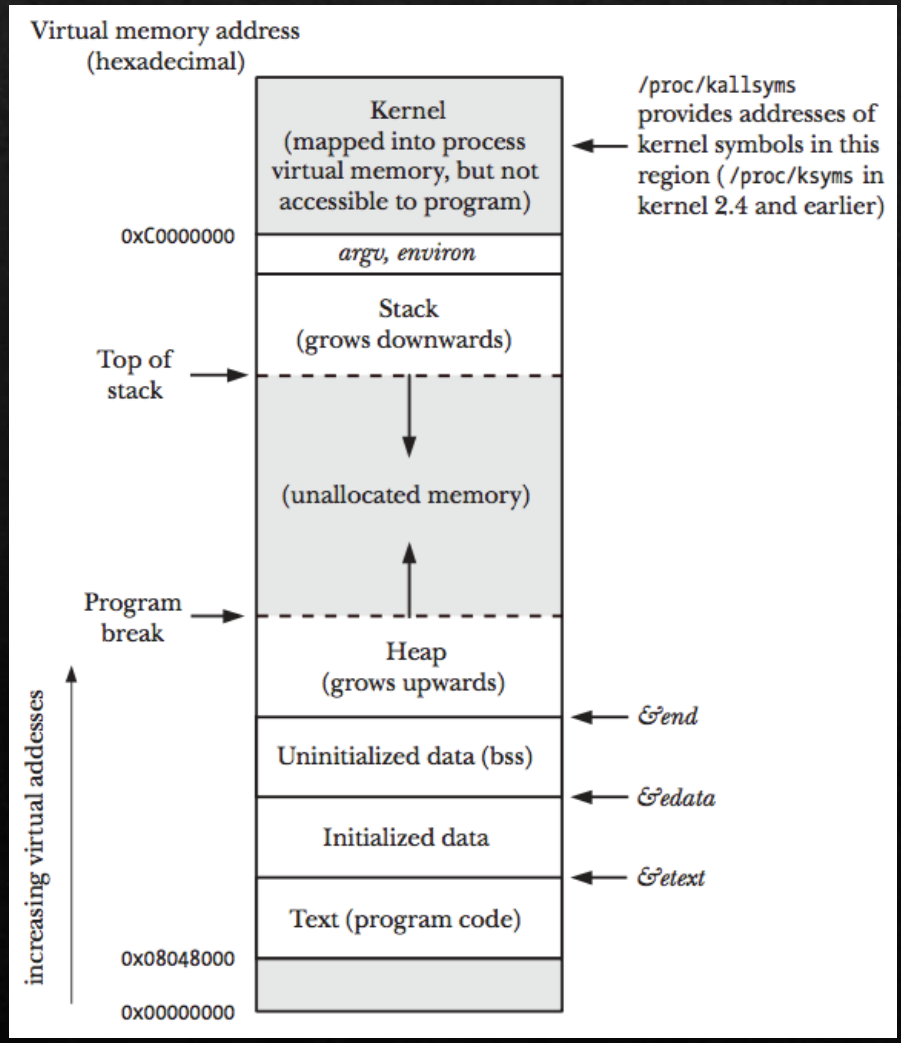
- Registers Store data
- Registers have slightly different names depending on architecture of the system
- Registers store different amounts of data depending on architecture
- Important Registers
  - Instruction Pointer (IP)
  - Registers used for general operations, such as passing data to function calls.
    - AX
    - BX
    - CX
    - DX
    - R8-R15 (General Registers for 64 Bit Processors)
  - Registers used for Stream (or string) operations
    - SI (Source Index) Pointer to start of data input location
    - DI (Destination Index) Pointer to data output location
  - Registers used for Managing the Stack
    - BP: (Base pointer) Points to the current base of the stack
    - SP: (Stack Pointer) Points to the current top of the stack

Some Assembly

```
# Move data into register (mov dest, src)
mov rdi, 10
# Add/Sub/Multiply data to register (add/sub/multiple dest, src)
add rdi, 10
sub rdi, 10
imul rdi, 10
# Division (integer math) the result is stored in rax
mov rax, rdi
div rsi
# https://www.felixcloutier.com/x86/
```

Memory (The Stack)

- In programming, the stack is a logical data structure that obeys LIFO (Last in First Out) principle
- In processes, the stack is a region of memory where data is stored
- Each new function of a process gets it own stack frame
- The stack starts at a high memory location and grows down



Stack Frames

- When a function is called, the data associated with it is stored in the stack frame

1. First any arguments from the calling function are pushed onto the stack.

- NOTE: There is a subtle difference between 32 and 64 bit architectures here:
  - A 32 Bit system will push the arguments via the stack
  - In 64 Bit systems registers are used to pass the first 6 arguments.

2. The address of the Current instruction (the Instruction Pointer) is pushed

  - This acts as a Return Address and tells the program what address it needs to go back to once the function finishes.

3. The Current Base pointer is also pushed

4. Finally the space required for any local Variables for the function is calculated and allocated in the new stack frame

First Assembly Program

```
.global _start
.intel_syntax noprefix

_start:
push 60
push 1337
pop rdi
pop rax
syscall
```

```
$ gcc write.s -o write.elf -nostdlib
-static
```

Assembly Activity

Program the following and watch them in a debugger:

1. Calculate (x * y) / z + a
2. Calculate X ^ 3
    i. Try to build a loop for this
3. Program the logic for:
    1. Calculate y = x/z
    2. If y is odd then exit
    3. If y is even, divide again, loop until odd

```
$ gcc write.s -o write.elf -nostdlib -static
```

Shellcode

Part 1 of Linux Binary Exploitation

Ben Roxbee Cox

Shellcoding

- **Shellcode** is a small piece of code used as the payload in the exploitation of a software vulnerability.
- It is called "**shellcode**" because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called **shellcode**.

Hello World

```
.global _start
.intel_syntax noprefix

_start:
mov rdi, 1
lea rsi, [rip+hello]
mov rdx, 15
mov rax, 1
syscall

hello:
    .ascii "Hello World"
```

```
> gcc write.s -o write.elf -nostdlib -static &&
objcopy --dump-section .text=flag.bin write.elf
```

System Calls and File Descriptors

- System calls are how we make our assembly code interact with the outside world.
- Common system calls are: read, write, open, execve, exit
- In Linux, everything is a file. As such, file descriptors are how we direct the result of our syscalls

x64

```
read(0,rsp, 100)
write(2, rsp, 50)
sendfile(1, 3, 0, 200)
```

https://github.com/Cov-ComSec/HelpfulResources/tree/main/buffer/syscalls

https://syscall.sh/

Example Shellcode 2

```
open:
lea rdi, [rip+flag]
xor rsi, rsi
xor rdx, rdx
mov rax, 2
syscall

sendfile:
mov rdi, 1
mov rsi, 3
mov rdx, 0
mov r10, 60
mov rax, 40
syscall

exit:
mov rax, 60
mov rsi, 42
syscall

flag:
   .ascii "flag\0"
```

- Open
  - file “/flag”
  - flags 0
  - mode 0
- Sendfile
  - To stdin
  - From fd opened by previous syscall
  - With offset 0
  - 60 bytes of data

Example Shellcode 1

```
open:
mov rdi, [rip+flag]
xor rsi, rsi
xor rdx, rdx
mov rax, 2
syscall

read:
mov rdi, rax
mov rsi, rsp
mov rdx, 60
mov rax, 0
Syscall

write:
mov rdi, 1
mov rsi, rsp
mov rdx, 60
mov rax, 1
syscall
flag:
    .ascii "/flag"
```

- Open
  - file “/flag”
  - flags 0
  - mode 0
- Read
  - From fd opened by open
  - From to the stack
  - 60 bytes of data
- Write
  - To stdout
  - From the stack
  - 60 bytes of data

Shellcode Activity

Program the following and watch them in a debugger:

- 3 docker containers available at:
- Python script available too
  - Put your shellcode in the shellcode variable
  - Run it