Intro to Reverse Engineering and Debugging with Radare2

By Chris James

0x00: Who am I?

- Systems Security Professional for Information Security Office @ UF
- Bachelor's of Science in Computer Science from UF
- Started learning about computer and C code in 2003 with SoftICE, Ollydbg, crackmes, and keygens
- Moved on to computer graphics
- Didn't pick it back up again until 2005, 2008 @ UF (Help Desk Malware Removal)
- Joined SIT in 2011/2012, started with forensics, misc, programming. But now back into binary analysis

0x01: Who are you?

- Minimum:
  - Interested in Computer Security
  - Can write programs in a programming language
    - Programming I/II (exposure to C/C++)
- Ideally:
  - Experience with C/C++ and some Assembly
  - Have taken some CS courses:
    - Computer Organization
    - Digital Logic
- Even Better:
  - Operating Systems

0x02: What I'm gonna cover

- Briefly:
  - Source Code
    - What a compiler Does
    - Machine Code
  - Memory Mapping
  - Calling conventions

- More In Depth:
  - Debugging
  - Assembly
    - What is disassembly?
  - Radare2
    - Disassembly
    - Debugging
    - Reversing

0x10: Binary Review

From source to CPU registers

- What does a compiler do to source?
- What is a binary file?
- How does a CPU execute binary?

0x11: Compiling source

- Source file is High-level code e.g. <C>
  - May include shared libraries like <stdlib.h>
- Compiler turns C into ELF/Machine code
  - Literally 1’s and 0’s, which can also be represented as Hexadecimal (base-16)

```
$ cat hello.c
#include <stdio.h>
#include <stdlib.h>

int main(){
    printf("Hello, world!\n");

    exit(0);
}

$ gcc -o hello hello.c
$ ./hello
Hello, world!
```

0x12: Looking at the Binary

- File Magic
  - ELF Header
- Lots of Boilerplate code
- Disassembly: Turning machine code back into ASM representation
- Entry point
  - Virtual Address
  - Real Address
- Map Binary to memory

```
$ file hello
Hello: ELF 64-bit LSB shared obj…

$ strings hello
/lib64/ld-linux-x86-64.so.2…

$ xxd hello | less
00000000: 7f45 4c46 0201 … ELF

$ objdump -Mintel -D ./hello | grep “main>:” -A 8
400546:      55          push rbp

$ readelf -h ./hello | grep Entry
  Entry point address:           0x400450
```

0x20: Memory and Registers

Virtual, Real, Registers

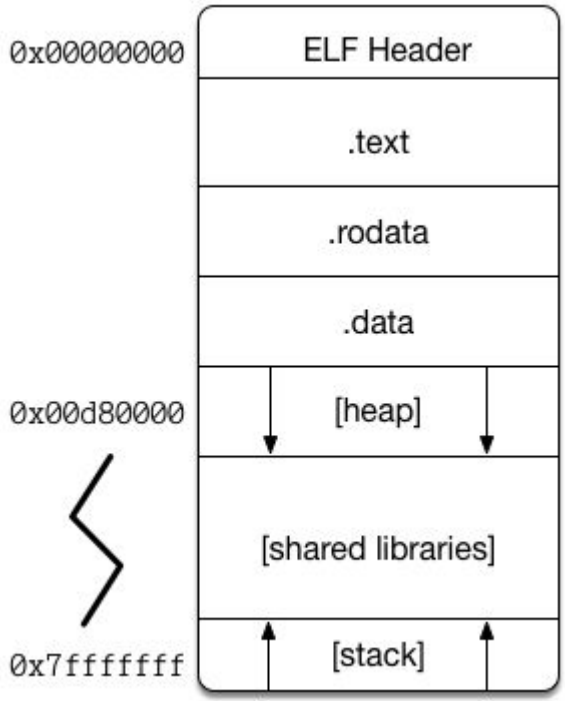
- What is Virtual Memory?
- What is physical Memory?
- Process Image Segments

0x21: Memory

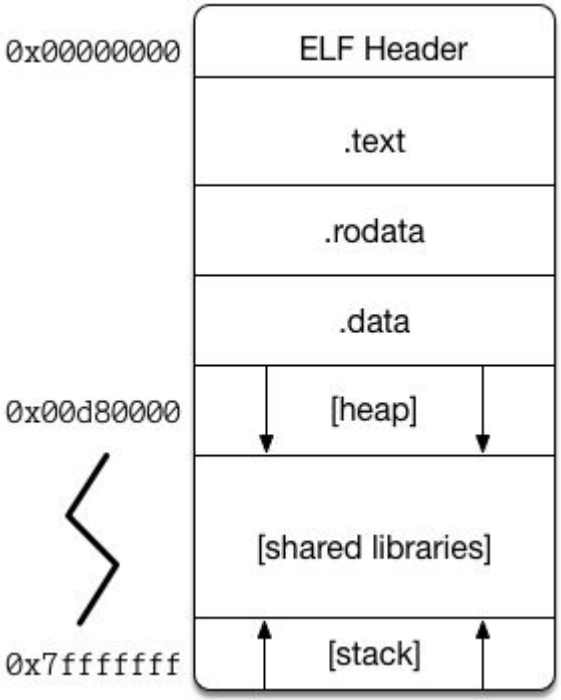
- Memory is addressed by byte
  - 1 byte == 8 bits
  - Value of 1 byte ranges from 0-255
    - 256 discrete values
    - Hex: 0x00 - 0xff
    - Bin: 0b00000000 - 0b11111111
- On 32-bit systems, 2 ^ 32 bytes of addressable memory:
  - 4,294,967,296 Bytes (4 Gibibytes) (approx. 4 Gigabytes)
  - 0x00000000 - 0xffffffff

0x21: Memory

- On 64-bit systems, 2 ^ 64 bytes of addressable memory:
  - 18,446,744,073,709,551,616 Bytes (16 Exbibytes) (approx. 16 Exabytes)
  - 0x0000000000000000 - 0xffffffffffffffff
- Every process granted full address space.
  - How? (Virtual Memory to Physical Memory)
  - But: processes rarely use anywhere near the total Virtual Memory space.

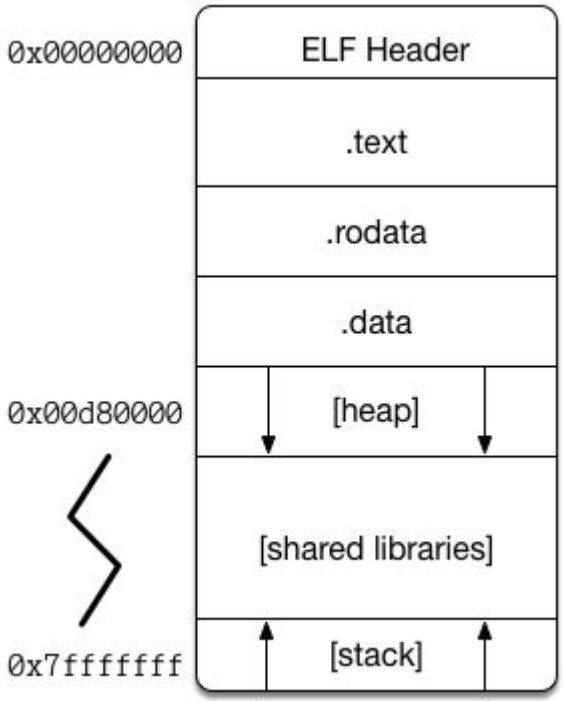


0x22: Process memory layout



- .text (0x400000)
  - Section with executable code
- .(ro)data
  - Sections with initialized variables
- heap
  - malloc scratchpad
- Shared libraries
  - C std lib
- Stack (0x7fffffff)
  - Local function scratchpads

0x22: Process memory layout



- All code (.text) and data exists between 0x0 and 0xd80000 (about 14 MB)
  - .0000000000766% of the way through address space
- Stack starts at 0x7fffffff
  - .0000000116% of the way through address space
  - 0x7f27ffff bytes between end of .text/.data and stack
    - Approx 2 GB of space for heap and Stack to grow

0x23: Registers

- CPU Registers == fastest memory
- Instruction Pointer:
  - **rip**: “what executes next”
- General Purpose:
  - **rax**: return values
  - **rbx**, **rcx**, **rdx**
- Stack:
  - **rsp**: stack pointer (top)
  - **rbp**: base pointer (bottom)
- Data:
  - **rsi**: source index
  - **rdi**: destination index
- Other:
  - **r8-r15**

- Can address different parts of a register:
- 0x1122334455667788

```
  ================ rax (64 bits)
          ======== eax (32 bits)
              ====  ax (16 bits)
              ==    ah (8 bits)
                ==  al (8 bits)
```

- Syscalls:
  - **rax**: syscall number
  - **rdi**: arg0
  - **rsi**: arg1
  - **rdx**: arg2
  - **r10-r8-r9**: arg3-arg5

0x30: Assembly

Machine code to logic

- Instructions
- Function Prologue & Epilogue
- Stack frames

0x31: Assembly Instructions

- Intel vs AT&T (Intel is better)
  - Intel: <inst> <dst>,<src>
  - AT&T: <inst> <src>,<dst>
- Side effects:
  - CPU Flags:
    - ZF: cmp, jump, test
  - Stack:
    - push, pop, call, leave, ret
- Control Flow:
  - call, jump

```
$ objdump -Mintel -D ./hello | grep “main>:” -A 8
400546:   55                    push   rbp
400547:   48 89 e5              mov    rbp,rsp
40054a:   bf e4 05 40 00        mov    edi,0x4005e4
40054f:   e8 dc fe ff ff        call   400430 <puts@plt>
400554:   bf 00 00 00 00        mov    edi,0x0
400559:   e8 e2 fe ff ff        call   400440 <exit@plt>
```

```
$ objdump -D ./hello | grep “main>:” -A 8
400546:   55                    push   %rbp
400547:   48 89 e5              mov    %rsp,%rbp
40054a:   bf e4 05 40 00        mov    $0x4005e4,%edi
40054f:   e8 dc fe ff ff        callq  400430 <puts@plt>
400554:   bf 00 00 00 00        mov    $0x0,%edi
400559:   e8 e2 fe ff ff        callq  400440 <exit@plt>
```

0x32: Function Prologue and Epilogue

- Syscalls:
  - **rax**: syscall number
  - **rdi**: arg0
  - **rsi**: arg1
  - **rdx**: arg2
  - **r10-r8-r9**: arg3-arg5
- Function Calls:
  - **rax**: return value
  - **rdi**: arg0
  - **rsi**: arg1
  - **rdx**: arg2
  - **rcx-r8-r9**: arg3-arg5

- **call** <address>
  - Same as:
    - **push** rip+len(instruc)
    - **jmp** <address>
- Function prologue:
  - **push** rbp
  - **mov** rbp, rsp
  - **sub** rsp, 0x20
- Function Epilogue
  - **leave**
    - Combines:
      - **mov** rsp, rbp
      - **pop** rbp
  - **ret**
    - Same as “**pop** rip”

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1c**c0**
- rbp: 0x7ffe89cd1c**d0**
- rip: 0x00400**97**
- Disassembly:

```
0x00400597    e8aaffffff  call 0x400546
0x0040059c    bf00000000  mov edi, 0
0x004005a1    e89afeffff  call sym.imp.exit
```

```
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1c**b8**
- rbp: 0x7ffe89cd1cd0
- rip: 0x00400**46**
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave
0x00400560   c3          ret
```

```
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1c**b0**
- rbp: 0x7ffe89cd1cd0
- rip: 0x0040054**7**
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave
0x00400560   c3          ret
```

```
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1cb0
- rbp: **0x7ffe89cd1cb0**
- rip: 0x0040054a
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave
0x00400560   c3          ret
```

```
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x0000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1ca0
- rbp: 0x7ffe89cd1cb0
- rip: 0x0040054e
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave
0x00400560   c3          ret
```

```
0x7ffe89cd1ca0 <uninitialized data>
0x7ffe89cd1ca8 <uninitialized data>
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x0000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1ca0
- rbp: 0x7ffe89cd1cb0
- rip: 0x00400**52**
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave
0x00400560   c3          ret
```

```
0x7ffe89cd1ca0 <uninitialized data>
0x7ffe89cd1ca8 0x00007ffe89cd1cc0
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x0000000004005b0
```

0x33: Stack Frames (skipped over puts to leave)

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1ca0
- rbp: 0x7ffe89cd1cb0
- rip: 0x0040055f
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave (mov rsp, rbp;pop rbp)
0x00400560   c3          ret
```

```
0x7ffe89cd1ca0 <uninitialized data>
0x7ffe89cd1ca8 0x00007ffe89cd1cc0
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x0000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1c**b8**
- rbp: 0x7ffe89cd1cd0
- rip: 0x0040056**0**
- Disassembly:

```
0x00400546   55          push rbp
0x00400547   4889e5      mov rbp, rsp
0x0040054a   4883ec10    sub rsp, 0x10
0x0040054e   48897df8    mov qword [rbp - 8], rdi
0x00400552   488b45f8    mov rax, qword [rbp - 8]
0x00400556   4889c7      mov rdi, rax
0x00400559   e8d2feffff  call sym.imp.puts
0x0040055e   90          nop
0x0040055f   c9          leave
0x00400560   c3          ret    (pop rip)
```

```
0x7ffe89cd1ca0 <uninitialized data>
0x7ffe89cd1ca8 0x00007ffe89cd1cc0
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x33: Stack Frames

- rdi: 0x7ffe89cd1cc0 char* -> Hello, world!
- rsp: 0x7ffe89cd1c**c0**
- rbp: 0x7ffe89cd1cd0
- rip: 0x0040059**c**
- Disassembly:

```
0x00400597    e8aafffffe   call 0x400546
0x0040059c    bf00000000   mov edi, 0
0x004005a1    e89afefffe   call sym.imp.exit
```

```
0x7ffe89cd1ca0 <uninitialized data>
0x7ffe89cd1ca8 0x00007ffe89cd1cc0
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x000000000040059c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x000000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x34: Quick note about Endianness

- Memory addresses in this binary are reprented in **little-endian** byte order.
- Thus, to the right, the address
  0x7ffe89cd1c**c0** addresses the byte '0x48',
  0x7ffe89cd1c**c1** addresses the byte '0x65',
  ...
  0x7ffe89cd1c**cc** addresses the byte '0x21'

- Thus memory addresses appear 'correct' when little-endianness is accounted for, but strings appear backward
- When printing in sequential order, memory address appear backward-ordered (by byte) but strings appear correct.
- TRY IT:
  - `px**q** 8 @ rbp` vs. `px 8 @ rbp`
  - `px**q** 16 @ str.Hello__world_`
  - `px 16 @ str.Hello__world_`

```
0x7ffe89cd1ca0 <uninitialized data>
0x7ffe89cd1ca8 0x00007ffe89cd1cc0
0x7ffe89cd1cb0 0x00007ffe89cd1cd0
0x7ffe89cd1cb8 0x0000000000400 59c
0x7ffe89cd1cc0 0x77202c6f6c6c6548
0x7ffe89cd1cc8 0x0000021646c726f
0x7ffe89cd1cd0 0x00000000004005b0
```

0x40: Radare2

Computer Wizard's Spellbook

- Disassembly
- Debugging
- Scripting

0x41: Configure radare2 for debugging

```
$ cat ~/.radare2rc
e scr.wheel=false
e stack.bytes=false
e stack.size=114

$ cat ./<programName>.rr2
#!/usr/bin/env rarun2
program=<programName>
arg0=”./<programName>”
stdio=/dev/pts/<##>
```

```
$ tty
/dev/pts/##

$ clear; sleep 99999999999999999;
```

0x42: Debugging in radare2

```
$ r2 -d rarun2 -R ./<programName>.rr2
```

- Every command is a mnemonic
- Use `?` to see help with any command
  - E.g. `a?` will show all analysis command reference
- Most commands have subcommands
  - `db?`, `dc?`
- Radare2 Essential commands:
  - `aaa`, `s`, `pd`, `px[wq]`, `ps`, `db`, `dbt`, `dc`, `dcr`, `ds`, `dr[r]`, `ood`, `dm`

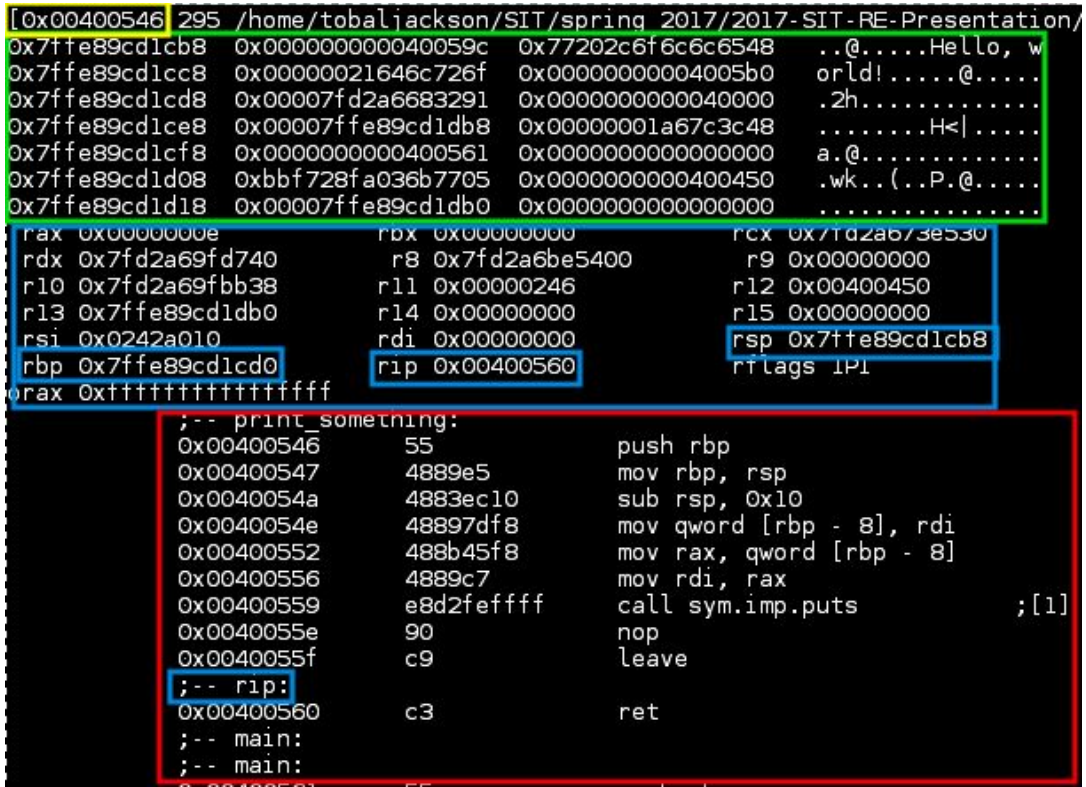
- Breakpoints are fundamental to debugging
  - `db <addr/sym>` to set a breakpoint
  - `dc` to continue execution until you hit a breakpoint or program completion
  - `ds` to step instructions and *into* calls
  - `dso` to step instructions and *over* calls
  - `dcr` continues until a `ret` instruction!

0x43: Visual Mode

- `**V**` to enter visual mode
- `**?**` to see visual mode keyboard shortcuts
- `**:**` to enter cmd mode
  - <**enter**> to exit cmd mode
- `**p**` to cycle view modes
- `**c**` to enter/exit cursor mode
  - `h**jk**l` to navigate cursor (vim keys), or arrow keys
  - `**b**` to set breakpoint
  - `**wx**` (in write mode) to write bytes
  - `**wa**` (in write mode) to write assembly
- `**u**` to undo seek
- `**s**` to step into
- `**S**` to step over (capitalized with <shift>)
- `**.**` to seek to rip
- `**_**` to view Flags

0x44: Visual Mode UI

- Yellow == Current Seek address
- Green == Stack view
- Blue == Registers
- Red == Disassembly



0x45: First binary walkthrough: hello

- Commandline Sequence:
  - aaa #analyze
  - db main #set break point
  - dc #continue exec
  - pd 10 #print 10 instruct.
  - 3ds #debug-step 3 times
  - s rip #seek to current rip
  - ps @ rdi #print str
  - pd 3 #print 3 instruct.
  - dso #step over
  - dc #continue

- Visual mode Sequence (**better!**):
  - aaa
  - db main
  - dc
  - **V** #start Viz mode
  - **pp** #switch to debug view
  - sss #step 3x
  - **:ps @ rdi** #print str @ rdi
  - **:<enter>** #exit cmdline mode
  - S #(capital) Step-over
  - **:dc** #debug continue

0x46: Helpful Tips for Exercises:

- Slides **0x42** and **0x43** provide useful commands for both command and visual modes
- Use `**?**` or `**??**` after a command for help!
- Split your terminal window with **<ctrl+shift+O>** and **<ctrl+shift+E>**!
- If you accidently end up in no-man's-land, using `**:ood <args>**` will re-open the binary in radare2 with any optional arguments you'd like (unless you used the .rr2 rarun2 profile)
- Refer to this site for assembly instruction reference.
- If you need to back out of any menus from visual mode use `**q**` to quit out of them.
- If you're new to all this, start at `**re1**` and open up `walkthrough.txt` using `less` or `nano` or `vim`:
  - `$ less walkthrough.txt`
- If you have any questions about anything, please ask me or any of the SIT officers and we'll be glad to help!
- I encourage you to work in groups since the complexity of this stuff is high and teamwork can help!
- `**:dcr**` will continue until return!

0x47: External resources

— — —

0x11: Compiling source
Working with Hexadecimal: https://learn.sparkfun.com/tutorials/hexadecimal
High-level article on compilers: https://en.wikipedia.org/wiki/Compiler

0x12: Looking at the Binary
What is File Magic?: https://en.wikipedia.org/wiki/Magic_number_(programming)#Format_indicator
Commands used: file, strings, xxd, less, objdump, grep,
For help with these commands, just use `man <command>` to show the manual pages.
For information on how linux PIPES (“|”) work, check out:
https://superuser.com/questions/756158/what-does-the-linux-pipe-symbol-do

0x20: Memory and Registers
Subject matter learned in Computer Organization: processor pipelining, memory types vs speed, Instruction decoding.
High-level Register reference: https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Overall/register.html

0x21: Memory
Virtual-Physical memory mapping learned in OS
High-level overview of Linux Memory Management: http://www.thegeekstuff.com/2012/02/linux-memory-management/

0x22: Process memory layout
Elf File format: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
Process memory overview: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/
Take note that the above link reverses address direction (high-on-top) whereas the better way is (low-on-top)

0x23 Registers:
Learned about memory timings and CPU caching in Comp Org
Register reference: https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start
Syscall table: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

0x47: External resources

— — —

0x31: Assembly Instructions
High-level overview of Assembly: http://ian.seyler.me/easy_x86-64/
x86 Instruction reference: https://www.aldeid.com/wiki/X86-assembly#Pages_in_this_category
Video tutorial of basic assembly: https://www.youtube.com/watch?v=busHtSyx2-w

0x32: Function Prologue and Epilogue
Look here for which registers are preserved across function/syscalls:
https://stackoverflow.com/questions/18024672/what-registers-are-preserved-through-a-linux-x86-64-function-call
Stack frame layout on x86-64: http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
Ridiculously drawn (with terrible audio) but accurate: https://www.youtube.com/watch?v=kSgrKtA0rJM

0x33: Stack Frames
Use `man ascii` to see what ordinal values correspond to which letters of the alphabet! (or visit a page like http://www.ascii-code.com/)

0x34: Quick note about Endianness
More about endianness: https://en.wikipedia.org/wiki/Endianness

0x40: Radare2
Official radare2 repo (with install instructions): https://github.com/radare/radare2
My custom radare2 Cheat Sheet:
https://docs.google.com/document/d/1our_fcFcufIJ13QsZoDuGOEBqftF6o0zEkDsqzAy43U/edit?usp=sharing
Unofficial radare2 Cheat Sheet (a little outdated):
https://github.com/pwntester/cheatsheets/blob/master/radare2.md