Reverse Engineering

Week 2: Memory Structures, Dynamic RE, Linus syscalls/ABI

Sergey Bratus, Travis Goodspeed and Ryan Speers -- Dartmouth College -- Winter 2022

Overview

What & Why

- Today will mostly be us stepping through examples followed by a lab to do on your own
- The goal is to get familiar with a few common utilities and to give you a chance to ask any questions
- The homework for today will just be doing what you learn today on a binary that you hadn’t seen before

Motivation

Cool RE

- https://reactos.org
- Windows is “closed source”
- This group is slowly reverse engineering Windows components and rewriting them in C/C++.

nasm - netwide assembler

What & Why

- Used to compile assembly programs into binary blobs or even ELF files
- I primarily use it to compile snippets of assembly to see the corresponding machine code
- Really useful for something you may learn later called shellcode.
- Connect to a babylon server and we will start debugging together

nasm - netwide assembler

Write the code

nasm - netwide assembler

Compile it

```
f00543d@babylon1:~/recourse/week2$ nasm -f bin assembly.asm
f00543d@babylon1:~/recourse/week2$
```

nasm - netwide assembler

Check it out

```
f00543d@babylon1:~/recourse/week2$ nasm -f bin assembly.asm
f00543d@babylon1:~/recourse/week2$ ndisasm -b 64 assembly
00000000  4889F8            mov rax,rdi
00000003  C3                ret
f00543d@babylon1:~/recourse/week2$
```

nasm - netwide assembler

Format it for use

```
f00543d@babylon1:~/recourse/week2$ nasm -f bin assembly.asm
f00543d@babylon1:~/recourse/week2$ ndisasm -b 64 assembly
00000000  4889F8            mov rax,rdi
00000003  C3                ret
f00543d@babylon1:~/recourse/week2$ xxd -i assembly
unsigned char assembly[] = {
  0x48, 0x89, 0xf8, 0xc3
};
unsigned int assembly_len = 4;
f00543d@babylon1:~/recourse/week2$
```

nasm - netwide assembler

Write a program to run the machine code

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

unsigned char assembly[] = {
  0x48, 0x89, 0xf8, 0xc3
};
unsigned int assembly_len = 4;

int main( int argc, char **argv)
{
  int retval = 0;

  int (*sc)(int) = (int(*)(int))assembly;

  retval = sc( atoi(argv[1])  );

  printf("Return value: %d\n", retval);

  return 0;
}
~
~
~
~
~
~
"test_exec.c" 20L, 315C                                          20,1          All
```

nasm - netwide assembler

Compile it

```
f00543d@babylon1:~/recourse/week2$ nasm -f bin assembly.asm
f00543d@babylon1:~/recourse/week2$ ndisasm -b 64 assembly
00000000  4889F8            mov rax,rdi
00000003  C3                ret
f00543d@babylon1:~/recourse/week2$ xxd -i assembly
unsigned char assembly[] = {
  0x48, 0x89, 0xf8, 0xc3
};
unsigned int assembly_len = 4;
f00543d@babylon1:~/recourse/week2$ gcc test_exec.c -z execstack -o test_exec
f00543d@babylon1:~/recourse/week2$
```

nasm - netwide assembler

Run it

```
f00543d@babylon1:~/recourse/week2$ nasm -f bin assembly.asm
f00543d@babylon1:~/recourse/week2$ ndisasm -b 64 assembly
00000000  4889F8            mov rax,rdi
00000003  C3                ret
f00543d@babylon1:~/recourse/week2$ xxd -i assembly
unsigned char assembly[] = {
  0x48, 0x89, 0xf8, 0xc3
};
unsigned int assembly_len = 4;
f00543d@babylon1:~/recourse/week2$ gcc test_exec.c -z execstack -o test_exec
f00543d@babylon1:~/recourse/week2$ ./test_exec 1337
Return value: 1337
f00543d@babylon1:~/recourse/week2$
```

gdb - Gnu Debugger

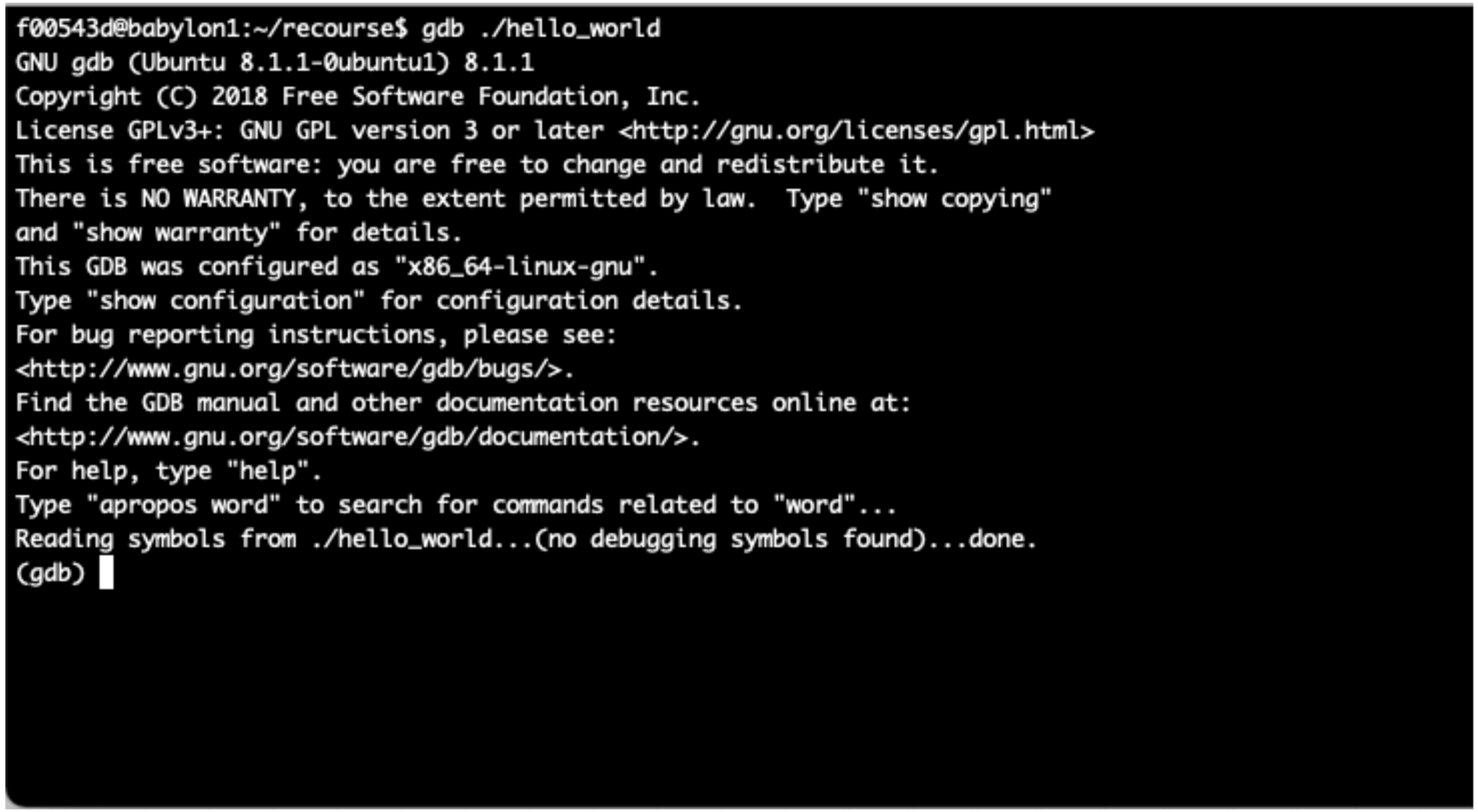
What & Why

- Can attach to an already running process or launch one itself
- Allows you to see the program state and step through a concrete execution
- Sometimes it really helps to see an actual execution when dealing with complex code
- Connect to a babylon server and we will start debugging together

gdb - Gnu Debugger

Launch

- Copy the `hello_world` binary into your folder and in your terminal run
  - `gdb hello_world`



gdb - Gnu Debugger

A look around

- The application isn't currently running but there are a few things that we can do.
- Let's check out the disassembly for main.
  - run 'disassemble main'
  - The output you see is in AT&T Gas syntax
  - To see it as INTEL syntax run
    - 'set disassembly-flavor intel'

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000000117d <+0>:     endbr64
   0x0000000000001181 <+4>:     push   %rbp
   0x0000000000001182 <+5>:     mov    %rsp,%rbp
   0x0000000000001185 <+8>:     sub    $0x10,%rsp
   0x0000000000001189 <+12>:    mov    %edi,-0x4(%rbp)
   0x000000000000118c <+15>:    mov    %rsi,-0x10(%rbp)
   0x0000000000001190 <+19>:    cmpl   $0x1,-0x4(%rbp)
   0x0000000000001194 <+23>:    jle    0x11ab <main+46>
   0x0000000000001196 <+25>:    mov    -0x10(%rbp),%rax
   0x000000000000119a <+29>:    add    $0x8,%rax
   0x000000000000119e <+33>:    mov    (%rax),%rax
   0x00000000000011a1 <+36>:    mov    %rax,%rdi
   0x00000000000011a4 <+39>:    callq  0x1149 <print_string>
   0x00000000000011a9 <+44>:    jmp    0x11b7 <main+58>
   0x00000000000011ab <+46>:    lea    0xe5e(%rip),%rdi        # 0x2010
   0x00000000000011b2 <+53>:    callq  0x1149 <print_string>
   0x00000000000011b7 <+58>:    mov    $0x0,%eax
   0x00000000000011bc <+63>:    leaveq
   0x00000000000011bd <+64>:    retq
End of assembler dump.
(gdb)
```

gdb - Gnu Debugger

A look around

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000000117d <+0>:     endbr64
   0x0000000000001181 <+4>:     push   rbp
   0x0000000000001182 <+5>:     mov    rbp,rsp
   0x0000000000001185 <+8>:     sub    rsp,0x10
   0x0000000000001189 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x000000000000118c <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000001190 <+19>:    cmp    DWORD PTR [rbp-0x4],0x1
   0x0000000000001194 <+23>:    jle    0x11ab <main+46>
   0x0000000000001196 <+25>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000000119a <+29>:    add    rax,0x8
   0x000000000000119e <+33>:    mov    rax,QWORD PTR [rax]
   0x00000000000011a1 <+36>:    mov    rdi,rax
   0x00000000000011a4 <+39>:    call   0x1149 <print_string>
   0x00000000000011a9 <+44>:    jmp    0x11b7 <main+58>
   0x00000000000011ab <+46>:    lea    rdi,[rip+0xe5e]        # 0x2010
   0x00000000000011b2 <+53>:    call   0x1149 <print_string>
   0x00000000000011b7 <+58>:    mov    eax,0x0
   0x00000000000011bc <+63>:    leave
   0x00000000000011bd <+64>:    ret
End of assembler dump.
(gdb)
```

gdb - Gnu Debugger

A look around

- You will see that the address of the first instruction is 0x117d. Remember this when we run the program.
- To view a single instruction run: 'x /i <address>'
  - Stands for eXamine instruction at <address>
  - You can print multiple instructions via 'x /##i <address>' where ## is how many to print

gdb - Gnu Debugger

A look around

```
   0x0000000000001181 <+4>:     push   rbp
   0x0000000000001182 <+5>:     mov    rbp,rsp
   0x0000000000001185 <+8>:     sub    rsp,0x10
   0x0000000000001189 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x000000000000118c <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000001190 <+19>:    cmp    DWORD PTR [rbp-0x4],0x1
   0x0000000000001194 <+23>:    jle    0x11ab <main+46>
   0x0000000000001196 <+25>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000000119a <+29>:    add    rax,0x8
   0x000000000000119e <+33>:    mov    rax,QWORD PTR [rax]
   0x00000000000011a1 <+36>:    mov    rdi,rax
   0x00000000000011a4 <+39>:    call   0x1149 <print_string>
   0x00000000000011a9 <+44>:    jmp    0x11b7 <main+58>
   0x00000000000011ab <+46>:    lea    rdi,[rip+0xe5e]        # 0x2010
   0x00000000000011b2 <+53>:    call   0x1149 <print_string>
   0x00000000000011b7 <+58>:    mov    eax,0x0
   0x00000000000011bc <+63>:    leave
   0x00000000000011bd <+64>:    ret
End of assembler dump.
(gdb) x /i main
   0x117d <main>:       endbr64
(gdb) x /i 0x117d
   0x117d <main>:       endbr64
(gdb)
```

gdb - Gnu Debugger

A look around

```
   0x00000000000011a9 <+44>:    jmp    0x11b7 <main+58>
   0x00000000000011ab <+46>:    lea    rdi,[rip+0xe5e]        # 0x2010
   0x00000000000011b2 <+53>:    call   0x1149 <print_string>
   0x00000000000011b7 <+58>:    mov    eax,0x0
   0x00000000000011bc <+63>:    leave
   0x00000000000011bd <+64>:    ret
End of assembler dump.
(gdb) x /i main
   0x117d <main>:       endbr64
(gdb) x /i 0x117d
   0x117d <main>:       endbr64
(gdb) x /5i main
   0x117d <main>:       endbr64
   0x1181 <main+4>:     push   rbp
   0x1182 <main+5>:     mov    rbp,rsp
   0x1185 <main+8>:     sub    rsp,0x10
   0x1189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb) x /5i 0x117d
   0x117d <main>:       endbr64
   0x1181 <main+4>:     push   rbp
   0x1182 <main+5>:     mov    rbp,rsp
   0x1185 <main+8>:     sub    rsp,0x10
   0x1189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb)
```

gdb - Gnu Debugger

Breakpoints

- Breakpoints are ways to stop execution at different points in the program so that you may examine what is going on
- 'break *<address>|<symbol>'
- Let's set a breakpoint on the main function before we run it. This way we can see the execution of the function. If we just ran the application without a breakpoint it would just execute the entire thing
- Can be done two ways:
  - break *0x117d
  - break *main

gdb - Gnu Debugger

A look around

```
   0x00000000000011b2 <+53>:    call   0x1149 <print_string>
   0x00000000000011b7 <+58>:    mov    eax,0x0
   0x00000000000011bc <+63>:    leave
   0x00000000000011bd <+64>:    ret
End of assembler dump.
(gdb) x /i main
   0x117d <main>:       endbr64
(gdb) x /i 0x117d
   0x117d <main>:       endbr64
(gdb) x /5i main
   0x117d <main>:       endbr64
   0x1181 <main+4>:     push   rbp
   0x1182 <main+5>:     mov    rbp,rsp
   0x1185 <main+8>:     sub    rsp,0x10
   0x1189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb) x /5i 0x117d
   0x117d <main>:       endbr64
   0x1181 <main+4>:     push   rbp
   0x1182 <main+5>:     mov    rbp,rsp
   0x1185 <main+8>:     sub    rsp,0x10
   0x1189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb) break *main
Breakpoint 1 at 0x117d
(gdb)
```

gdb - Gnu Debugger

A look around

- Now let's run the application
- type 'r'
- We want to be able to see the instructions as we step through so run
  - display /i $rip
  - This is a command that runs after each step to print the instruction that is pointed to by the RIP register.

gdb - Gnu Debugger

A look around

```
(gdb) x /i 0x117d
   0x117d <main>:       endbr64
(gdb) x /5i main
   0x117d <main>:       endbr64
   0x1181 <main+4>:     push   rbp
   0x1182 <main+5>:     mov    rbp,rsp
   0x1185 <main+8>:     sub    rsp,0x10
   0x1189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb) x /5i 0x117d
   0x117d <main>:       endbr64
   0x1181 <main+4>:     push   rbp
   0x1182 <main+5>:     mov    rbp,rsp
   0x1185 <main+8>:     sub    rsp,0x10
   0x1189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb) break *main
Breakpoint 1 at 0x117d
(gdb) r
Starting program: /thayerfs/home/f00543d/recourse/hello_world

Breakpoint 1, 0x000055555555517d in main ()
(gdb) display/i $rip
1: x/i $rip
=> 0x55555555517d <main>:       endbr64
(gdb)
```

gdb - Gnu Debugger

A look around

- Step through a few instructions by typing ‘si'
- If you type it once then continue hitting enter then gdb will repeate your previous command.

```
(gdb) r
Starting program: /thayerfs/home/f00543d/recourse/hello_world

Breakpoint 1, 0x000055555555517d in main ()
(gdb) display/i $rip
1: x/i $rip
=> 0x55555555517d <main>:       endbr64
(gdb) si
0x0000555555555181 in main ()
1: x/i $rip
=> 0x555555555181 <main+4>:     push   rbp
(gdb)
0x0000555555555182 in main ()
1: x/i $rip
=> 0x555555555182 <main+5>:     mov    rbp,rsp
(gdb)
0x0000555555555185 in main ()
1: x/i $rip
=> 0x555555555185 <main+8>:     sub    rsp,0x10
(gdb)
0x0000555555555189 in main ()
1: x/i $rip
=> 0x555555555189 <main+12>:    mov    DWORD PTR [rbp-0x4],edi
(gdb)
```

gdb - Gnu Debugger

Short aside: ASLR

- Address Space Layout Randomization
- Many years ago the address space wasn’t randomised. An application that was executed on one computer would have the same address space each time it ran on any computer it ran on.
- See any problems with this?
- Now: By default, programs are loaded at random addresses. Also, the stack and heap spaces are usually randomised
- Caveat, when you launch a program from gdb ASLR is often turned off.

gdb - Gnu Debugger

A look around

- Check out the register state: ‘i r’ short for ‘info reg’
- What is rdi? rsi?

```
rax            0x55555555517d   93824992235901
rbx            0x0      0
rcx            0x5555555551c0   93824992235968
rdx            0x7fffffffe5c8   140737488348616
rsi            0x7fffffffe5b8   140737488348600
rdi            0x1      1
rbp            0x7fffffffe4d0   0x7fffffffe4d0
rsp            0x7fffffffe4c0   0x7fffffffe4c0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x2      2
r11            0x1f     31
r12            0x555555555060   93824992235616
r13            0x7fffffffe5b0   140737488348592
r14            0x0      0
r15            0x0      0
rip            0x555555555189   0x555555555189 <main+12>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
---Type <return> to continue, or q <return> to quit---
```

gdb - Gnu Debugger

A look around

- Examine Memory:
- ‘x /gx’
  - ‘x’ Says that we want to look at something
  - ‘g’ indicates the size. Could be ‘g’, ’w’, ‘h’, ‘b’ for 64-,32-,16-, and 8-bits
  - ‘x ‘ display as hex digits. You can also specify other types such as ’s’ for string

```
es             0x0      0
fs             0x0      0
---Type <return> to continue, or q <return> to quit---
gs             0x0      0
k0             0x0      0
k1             0x0      0
k2             0x0      0
k3             0x0      0
k4             0x0      0
k5             0x0      0
k6             0x0      0
k7             0x0      0
(gdb) x /gx $rsi
0x7fffffffe5b8: 0x00007fffffffe839
(gdb) x /2gx $rsi
0x7fffffffe5b8: 0x00007fffffffe839      0x0000000000000000
(gdb) x /gx 0x00007fffffffe839
0x7fffffffe839: 0x6672657961687420f
(gdb) x /s 0x00007fffffffe839
0x7fffffffe839: "/thayerfs/home/f00543d/recourse/hello_world"
(gdb) x /16bx 0x00007fffffffe839
0x7fffffffe839: 0x2f    0x74    0x68    0x61    0x79    0x65    0x72    0x66
0x7fffffffe841: 0x73    0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x66
(gdb)
```

gdb - Gnu Debugger

A look around

- Single step until the comparison
- Examine the memory referenced by that address
- What are we looking at?

```
(gdb) x /gx $rsi
0x7fffffffe5b8: 0x00007fffffffe839
(gdb) x /2gx $rsi
0x7fffffffe5b8: 0x00007fffffffe839      0x0000000000000000
(gdb) x /gx 0x00007fffffffe839
0x7fffffffe839: 0x667265796168742f
(gdb) x /s 0x00007fffffffe839
0x7fffffffe839: "/thayerfs/home/f00543d/recourse/hello_world"
(gdb) x /16bx 0x00007fffffffe839
0x7fffffffe839: 0x2f    0x74    0x68    0x61    0x79    0x65    0x72    0x66
0x7fffffffe841: 0x73    0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x66
(gdb) si
0x000055555555518c in main ()
1: x/i $pc
=> 0x55555555518c <main+15>:    mov    QWORD PTR [rbp-0x10],rsi
(gdb) si
0x0000555555555190 in main ()
1: x/i $pc
=> 0x555555555190 <main+19>:    cmp    DWORD PTR [rbp-0x4],0x1
(gdb) x /dx $rbp-4
0x7fffffffe4cc: 0x01
(gdb) x /wx $rbp-4
0x7fffffffe4cc: 0x00000001
(gdb)
```

gdb - Gnu Debugger

A look around

- Check out the flags that are set prior to the comparison.
- Single step then look at them again.
- Did anything change?
- If so what?
- Will this jump be taken?

```
(gdb) x /16bx 0x00007fffffffe839
0x7fffffffe839: 0x2f    0x74    0x68    0x61    0x79    0x65    0x72    0x66
0x7fffffffe841: 0x73    0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x66
(gdb) si
0x000055555555518c in main ()
1: x/i $pc
=> 0x55555555518c <main+15>:    mov    QWORD PTR [rbp-0x10],rsi
(gdb) si
0x0000555555555190 in main ()
1: x/i $pc
=> 0x555555555190 <main+19>:    cmp    DWORD PTR [rbp-0x4],0x1
(gdb) x /dx $rbp-4
0x7fffffffe4cc: 0x01
(gdb) x /wx $rbp-4
0x7fffffffe4cc: 0x00000001
(gdb) i r $eflags
eflags         0x206    [ PF IF ]
(gdb) si
0x0000555555555194 in main ()
1: x/i $pc
=> 0x555555555194 <main+23>:    jle    0x5555555551ab <main+46>
(gdb) i r $eflags
eflags         0x246    [ PF ZF IF ]
(gdb)
```

gdb - Gnu Debugger

A look around

- Single step and you see that the jump is taken.
- The next instruction show an interesting way to address a section in memory called “rip-relative addressing”
- It happens when you need to access something where the address is unknown at compile time but the location relative to the instruction is known.

```
0x0000555555555518c in main ()
1: x/i $pc
=> 0x55555555518c <main+15>:    mov    QWORD PTR [rbp-0x10],rsi
(gdb) si
0x0000555555555190 in main ()
1: x/i $pc
=> 0x555555555190 <main+19>:    cmp    DWORD PTR [rbp-0x4],0x1
(gdb) x /dx $rbp-4
0x7fffffffe4cc: 0x01
(gdb) x /wx $rbp-4
0x7fffffffe4cc: 0x00000001
(gdb) i r $eflags
eflags         0x206    [ PF IF ]
(gdb) si
0x0000555555555194 in main ()
1: x/i $pc
=> 0x555555555194 <main+23>:    jle    0x5555555551ab <main+46>
(gdb) i r $eflags
eflags         0x246    [ PF ZF IF ]
(gdb) si
0x00005555555551ab in main ()
1: x/i $pc
=> 0x5555555551ab <main+46>:    lea    rdi,[rip+0xe5e]        # 0x555555556010
(gdb)
```

gdb - Gnu Debugger

A look around

- Single step and let's see what rdi is pointing to.
- We are about to execute a call instruction. Can someone explain what this instruction will do?
- What will the return address be?

```
=> 0x555555555190 <main+19>:    cmp    DWORD PTR [rbp-0x4],0x1
(gdb) x /dx $rbp-4
0x7fffffffe4cc: 0x01
(gdb) x /wx $rbp-4
0x7fffffffe4cc: 0x00000001
(gdb) i r $eflags
eflags         0x206    [ PF IF ]
(gdb) si
0x0000555555555194 in main ()
1: x/i $pc
=> 0x555555555194 <main+23>:    jle    0x5555555551ab <main+46>
(gdb) i r $eflags
eflags         0x246    [ PF ZF IF ]
(gdb) si
0x00005555555551ab in main ()
1: x/i $pc
=> 0x5555555551ab <main+46>:    lea    rdi,[rip+0xe5e]        # 0x555555556010
(gdb) si
0x00005555555551b2 in main ()
1: x/i $pc
=> 0x5555555551b2 <main+53>:    call   0x555555555149 <print_string>
(gdb) x /s $rdi
0x555555556010: "Hello World!"
(gdb)
```

gdb - Gnu Debugger

A look around

- 0x5555555551b7
- ‘bt’ backtrace shows the call stack
- Single step until you reach the call to printf.
- Look at the function arguments. What are they?

```
1: x/i $pc
=> 0x555555555194 <main+23>:    jle    0x5555555551ab <main+46>
(gdb) i r $eflags
eflags         0x246    [ PF ZF IF ]
(gdb) si
0x00005555555551ab in main ()
1: x/i $pc
=> 0x5555555551ab <main+46>:    lea    rdi,[rip+0xe5e]        # 0x555555556010
(gdb) si
0x00005555555551b2 in main ()
1: x/i $pc
=> 0x5555555551b2 <main+53>:    call   0x555555555149 <print_string>
(gdb) x /s $rdi
0x555555556010: "Hello World!"
(gdb) si
0x0000555555555149 in print_string ()
1: x/i $pc
=> 0x555555555149 <print_string>:       endbr64
(gdb) x /gx $rsp
0x7fffffffe4b8: 0x00005555555551b7
(gdb) bt
#0  0x0000555555555149 in print_string ()
#1  0x00005555555551b7 in main ()
(gdb)
```

gdb - Gnu Debugger

A look around

```
0x0000555555555160 in print_string ()
1: x/i $pc
=> 0x555555555160 <print_string+23>:    mov    rax,QWORD PTR [rbp-0x8]
(gdb)
0x0000555555555164 in print_string ()
1: x/i $pc
=> 0x555555555164 <print_string+27>:    mov    rsi,rax
(gdb)
0x0000555555555167 in print_string ()
1: x/i $pc
=> 0x555555555167 <print_string+30>:    lea    rdi,[rip+0xe96]        # 0x555555556004
(gdb)
0x000055555555516e in print_string ()
1: x/i $pc
=> 0x55555555516e <print_string+37>:    mov    eax,0x0
(gdb)
0x0000555555555173 in print_string ()
1: x/i $pc
=> 0x555555555173 <print_string+42>:    call   0x555555555050 <printf@plt>
(gdb) x /s $rdi
0x555555556004: "String: %s\n"
(gdb) x /s $rsi
0x555555556010: "Hello World!"
(gdb)
```

gdb - Gnu Debugger

A look around

- We don’t want to go into printf so instead of using “single step” use “ni” for next instruction
- This command “skips” the function call.
- It actually just executes it and stops after it returns.

```
1: x/i $pc
=> 0x555555555164 <print_string+27>:    mov    rsi,rax
(gdb)
0x0000555555555167 in print_string ()
1: x/i $pc
=> 0x555555555167 <print_string+30>:    lea    rdi,[rip+0xe96]        # 0x555555556004
(gdb)
0x000055555555516e in print_string ()
1: x/i $pc
=> 0x55555555516e <print_string+37>:    mov    eax,0x0
(gdb)
0x0000555555555173 in print_string ()
1: x/i $pc
=> 0x555555555173 <print_string+42>:    call   0x555555555050 <printf@plt>
(gdb) x /s $rdi
0x555555556004: "String: %s\n"
(gdb) x /s $rsi
0x555555556010: "Hello World!"
(gdb) ni
String: Hello World!
0x0000555555555178 in print_string ()
1: x/i $pc
=> 0x555555555178 <print_string+47>:    jmp    0x55555555517b <print_string+50>
(gdb)
```

gdb - Gnu Debugger

A look around

- We made it through a single concrete execution
- Let’s add a command line argument.
- Type ‘r “This is a single argument”’
- Enter ‘y’ when it asks if you are sure.

```
(gdb)
0x00005555555551bc in main ()
1: x/i $pc
=> 0x5555555551bc <main+63>:    leave
(gdb)
0x00005555555551bd in main ()
1: x/i $pc
=> 0x5555555551bd <main+64>:    ret
(gdb)
__libc_start_main (main=0x55555555517d <main>, argc=1, argv=0x7fffffffe5b8, init=<optimized out>,
    fini=<optimized out>, rtld_fini=<optimized out>, stack_end=0x7fffffffe5a8)
    at ../csu/libc-start.c:344
344     ../csu/libc-start.c: No such file or directory.
1: x/i $pc
=> 0x7ffff7a03bf7 <__libc_start_main+231>:      mov    edi,eax
(gdb) r "This is a single argument"
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /thayerfs/home/f00543d/recourse/hello_world "This is a single argument"

Breakpoint 1, 0x000055555555517d in main ()
1: x/i $pc
=> 0x55555555517d <main>:       endbr64
(gdb)
```

gdb - Gnu Debugger

A look around

- Take some time and do the same thing that we did before.
- Look at the flags before and after the CMP.
- Look at the return value of print_string
- Check out the arguments of printf()

gdb - Gnu Debugger

Memory Regions

- Make sure that you are still in gdb with a running application
- Run ‘info proc mappings’
- Look for [stack] and [heap]

```
process 37716
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555555000     0x555555556000     0x1000     0x1000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555556000     0x555555557000     0x1000     0x2000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555557000     0x555555558000     0x1000     0x2000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555558000     0x555555559000     0x1000     0x3000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x7ffff79e2000     0x7ffff7bc9000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7bc9000     0x7ffff7dc9000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dc9000     0x7ffff7dcd000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcd000     0x7ffff7dcf000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcf000     0x7ffff7dd3000     0x4000        0x0
      0x7ffff7dd3000     0x7ffff7dfc000    0x29000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7fa6000     0x7ffff7fa8000     0x2000        0x0
      0x7ffff7ff5000     0x7ffff7ff7000     0x2000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
---Type <return> to continue, or q <return> to quit---
```

gdb - Gnu Debugger

Memory Regions

- Make sure that you are still in gdb with a running application
- Run ‘info proc mappings’
- Look for [stack].
- The pointer in rsp should be within this range.

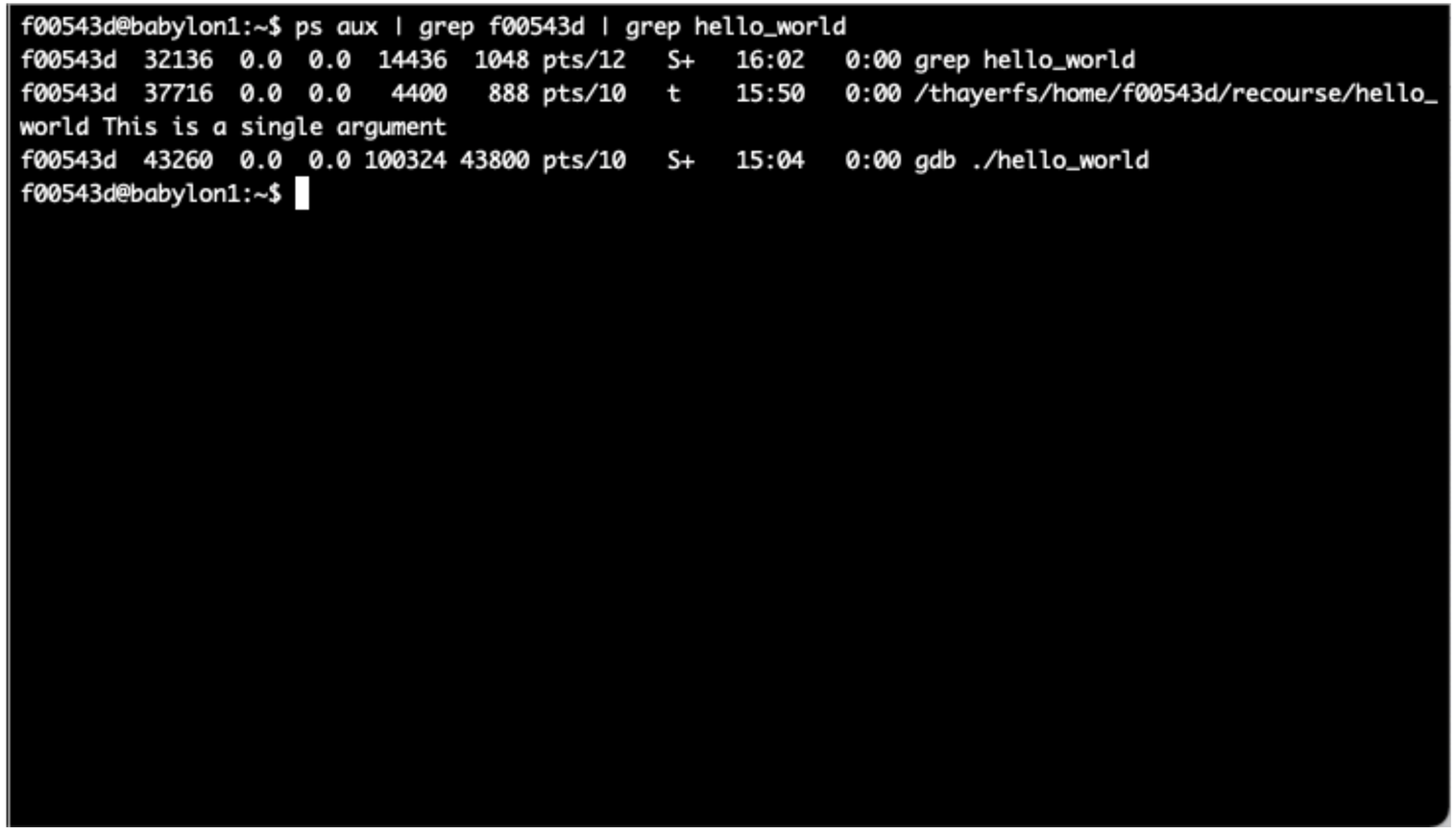
```
process 37716
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555555000     0x555555556000     0x1000     0x1000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555556000     0x555555557000     0x1000     0x2000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555557000     0x555555558000     0x1000     0x2000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x555555558000     0x555555559000     0x1000     0x3000 /thayerfs/home/f00543d/recourse/hello_wor
ld
      0x7ffff79e2000     0x7ffff7bc9000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7bc9000     0x7ffff7dc9000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dc9000     0x7ffff7dcd000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcd000     0x7ffff7dcf000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcf000     0x7ffff7dd3000     0x4000        0x0
      0x7ffff7dd3000     0x7ffff7dfc000    0x29000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7fa6000     0x7ffff7fa8000     0x2000        0x0
      0x7ffff7ff5000     0x7ffff7ff7000     0x2000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
---Type <return> to continue, or q <return> to quit---
```

/proc

Memory Regions

- /proc is a file system that contains process information
- Open another terminal on the same babylon server.
- Run ‘ps -aux | grep <netid> | grep hello_world’ to get the process id



/proc

Memory Regions

- Run ‘cat /proc/<pid>/maps | less’
- You get mostly the same information but the interesting bits are the permissions.
- r: read
- w: write
- x: execute
- p: private (copy on write)
- https://www.kernel.org/doc/html/latest/filesystems/proc.html

```
555555554000-555555555000 r--p 00000000 00:38 23537167833                /thayerfs/home/f00543d/recours
e/hello_world
555555555000-555555556000 r-xp 00001000 00:38 23537167833                /thayerfs/home/f00543d/recours
e/hello_world
555555556000-555555557000 r--p 00002000 00:38 23537167833                /thayerfs/home/f00543d/recours
e/hello_world
555555557000-555555558000 r--p 00002000 00:38 23537167833                /thayerfs/home/f00543d/recours
e/hello_world
555555558000-555555559000 rw-p 00003000 00:38 23537167833                /thayerfs/home/f00543d/recours
e/hello_world
7ffff79e2000-7ffff7bc9000 r-xp 00000000 09:7e 1311323                    /lib/x86_64-linux-gnu/libc-2.2
7.so
7ffff7bc9000-7ffff7dc9000 ---p 001e7000 09:7e 1311323                    /lib/x86_64-linux-gnu/libc-2.2
7.so
7ffff7dc9000-7ffff7dcd000 r--p 001e7000 09:7e 1311323                    /lib/x86_64-linux-gnu/libc-2.2
7.so
7ffff7dcd000-7ffff7dcf000 rw-p 001eb000 09:7e 1311323                    /lib/x86_64-linux-gnu/libc-2.2
7.so
7ffff7dcf000-7ffff7dd3000 rw-p 00000000 00:00 0
7ffff7dd3000-7ffff7dfc000 r-xp 00000000 09:7e 1310773                    /lib/x86_64-linux-gnu/ld-2.27.
so
7ffff7fa6000-7ffff7fa8000 rw-p 00000000 00:00 0
7ffff7ff5000-7ffff7ff7000 rw-p 00000000 00:00 0
:
```

gdb - Gnu Debugger

On your own

- At this point you have the basic knowledge of how to dynamically reverse engineer an application
- There are more things to learn that are helpful
  - Conditional breakpoints
  - Modifying memory/registers
  - Scripting
  - I like using “gef” to make gdb a bit more useful

gdb - Gnu Debugger

Lab

- Open week2_day1_lab in Ghidra
- You can also use all the other tools that we have discussed
- What inputs are required to have the “SUCCESS!” string printed?

Homework

- Use the tools you know to figure out valid inputs
- This is an example of what we call CrackMes.
- They are designed to “legally” break software serial algorithms.
- For example, Windows required a serial number to verify that you had purchased the software. However, there existed an algorithm to verify what you entered. What if you just reverse engineered the algorithm and wrote some code that automatically generated serial numbers? Don’t do this.
- CrackMes allow you to scratch the itch of breaking something but in a way where the feds won’t show up at your door. :)
- There also exist legal ways to hack into servers that we will discuss at the end of the course.

Day 2 Homework

- You will be compiling and looking at the disassembly of 3 C programs
- If your machine isn't x86 you can log into the babylon servers and use those for compiling