Dynamic Analysis

Debuggers and Emulators

DawgCTF

I know we’ve been harping on this a lot

March 11th! Sign up!

Lunch and dinner! Prizes! Cool challenges!

Dynamic Analysis

Up until now, we haven’t run the binaries we’re reversing (other than to check the solution)

With dynamic analysis, we have the tools to change that!

Dynamic analysis is about running programs in some way and reasoning about their state at some point in the program

A few tools you can use - mainly, debuggers and instrumentation frameworks

Uses for a debugger

- With a debugger, we can get the contents of registers and memory at any instruction in the program
    - This can be extremely helpful for figuring out how some assembly snippet works
    - It can also let you bypass static analysis: sometimes the program decrypts the data you want for you!
- We can also change the contents of memory and registers
    - This can let you skip some useless checks
    - It also helps you enter input into a program that takes it's input from a contrived way (eg, RF transmissions)

Debuggers you can use

IDA Pro debugger - download the demo and you can use it on 32 bit binaries

GDB - it sucks, but it’s the standard for linux stuff

EDB - graphical debugger for Linux that is sort of old

Vivisect/VDB - Never used but looks cool

On Windows: x64dbg, Windbg, Cheat Engine

OSX: lldb

GDB

- Generally, the debugger of choice for Linux debugging
- Designed for debugging programs that you wrote, has some weird limitations when reverse engineering
- Console program, but graphical front ends do exist
- Several extensions to make life easier for reverse engineers exist:
    - Gdb-peda: Python Exploit Development Assistance
        - Easiest to use
    - GEF: GDB Enhanced Features
    - Pwndbg: Windbg compatibility layer and other stuff
        - My favorite, also meshes with pwntools
    - Voltron - “graphical” frontend for GDB
        - Binjatron - lets you control GDB from inside Binja! (sort of broken)

GDB cheat sheet

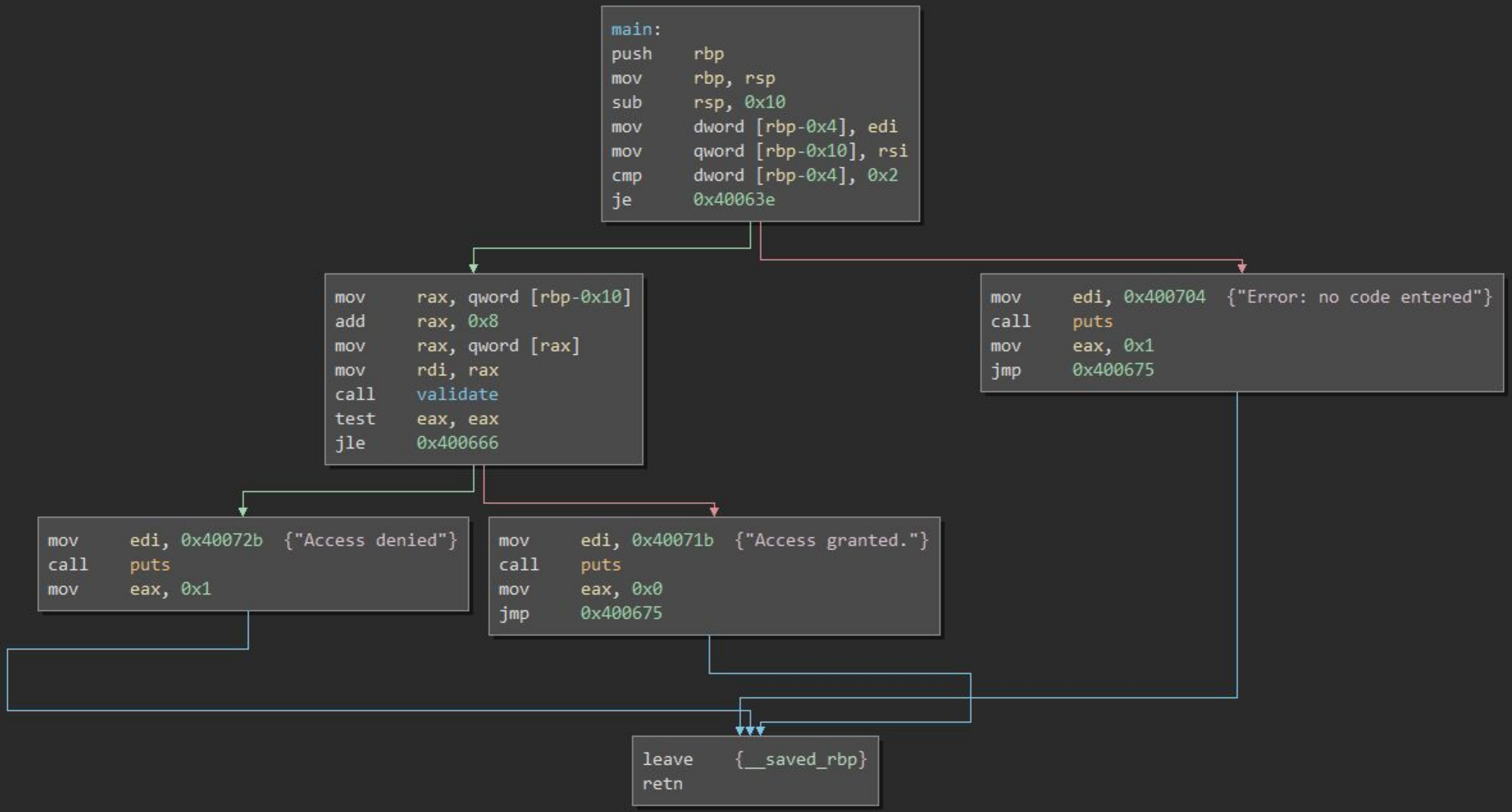
- gdb some_program
  - run - start the program
  - break *0xdeadbeef - breakpoint on some address
  - b main - break on some symbol
  - set disassembly-flavor intel
  - disassemble main (disas main) - print out a linear disassembly ala objdump
  - stepi (s) - step into
  - nexti (n) - step over, skip calls
  - info registers (i r) - Get information about registers at current instruction
  - print/x $rdi - Print out the contents of just rdi (or any other register)

GDB - looking at memory:

- Examine memory: x/NFU address
  - N = number of things to display
  - F = format (x = hexadecimal, d = decimal)
  - U = units (w = word, b = byte, dw = dword
- Examples:
  - x/10xb 0xdeadbeef, examine 10 bytes in hex
  - x/xw 0xdeadbeef, examine 1 word in hex
  - x/s 0xdeadbeef, examine null terminated string

GDB Demo!

Chris shows you how you can use GDB to make reversing stuff really easy



```
main:
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
mov     dword [rbp-0x4], edi
mov     qword [rbp-0x10], rsi
cmp     dword [rbp-0x4], 0x2
je      0x40063e
```

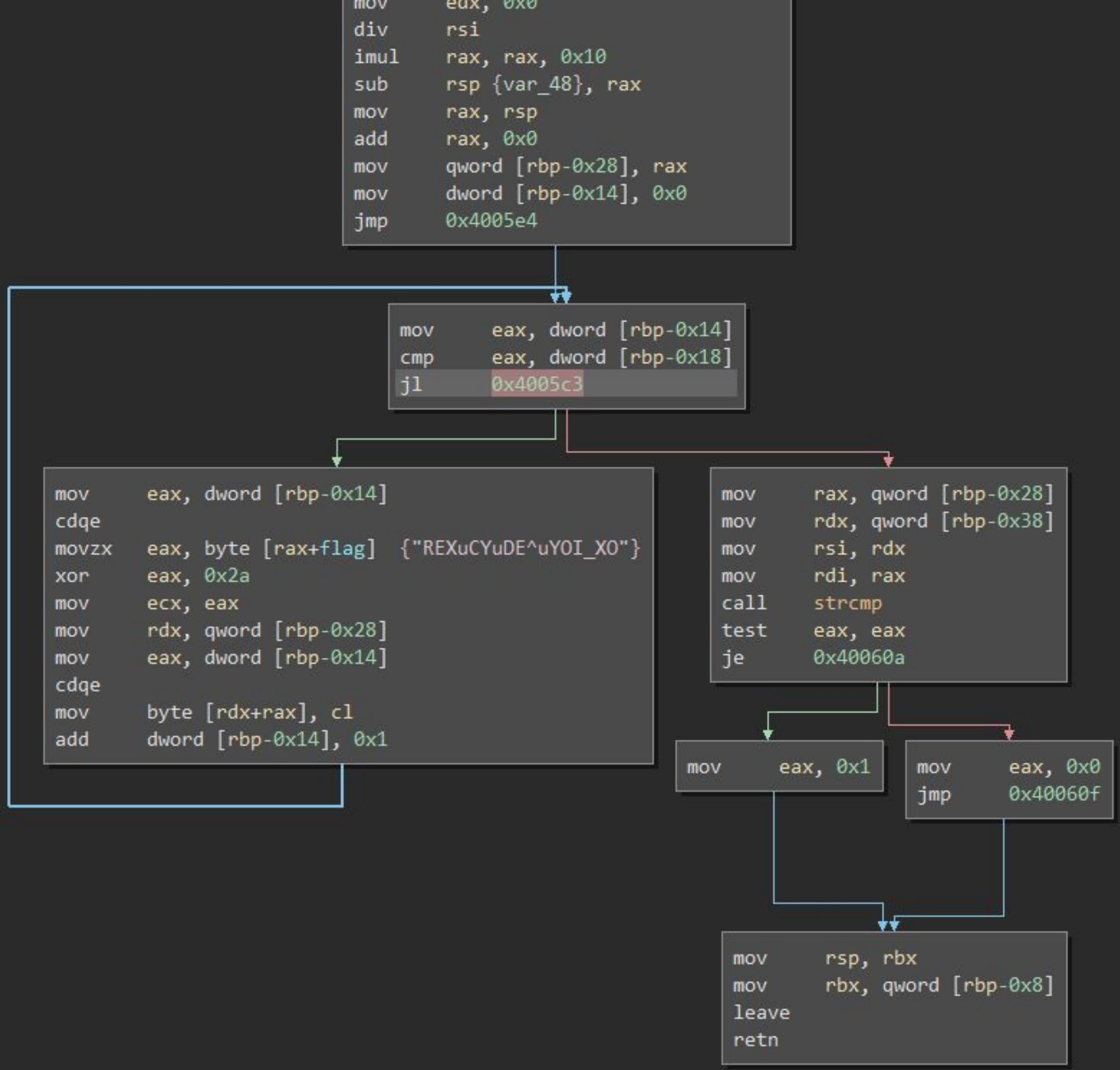
```
mov     rax, qword [rbp-0x10]
add     rax, 0x8
mov     rax, qword [rax]
mov     rdi, rax
call    validate
test    eax, eax
jle     0x400666
```

```
mov     edi, 0x400704  {"Error: no code entered"}
call    puts
mov     eax, 0x1
jmp     0x400675
```

```
mov     edi, 0x40072b  {"Access denied"}
call    puts
mov     eax, 0x1
```

```
mov     edi, 0x40071b  {"Access granted."}
call    puts
mov     eax, 0x0
jmp     0x400675
```

```
leave    {__saved_rbp}
retn
```



```
mov     edx, 0x0
div     rsi
imul    rax, rax, 0x10
sub     rsp {var_48}, rax
mov     rax, rsp
add     rax, 0x0
mov     qword [rbp-0x28], rax
mov     dword [rbp-0x14], 0x0
jmp     0x4005e4
```

```
mov     eax, dword [rbp-0x14]
cmp     eax, dword [rbp-0x18]
jl      0x4005c3
```

```
mov     eax, dword [rbp-0x14]
cdqe
movzx   eax, byte [rax+flag]  {"REXuCYuDE^uYOI_XO"}
xor     eax, 0x2a
mov     ecx, eax
mov     rdx, qword [rbp-0x28]
mov     eax, dword [rbp-0x14]
cdqe
mov     byte [rdx+rax], cl
add     dword [rbp-0x14], 0x1
```

```
mov     rax, qword [rbp-0x28]
mov     rdx, qword [rbp-0x38]
mov     rsi, rdx
mov     rdi, rax
call    strcmp
test    eax, eax
je      0x40060a
```

```
mov     eax, 0x1
```

```
mov     eax, 0x0
jmp     0x40060f
```

```
mov     rsp, rbx
mov     rbx, qword [rbp-0x8]
leave
retn
```

Other Dynamic Analysis tools

Tracers - strace, ltrace

Emulators - QEMU

Instrumentation frameworks - PANDA, PIN

Emulators with instrumentation capabilities - Unicorn, Usercorn

“Timeless” debuggers - Qira, rr

Tracers

Strace and ltrace run the program and show all system calls/library calls

Strace works on everything, ltrace only on dynamically linked binaries

Can be helpful if the thing you’re reversing is obfuscated but makes a lot of calls to external things

QEMU

- QEMU is an emulator that lets you run programs that were compiled for a different architecture, and debug them
- Two modes:
- qemu-static: Emulate an entire system, debug it
- qemu-user: Run a binary written for another architecture (or the same one), translate syscalls to native syscall ABI
    - Generally much more useful for reverse engineering
    - Pretty much essential for REing Linux programs written for ARM/etc
    - Can debug binaries being run under it with -g

PANDA

Platform for Architecture Neutral Dynamic Analysis

Emulates an entire system and records a trace, letting you replay the trace and debug at any point

Sort of similar to timeless debugging

https://github.com/panda-re/panda

PIN

PIN is a tool, made by Intel, that lets you instrument programs without having to emulate them (all the other instrumentation tools here are emulation based)

Give it an address and some callback code, PIN will automatically run your code (in a performant way) when that address is reached.

This is the way to do instrumentation at scale

Unicorn

Have you ever wanted to just run a small snippet of assembly code to see what it does, or instrument it? Have you ever wanted to do that without having to deal with coercing a binary into running your code?

Unicorn can do all that! And it’s got Python bindings!

There’s a Binary Ninja plugin called ripr that will ‘export’ code from binja into a python script that uses Unicorn.

https://github.com/unicorn-engine/unicorn

https://github.com/pbiernat/ripr

Usercorn

Like Usercorn, but lets you emulate/instrument entire binaries (simulating the ABI and everything), instead of just assembly snippets

Will emulate the kernel and userspace interfaces for you, and let you ‘call into’ a binary (call certain functions only)

Also includes a debugger of sorts

Project is slightly abandoned but it’s really cool

https://github.com/lunixbochs/usercorn

Timeless debugging

Often when debugging we get somewhere and realize we needed some information that is easily found at a previously executed instruction. Normally you just have to restart your debugging session

With timeless debugging you can ‘rewind’ execution to that state, get the info you need, and jump back to the current instruction

Qira (https://qira.me) does this, although its sort of unuseable. Windbg (on windows) can do it.

rr is a cool framework from Mozilla that lets you do this

Lab?

Unfortunately due to the upcoming CTF I wasn’t able to prepare any new challenges for today, however the old challenges are still up and I encourage you to take a look at them with GDB, to see if they are any easier.