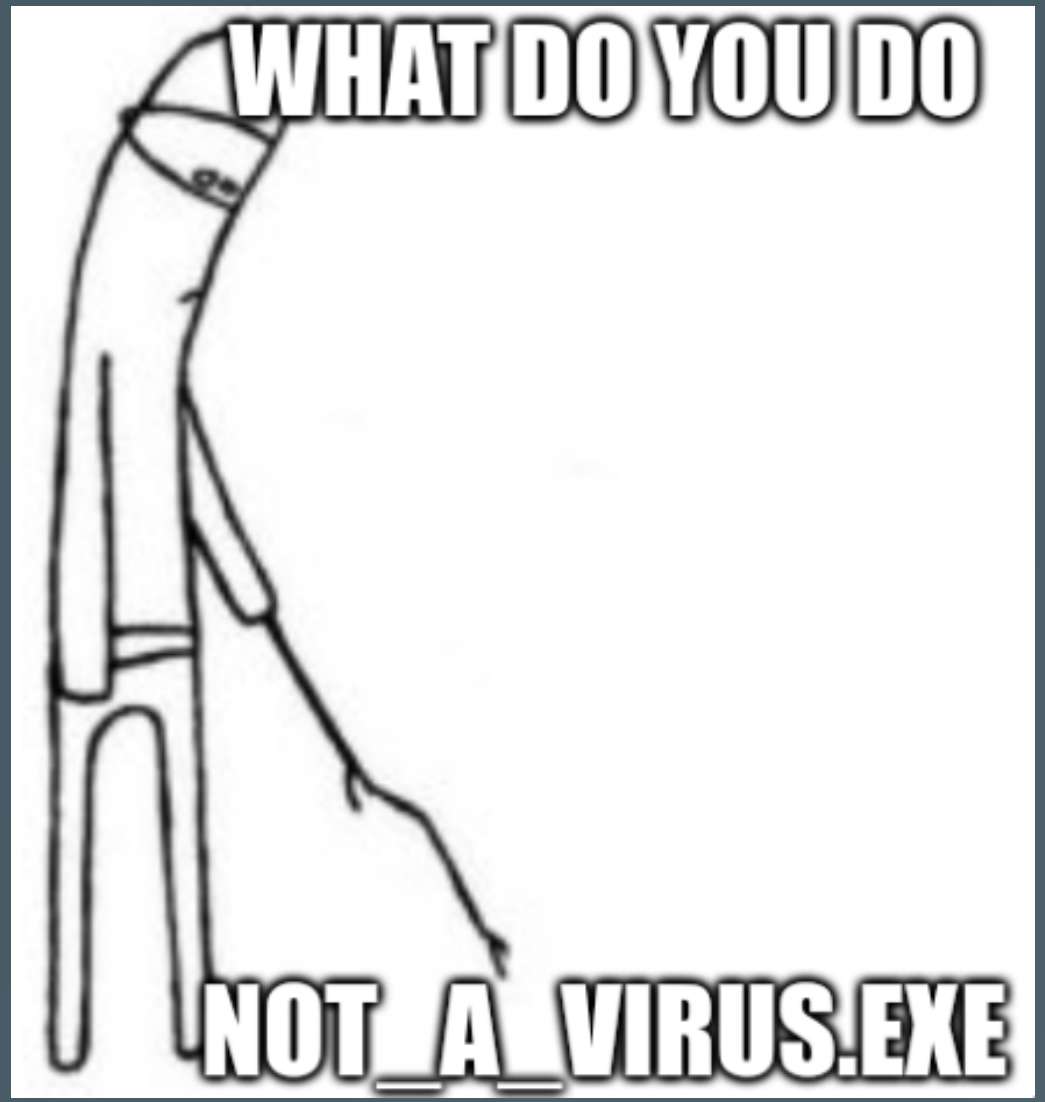
Reverse Engineering 101

As I like to call it:



Assembly Recap

CPUs work with registers and memory
x86-64 has many registers such as

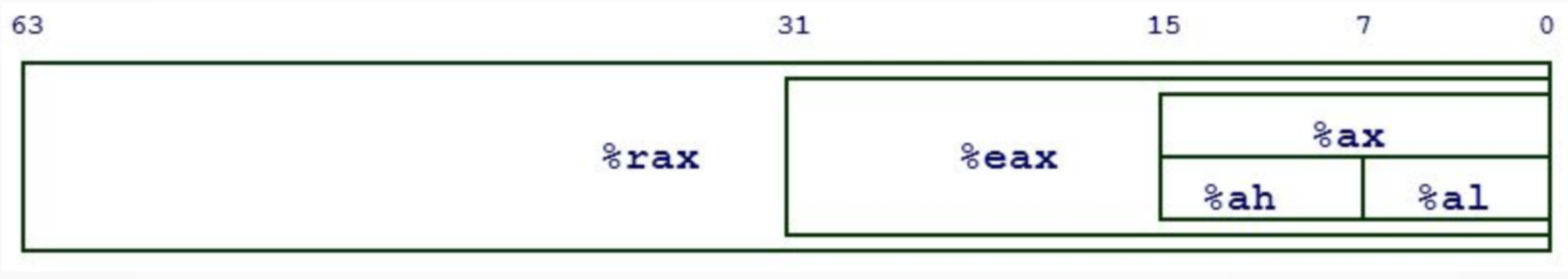
`rax, rbx, rcx, rdx, rdi, rsi, rsp, rip, r8-r15`

Special registers:

`rip` : Current instruction pointer
`rsp` : Current stack pointer
`rbp` : Stack frame base pointer
`cr3` : Virtual memory selector for a process

Assembly Recap

We can access certain bits of registers individually:



1

This allows for backwards compatability
32-bit programs can just use `eax`

Assembly Recap

Instruction format:

`<instruction_mnemonic> <destination>, <source>`

`mov rax, rbx`

-> Means move rbx to rax

-> The compiler turns assembly into actual opcodes

`mov rax, rbx` => 0x48, 0x89, 0xd8

Online (dis)assembler: https://defuse.ca/online-x86-assembler.htm#disassembly

Assembly Recap

Data Movement

`mov rax, rbx` Moves rbx into rax

`mov rax, 0x4000` Moves 0x4000 into rax

`mov rax, [rbx]` Moves the 8-byte value at the address of rbx into rax

=> rbx = 0x400000 `mov rax, [0x400000]`

=> `rax = *0x400000;`

Assembly Recap

Arithmetics

`add rax, rbx`	Adds rbx to rax
`sub rax, rbx`	Substracts rbx from rax
`xor rax, rbx`	…
`and rax, rbx`	…

…

Assembly Recap

Control Flow

`call function` Calls a function

`ret` Returns from a function to the next instruction

Example:

`call target`

=> `mov rax, 3`

=> `ret`

`mov rbx, rax` <=

rbx = 3

Assembly Recap

Control Flow

`jmp address` Always jump to address

`jnz address` Jump if not zero

`je address` Jump if equal

`jle address` Jump if less or equal

=> Based on EFLAGS (special registers)

Assembly Recap

Control Flow

```
cmp rax, rbx
jle error
ret
```

Jump to error IF RAX <= RBX
Otherwise return from the function

C to assembly

int x;
x = 10;

```
mov rax, 10
```

Not every C line is atomic in asm:

x = x + 10;

```
mov rbx, rax // temporary value
add rbx, 10  // add 10
mov rax, rbx // move temp back to x
```

C to assembly

C to asm in the browser:
https://godbolt.org/#

```c
#include <stdio.h>
int main()
{
    printf("Hello, World!");
    return 0;
}
```

```
.LC0:
        .string "Hello, World!"
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

Done by https://defuse.ca/online-x86-assembler.htm#disassembly

Assembly

Raw Hex (zero bytes in bold):

554889E5BF00608000B800000000E800000000B800000000**00**5DC3

String Literal:

"\x55\x48\x89\xE5\xBF\x00\x60\x80\x00\xB8\x00\x00\x00\x00\xE8\x00\x00\x00\x00\x

Array Literal:

```
{ 0x55, 0x48, 0x89, 0xE5, 0xBF, 0x00, 0x60, 0x80, 0x00, 0xB8, 0x00, 0x00, 0x00,
0x00, 0xE8, 0x00, 0x00, 0x00, 0x00, 0xB8, 0x00, 0x00, 0x00, 0x00, 0x5D, 0xC3 }
```

Disassembly:

```
0:  55                      push   rbp
1:  48 89 e5                mov    rbp,rsp
4:  bf 00 60 80 00          mov    edi,0x806000
9:  b8 00 00 00 00          mov    eax,0x0
e:  e8 00 00 00 00          call   13 <_main+0x13>
13: b8 00 00 00 00          mov    eax,0x0
18: 5d                      pop    rbp
19: c3                      ret
```

Rev 101

- Analysis of a system, program or (obfuscated) source code
  - Often binary analysis
- Find out what it’s doing
  - Revertible, Exploitable?

Real world usage

- Malware research
- Bug hunting in consumer software & operating systems
- Modding games
  - Cracking
- Debugging

Executables

- ELF
  - Executable and Linking Format (UNIX)
- PE
  - Portable Executable (WINDOWS)
- Tells our OS how to load and execute it
- Contains Imports (Libraries), Exports, Sections, Entrypoint

Tools for executables

- UNIX:
  - file: Tries to determine the filetype
  - strings: Print all ascii strings in the file
  - hexdump: See raw bytes of the file
  - readelf: Parses the elf file and prints info
  - objdump: ELF infos & disassembly
- Windows:
  - CFF Explorer/ Explorer Suite by NTCore

Concepts

- Type of with the binary
  - Static
  - Dynamic
- Emulation/Tracing
- Diffing
- Patching
- Sidechannels
- Symbolic execution

Static analysis

- "Offline" analysis
  - Binary is not executed
- Disassembler
  - Turns opcodes into asm instructions
  - `68 6e 2f 73 68` => `push 0x68732f6e`
- Decompiler
  - Turn asm instructions into somewhat readable code

Static tools

- Native binaries:
  - Ghidra (Free, works well on most arches + languages)
    - Gui sucks => Cutter Plugin
  - IDA: Gold standard for x86, okayish on other arches
  - BinaryNinja: Mix of IDA and Ghidra
    - Especially good for newer languages such as Go and Rust

Static tools cont.

- Python:
  - Pyinstxtractor
    - Extract bundled python files
  - Pycdc
    - Disassemble/Decompile python bytecode

Static tools cont.

- Android APKs
  - Essentially Java
  - Jadx: GUI for apktool essentially
  - apktool: CLI to decompile/compile apks
  - github/patrickfav/uber-apk-signer: Automatically sign apks

Static tools cont.

- .NET
  - DotPeek: Disassembler/Decompiler for .NET
  - ILSpy: Same as above
  - github/Droppers/SingleFileExtractor: Extract .NET from native libraries

Dynamic analysis

- Run/emulate the binary and attach a debugger/tracer
- Breakpoints
  - Addresses in memory where execution shall be paused
    - PAUSE = rip == TARGET
  - Prints infos about current registers/memory
  - Static analysis to find breakpoints
- Single stepping / tracing
  - One instruction at a time, print infos

Dynamic tools

- Native:
  - strace/ltrace: Traces syscalls/library calls
  - GDB
    - pwndbg, gef
  - Emulators
    - QEMU
    - qiling
  - Inbuilt debuggers of decompilers
    - Supports breaking in pseudocode

Dynamic tools cont.

- Android APKs:
  - Android Studio for emulation
  - FRIDA
- .NET
  - JetBrains RIDER
    - Supports binary debugging
    - Disassembles automatically

"Just run it lmao" - analysis

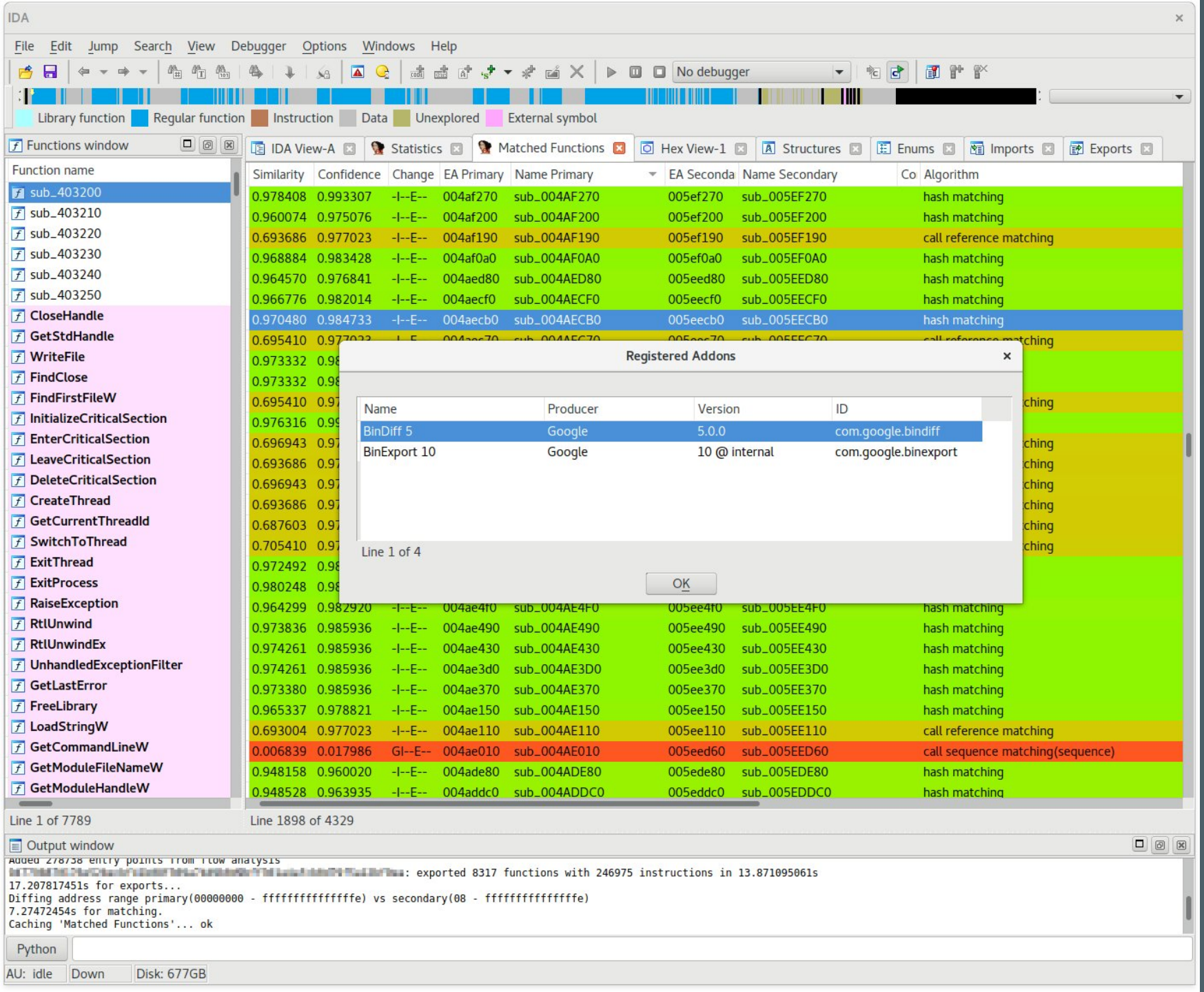
- Running unknown executables
  - **Bad idea**
- Even dockerfiles can be malicious
  - Insomnihack 23 (https://cryptax.github.io/2023/03/25/shame.html)
- Always emulate unknown binaries or use a VM

"Just run it lmao" - done right

- Emulation
  - Works cross OS
- Full system emulation
  - Qiling, QEMU System/Usermode
- Instruction emulation
  - **No syscall support**
  - e.g. Unicorn Engine
  - Lots of manual work

Diffing

- Prerequisite: Static analysis
- Needs 2+ program databases (e.g. from IDA)
- BinDiff databases
  - Find matching functions/patterns
  - See newly added functions


IDA
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol
Functions window
Function name
sub_403200
sub_403210
sub_403220
sub_403230
sub_403240
sub_403250
CloseHandle
GetStdHandle
WriteFile
FindClose
FindFirstFileW
InitializeCriticalSection
EnterCriticalSection
LeaveCriticalSection
DeleteCriticalSection
CreateThread
GetCurrentThreadId
SwitchToThread
ExitThread
ExitProcess
RaiseException
RtlUnwind
RtlUnwindEx
UnhandledExceptionFilter
GetLastError
FreeLibrary
LoadStringW
GetCommandLineW
GetModuleFileNameW
GetModuleHandleW
IDA View-A
Statistics
Matched Functions
Hex View-1
Structures
Enums
Imports
Exports
Similarity Confidence Change EA Primary Name Primary EA Seconda Name Secondary Co Algorithm
0.978408 0.993307 -I--E-- 004af270 sub_004AF270 005ef270 sub_005EF270 hash matching
0.960074 0.975076 -I--E-- 004af200 sub_004AF200 005ef200 sub_005EF200 hash matching
0.693686 0.977023 -I--E-- 004af190 sub_004AF190 005ef190 sub_005EF190 call reference matching
0.968884 0.983428 -I--E-- 004af0a0 sub_004AF0A0 005ef0a0 sub_005EF0A0 hash matching
0.964570 0.976841 -I--E-- 004aed80 sub_004AED80 005eed80 sub_005EED80 hash matching
0.966776 0.982014 -I--E-- 004aecf0 sub_004AECF0 005eecf0 sub_005EECF0 hash matching
0.970480 0.984733 -I--E-- 004aecb0 sub_004AECB0 005eecb0 sub_005EECB0 hash matching
0.973836 0.985936 -I--E-- 004ae490 sub_004AE490 005ee490 sub_005EE490 hash matching
0.974261 0.985936 -I--E-- 004ae430 sub_004AE430 005ee430 sub_005EE430 hash matching
0.974261 0.985936 -I--E-- 004ae3d0 sub_004AE3D0 005ee3d0 sub_005EE3D0 hash matching
0.973380 0.985936 -I--E-- 004ae370 sub_004AE370 005ee370 sub_005EE370 hash matching
0.965337 0.978821 -I--E-- 004ae150 sub_004AE150 005ee150 sub_005EE150 hash matching
0.693004 0.977023 -I--E-- 004ae110 sub_004AE110 005ee110 sub_005EE110 call reference matching
0.006839 0.017986 GI--E-- 004ae010 sub_004AE010 005eed60 sub_005EED60 call sequence matching(sequence)
0.948158 0.960020 -I--E-- 004ade80 sub_004ADE80 005ede80 sub_005EDE80 hash matching
0.948528 0.963935 -I--E-- 004addc0 sub_004ADDC0 005eddc0 sub_005EDDC0 hash matching
Registered Addons
Name Producer Version ID
BinDiff 5 Google 5.0.0 com.google.bindiff
BinExport 10 Google 10 @ internal com.google.binexport
Line 1 of 4
OK
Line 1 of 7789
Line 1898 of 4329
Output window
: exported 8317 functions with 246975 instructions in 13.871095061s
17.207817451s for exports...
Diffing address range primary(00000000 - fffffffffffffffe) vs secondary(08 - fffffffffffffffe)
7.27472454s for matching.
Caching 'Matched Functions'... ok
Python
AU: idle Down Disk: 677GB

Patching

- Modify instructions to get different behaviour
  - e.g. `jnz address` => `jz address`
- Remove instructions by using NOPs
  - `mov eax, ebx` => `nop nop`
- Used to bypass checks or security
- What happens if we leak some infos by doing this?

Sidechannels

- Leak infos
- Bruteforce inputs much faster e.g 26*6 instead of 26**6
- Timing attacks
  or
- CPU metric attacks
  - perf-tools on Linux

Symbolic execution

- Execute a program
- Find all paths and values that satisfy each branching condition
- Output inputs that satisfy certain branches

Given this function, how many paths are there?

```
int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;
  else
    return 1;
}
```

Three branching conditions, which inputs satisfy each path?

```
int get_sign(int x) {
  if (x == 0)
    ...
  if (x < 0)
  ...
  if (x > 0)
  ...
}
```

Symbolic execution tools

- angr
  - Black box (works on binary level)
- klee
  - White box (requires source code)
- manticore (unmaintained)
  - Like angr black box, requires more fine tuning

How2Start

1. Run strings and gather infos about the binary
2. What's the goal?
    - Want a key/input?
    - Optimization problem?
3. Optional: Can we cheese it?
    - Sidechannels? Do we have an oracle?
    - Symbolic execution
    - Patching or info leaks?

How2Start cont.

4. *Actually* reverse the binary and figure out the *actual* solution
5. ???
6. Validate solution

Live demo - cracking

References

https://ike.mahaloz.re/
https://godbolt.org/
https://dogbolt.org/
https://defuse.ca/online-x86-assembler.htm#disassembly
https://ctf101.org/