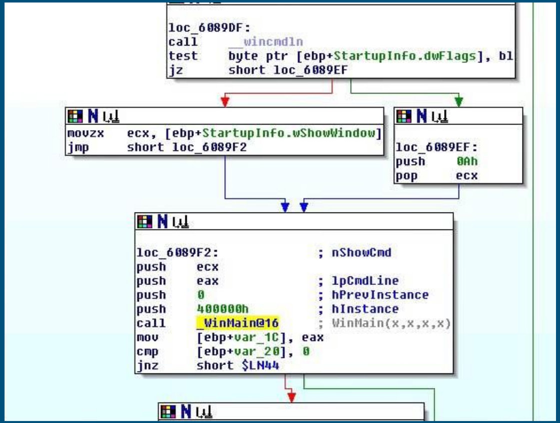
Introduction to reverse engineering



x86 / x64

Summary

- A little bit of theory
- Static analysis
- 1st exercise and solution
- Debugging (dynamic analysis)
- 2nd hands-on and solution
- Last tricks

Whoami: @0xdidu

Work-wise:

- Started as a developer at Microsoft
- Security consultant for about 5 years (dev: 3.5 and reverse engineer: 1.5)
- Now a Security Engineer at Google

And apart from that:

- Reverse engineering enthusiast, interested in low-level layers and a Windows fan
- Proud member of BlackHoodie (https://blackhoodie.re)
- I love traveling and knitting... OK out of topic

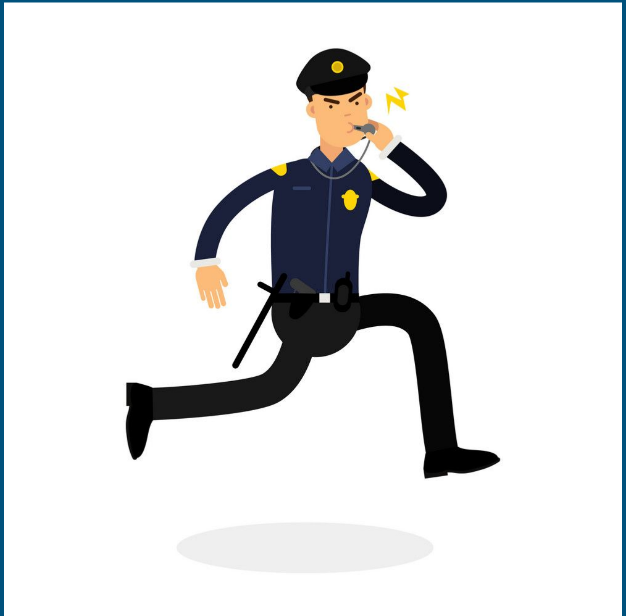
Introduction

What is reverse engineering?

- Starting from the product to get back to the plan

- It could be done for a computer binary, the design of a new dishwasher…

Be careful not to break laws



- Reverse engineering proprietary software is often restrained by some rules (publishing for example can be banned)
- The laws depend on countries
- Responsible disclosure
    - 3 months period

What is x86/x64?

- Binary
- A bunch of bytes…
  Language understandable by x86 CPUs
- Most workstations and servers
- Not designed for human readability
- X86 = 32 bits
  X64 = 64 bits
  Related to the amount of addressable memory

- Note: there are other archs like ARM (phones, ...). A different language.

```
$ hexdump -C asm_smile

00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  78 00 40 00 00 00 00 00  |..>.....x.@.....|
00000020  40 00 00 00 00 00 00 00  b0 01 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  01 00 40 00 05 00 02 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  99 00 00 00 00 00 00 00  99 00 00 00 00 00 00 00  |................|
00000070  00 00 20 00 00 00 00 00  b0 01 48 89 c7 48 c7 c6  |.. .......H..H..|
00000080  8f 00 40 00 b2 0b 0f 05  b0 3c 48 31 ff 0f 05 5b  |..@......<H1...[|
00000090  5e 30 5e 5d 20 75 21 21  0a 00 00 00 00 00 00 00  |^0^] u!!........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 01 00  |................|
000000c0  78 00 40 00 00 00 00 00  00 00 00 00 00 00 00 00  |x.@.............|
000000d0  01 00 00 00 04 00 f1 ff  00 00 00 00 00 00 00 00  |................|
000000e0  00 00 00 00 00 00 00 00  0d 00 00 00 00 00 01 00  |................|
000000f0  8f 00 40 00 00 00 00 00  00 00 00 00 00 00 00 00  |..@.............|
00000100  16 00 00 00 10 00 01 00  78 00 40 00 00 00 00 00  |........x.@.....|
00000110  00 00 00 00 00 00 00 00  11 00 00 00 10 00 01 00  |................|
00000120  99 00 60 00 00 00 00 00  00 00 00 00 00 00 00 00  |..`.............|
00000130  1d 00 00 00 10 00 01 00  99 00 60 00 00 00 00 00  |..........`.....|
00000140  00 00 00 00 00 00 00 00  24 00 00 00 10 00 01 00  |........$.......|
00000150  a0 00 60 00 00 00 00 00  00 00 00 00 00 00 00 00  |..`.............|
00000160  00 61 73 6d 5f 73 6d 69  6c 65 2e 6f 00 6d 73 67  |.asm_smile.o.msg|
00000170  00 5f 5f 62 73 73 5f 73  74 61 72 74 00 5f 65 64  |.__bss_start._ed|
00000180  61 74 61 00 5f 65 6e 64  00 00 2e 73 79 6d 74 61  |ata._end...symta|
00000190  62 00 2e 73 74 72 74 61  62 00 2e 73 68 73 74 72  |b..strtab..shstr|
000001a0  74 61 62 00 2e 74 65 78  74 00 00 00 00 00 00 00  |tab..text.......|
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

The basics of x64 binaries

Closer look at a binary structure

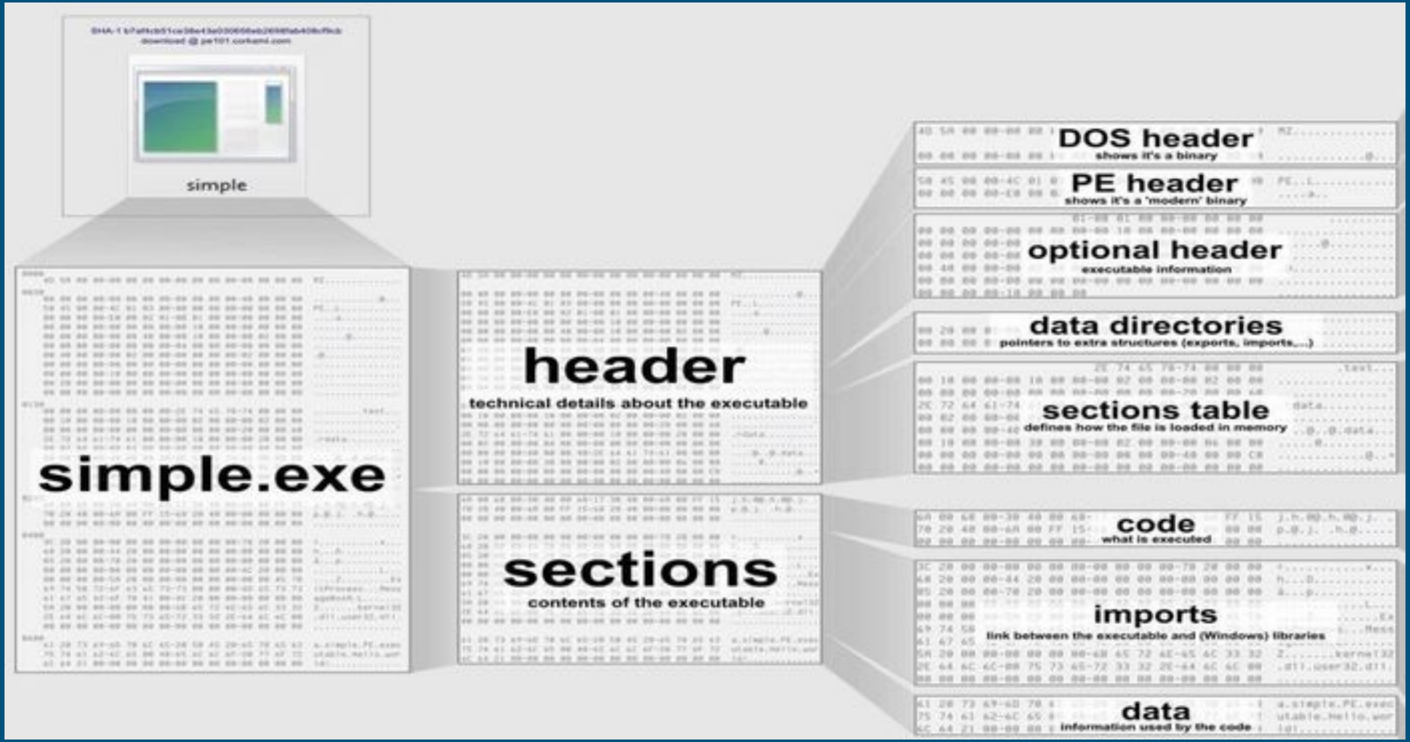
- Compiled code wrapped
- Headers to give context
- Look for magics!

- Can do some introspection with tools such as
  - CFE explorer (Windows)
  - Readelf (Linux)

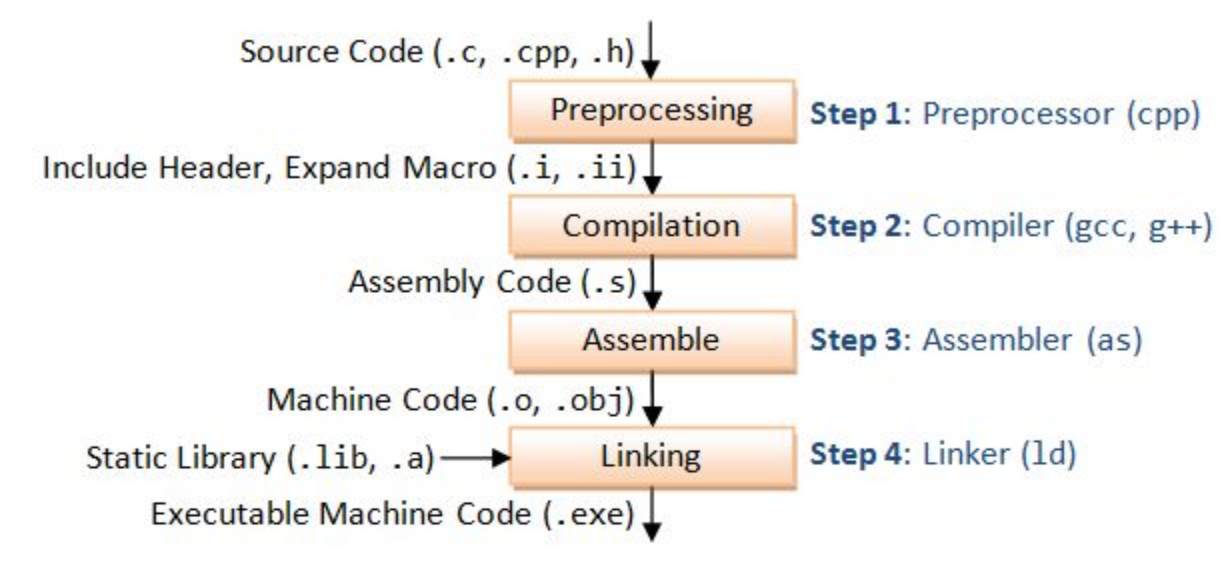
Example - PE file (Windows) - flat file

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	DOS header
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000030	00	00	00	00	00	00	00	00	00	00	00	00	80	00	00	00	
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	DOS stub
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	
00000080	50	45	00	00	4C	01	03	00	8D	FA	81	4D	00	00	00	00	PE signature, PE file header
00000090	00	00	00	00	E0	00	02	01	0B	01	08	00	00	0A	00	00	PE standard fields
000000A0	00	08	00	00	00	00	00	00	9E	28	00	00	00	20	00	00	
000000B0	00	40	00	00	00	00	40	00	00	20	00	00	00	02	00	00	
000000C0	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	PE NT fields
000000D0	00	80	00	00	00	02	00	00	01	82	00	00	03	00	40	85	
000000E0	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	
000000F0	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	
00000100	4C	28	00	00	4F	00	00	00	00	40	00	00	A8	05	00	00	
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	Data directories
00000120	00	60	00	00	0C	00	00	00	A4	27	00	00	1C	00	00	00	
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000150	00	00	00	00	00	00	00	00	00	20	00	00	08	00	00	00	
00000160	00	00	00	00	00	00	00	00	08	20	00	00	48	00	00	00	
00000170	00	00	00	00	00	00	00	00	2E	74	65	78	74	00	00	00	.text section header
00000180	A4	08	00	00	00	20	00	00	00	0A	00	00	00	02	00	00	
00000190	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60	
000001A0	2E	72	73	72	63	00	00	00	A8	05	00	00	00	40	00	00	.rsrc section header
000001B0	00	06	00	00	00	0C	00	00	00	00	00	00	00	00	00	00	
000001C0	00	00	00	00	40	00	00	40	2E	72	65	6C	6F	63	00	00	.reloc section header
000001D0	0C	00	00	00	00	60	00	00	00	02	00	00	00	12	00	00	
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	42	
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000200	80	28	00	00	00	00	00	00	48	00	00	00	02	00	05	00	.text section
00000210	E4	20	00	00	C0	06	00	00	09	00	00	00	01	00	00	06	
00000220	00	00	00	00	00	00	00	00	50	20	00	00	80	00	00	00	
00000230	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

PE file - in perspective



How is it made? A note about compilation



Concepts behind running a binary

- Binary image mapped in memory
- Threads and context
- Interaction between CPU and memory
- Different sections in memory:
    - text (code)
    - stack
    - rodata (strings)
    - ...

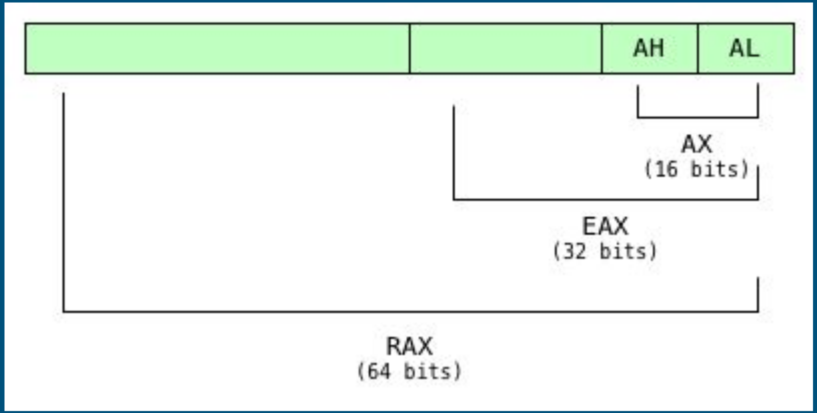
Diving into ASM

Concept of (general) registers

- Int32 (x86) or Int64 (x64) values
- Start with E in x86, R in x64
- Way for the CPU to:
    - save the state:
        - RSP = Stack Pointer
        - RIP = Instruction pointer
        - RBP = base of the stack for the current function
    - Store variable values: RAX, RBX, RCX, RDX, RSI, RDI, R8..R15
    - Can have specific uses for some functions, some can be overwritten

Length of registers

- Can be split: RCX, ECX (lower 32 bits), CX (lower 16 bits), CH, CL (8 bits)
- Example with RAX:



Most common instructions

- MOV: it copies a value (does not move it)
- LEA: Load effective address: it assigns the address of the right operand to the left operand (in Intel syntax, explained in the next slide)
- CALL
- PUSH
- POP
- RETN
- JMP
- ADD
- SUB
- Binary operations: AND, XOR, OR, ...

Intel versus AT&T syntax

- Changes in which order the operands should be read
- Small syntax changes

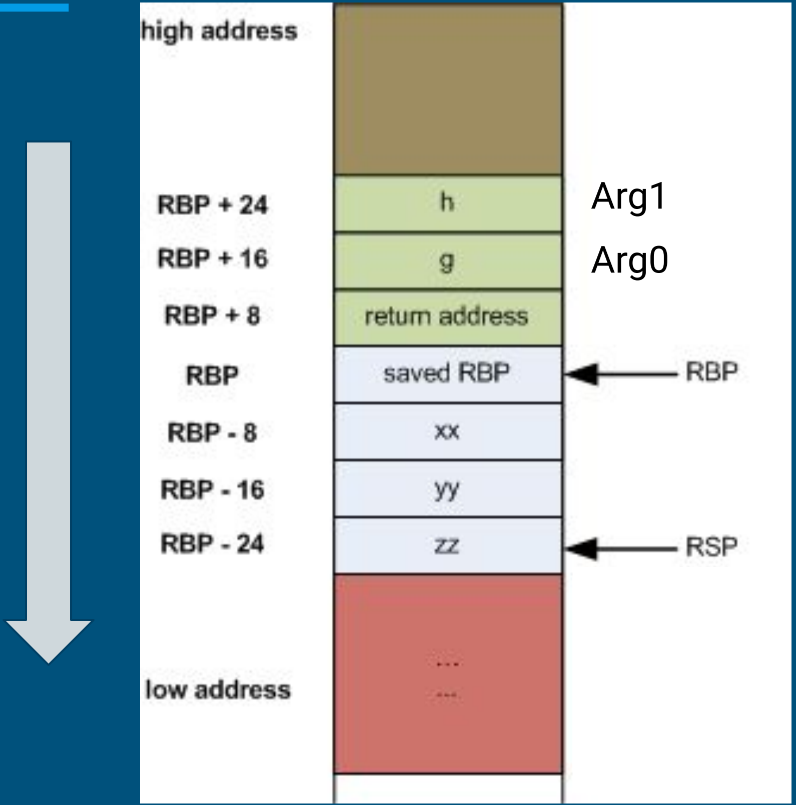
- Examples
  - Intel syntax (used in IDA): mov eax, 2
  - AT&T syntax (used in gdb): mov $0x2, %eax

- From now on in this presentation: Intel syntax

Conditions and branches

- Conditional instructions
  - TEST, CMP, …
  - Example: CMP rax, 6
- Set some special registers
- Followed by branching instructions
  - JA (jump if above)
  - JB (jump if below)
  - JE (jump if equal)
  - JNE (jump if not equal)
  - JZ (jump if zero)
  - JNZ (jump if not zero)
  - ...

Concept of stack in memory



Typical prolog and epilog

```
PUSH rbp
MOV rbp, rsp

…

POP rbp
RETN
```

Some remarks

- Dereference with []:
  - [eax] = value at address eax
  - Example: MOV ecx, [eax]: Copies into ecx the value pointed by eax

- Arguments of a function pushed right to left

Calling conventions

- Answers the question: how do I pass my arguments, clean them after use
- Default for x64: fastcall
  - Return value in RAX
  - Careful: different way to pass arguments on Unix based / Windows systems!
  - Simple case (Windows): RCX, RDX, R8, R9 + stack

```
func1(int a, int b, int c, int d, int e);
// a in RCX, b in RDX, c in R8, d in R9, e pushed on stack
```

  - Equivalent on Linux: RDI, RSI, RDX, RCX, R8, R9 + stack
- For C++: default to “thiscall” in x86
- Volatile vs non-volatile registers

But keep in mind...

… that these are just conventions

A specific compiler could have different calling patterns, a different way to deal with the stack, etc.

Signs, overflows, EFLAGS

- There is no such thing as uint, int, …
- In x86: 0x FFFF FFFF can be both -1 and max int
- Signed operations can guide

- Simple arithmetics: 0xFFFF FFFF + 1?

Other registers (FYI)

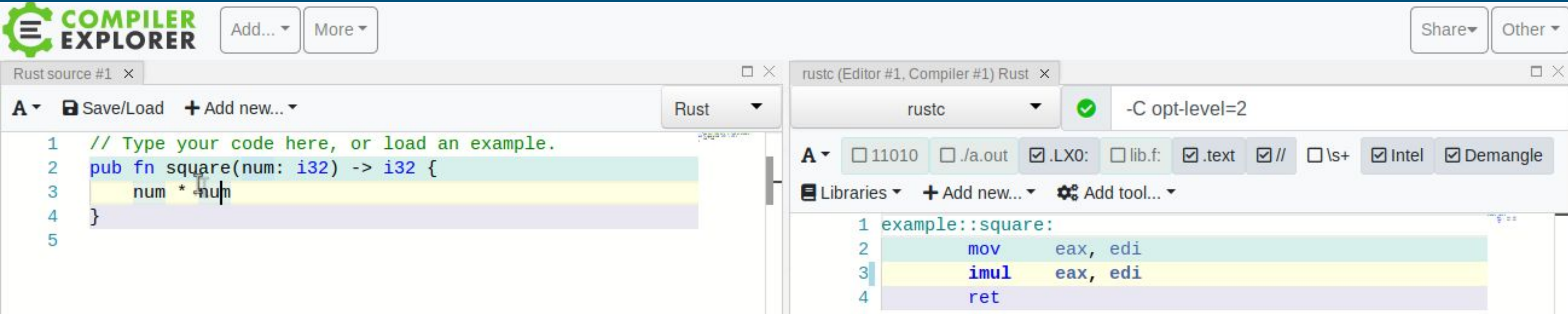
- EFLAGS: special register that carry state information (used for the results of comparison functions, ...)
- Debug Registers (DR0-DR3)
- Low level system registers: CR0, CR3, …
- MSRs
- Floats handled by XMMi

DIY to learn

Compiler Explorer: https://godbolt.org/

C++ (or another of the listed languages) -> ASM

Change the compiler, level of optimizations, ...



Static analysis

First, simple tools for analysis

- Strings
- Readelf
- CFF Explorer
- objdump

Disassemblers

- IDA
  - Used here
  - Legacy option
  - Free version with limited number of features (no debugging, no pseudocode, etc.)
- Ghidra
  - new tool published by the NSA
  - Free
- Binary ninja
- …

It is a matter of taste, just as if I was asking you your favorite color.

Sample - printHello

- Open IDA64: run “ida64”
  > New
    > select 0-printHello

```c
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

- Note: IDA will not patch your file: information stored in .idb/.i64

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
lea     rdi, s          ; "Hello, World!"
call    _puts
mov     eax, 0
pop     rbp
retn
; } // starts at 63A
main endp
```

```
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

Strategies

- Look for the **main** function in the left pane
- Look for **strings** (View > Open subviews > Strings)
- Look at the **imported** functions
- …
- Usually, binaries too large to be analyzed from A to Z

Useful commands on IDA

- Space: switch views
- N: rename element
- Y: prototype functions
- X: cross reference a function, a variable inside the binary
- Double click: enter (function, view of data, etc)
- Esc: back to the previous location
- ‘:’ and ‘;’: for comments

- C: analyze the following bytes as code
- U: undefine

Other tips for IDA

- Structures
  - Structure tab
  - Either import well known structures or create them
  - Alt+Q to apply structures
  - Can import .h files
- In the options tab, in “general”
  - Can check auto-comment to display information about the instructions
  - Can display the opcodes (by selecting 10 for example)

First exercise

Instructions: 1-WhatDoIDo

- Please open the binary in IDA.
- What does it do?

- Tips
  - Rename the local variables (N)
  - Comment as much as you can (:)
  - Prototype your functions (Y) -
    IDA will propagate the information

Solution...

```
#include <stdio.h>

int arithmetics (int a, int b) {
  int sum = a + b;
  return sum;
}

int inverse (int c) {
  return -c;
}

int main(int argc, char** argv) {
  int a = 1;
  int b = 6;
  int res = arithmetics(inverse(a), b);
  printf("The output is: %d\n", res);
  return res;
}
```

Debug

Warning

- When debugging,
  the binary really RUNS on your machine …
  … even if it is MALWARE

- If you want to study malware / unknown binary, do it in a proper **isolated** environment
- At home: in a VM, no network card, no shared folders with the host …

In IDA

- Setup
  - Tab “Debugger”
  - F9 - Select a debugger
  - “Local Linux debugger”
- Add breakpoints (F2)
- Run (F9)
- Step (into: F7 - over: F8)

- Side note for Linux: first, chmod +x your binary ;)

Other debuggers

- Gdb on Linux
- WinDbg on Windows
  - Binding IDA <-> WinDbg: plugin ret-sync
- x64 on Windows
- etc.

Architecture matters

- You can only run something architecture-compatible:
  - no ELF on Windows
  - no PE file on Linux
  - no ARM binary on your x86 machine
- Remote debugging in such case

The developers might make it harder for you

- Obfuscation
- Anti-debug
- …

Second exercise

Instructions - crackme

You can either:

- Analyse the binary statically
- Or statically + dynamically
- Tip: there is a python console at the bottom

Solution

```c
#include <stdio.h>

char myResult[9] = {19,4,18,52,14,13,13,4,41};

int processPassword(char* userInput){
  int i;
  char c;
  for (i=0; i<9; i++){
    c = userInput[i] ^ 'a';
    if (myResult[8-i] != c) {
      return 1;
    }
  }
  return 0;
}
```

```c
int main(int argc, char** argv) {
  char userInput[16];
  int ret;
  printf("Can you defeat this challenge?\n");
  printf("Please enter the code: ");
  fgets(userInput, 16, stdin);
  ret = processPassword(userInput);
  if (ret == 0){
    printf("Success\n");
  }
  else {
    printf("Try again\n");
  }
  return 0;
}
```

A little taste of automation with Angr

```
win_addr =  0x4007CD

# Load the binary
p = angr.Project('./challenge_RE101')

# Define the input - the second parameter is the number of bits, a char has 8 bits, and the input is set to 9 chars
inp = claripy.BVS("inp", 8*9)

# Starts execution only at that point to focus on the function of interest
state = p.factory.entry_state(addr=0x4007B8)

# The function only has one input set here. The idea is to populate the stack at RBP+s = RBP-0x20
state.memory.store(state.regs.rbp-0x20,inp)

# Starts the simulation
sm = p.factory.simulation_manager(state)
sm.explore(find = win_addr)
found = sm.found[0]

# Formatting and printing
solution = str(found.solver.eval(inp,cast_to=bytes))
print(solution)
```

To go further ...

The “cheating” way (that will save you time)

… and pseudocode appeared

Only with IDA Pro - F5

More advanced topics

- IDA scripting in Python
- Plugins
- How to identify and handle crypto
- How to reverse C++ in IDA
- Specific tools to reverse Java, .NET, etc.
- Angr and other solvers
- Remote debugging in a VM

Resources

- 1 book: “Practical reverse engineering” by B. Dang, A. Gazet, E. Bachaalany
- Practice practice practice:
  - rootme : https://www.root-me.org
  - CTF of all sorts
  - You can visit this site to get inspiration: https://thehacktoday.com/22-hacking-sites-ctfs-and-wargames-to-practice-your-hacking-skills/
  - Advent CTF: 1 challenge per day before Xmas :)
  - Different types of challenges: crackmes, exploits, …
- "Exercises for Teaching Reverse Engineering" by John Aycock
- And to get to know the instructions: Intel books (available online)

Thanks

Have fun :)
And please don’t hack the planet now.