

# CS356: Reverse Engineering

W. Michael Petullo

University of Wisconsin–La Crosse

As of November 28, 2021





# Reverse Engineering: Introduction

Reverse engineering is the process of analyzing a subject system to

- ▶ identify the system's components and their interrelationships and
- ▶ create representations of the system in another form or at a higher level of abstraction.

Chikofsky and Cross, "Reverse Engineering and Design Recovery—A Taxonomy"

Binary reverse engineering starts with machine code and aims to recreate some approximation of the original source code and therefore recover how the system works or what it does.



# Reverse Engineering: Why?

- ▶ Interfacing with an existing system.
- ▶ Coming to understand the purpose of a piece of malware.
- ▶ Bypassing copyright protections.\*
- ▶ Analyzing the correctness or security of a system.
- ▶ Cost analysis.
- ▶ Modifying a component for a new use.

\*Might violate the law.



Understanding a program starts with understanding two things:

**Control Flow** Build a graph of execution flows through program; identify loops and conditionals; basic blocks.

**Data Flow** Understand the flow of information through the program; inputs, variables, and outputs.



- ▶ `objdump -D BINARY`
- ▶ GDB
- ▶ IDA Pro (\$)
- ▶ NSA's Ghidra (open source)



## Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2 (2A, 2B, 2C & 2D):  
Instruction Set Reference, A-Z

**NOTE:** The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325383-075US  
June 2021

Volume 2 of Intel's architecture manual describes the instructions available to application developers. A link to this document exists on the course resources page.

Objdump's default output uses AT&T syntax; `--disassembler-options=intel` switches to Intel's syntax and ordering. GDB has a similar option.

```
printf(" Hello , _world!\n" );
```

Examples compiled with `-O0`, which turns off compiler optimizations. Optimizations can lead to surprising changes to binaries.

```
401126: push    %rbp
401127: mov     %rsp,%rbp
40112a: mov     $0x402010,%edi
40112f: callq  401030 <puts>
401134: mov     $0x0,%eax
401139: pop     %rbp
40113a: retq
```

Remember: f(rdi, rsi, rdx, rcx, r8, r9, stack ...)

```
bool b = true;
if (b) {
    printf(" Hello ,_world!\n");
} else {
    printf(" Goodbye ,_...!\n");
}
```

```
401126: push    %rbp
401127: mov     %rsp,%rbp
40112a: sub     $0x10,%rsp
40112e: movb   $0x1,-0x1(%rbp)
401132: cmpb   $0x0,-0x1(%rbp)
401136: je     401144 <main+0x1e>
401138: mov     $0x402010,%edi
40113d: callq  401030 <puts>
401142: jmp    40114e <main+0x28>
401144: mov     $0x40201e,%edi
401149: callq  401030 <puts>
40114e: mov     $0x0,%eax
401153: leaveq
401154: retq
```



# Reverse Engineering: case



```
int i = 1;
switch(i) {
case 0:
    printf(" Hello!\n");
    break;
case 1:
    printf(" Goodbye!\n");
    break;
case 2:
    printf(" Other!\n");
    break;
}
```

```
40112e: movl    $0x1,-0x4(%rbp)
401135: cmpl   $0x2,-0x4(%rbp)
401139: je     401167 <main+0x41>
40113b: cmpl   $0x2,-0x4(%rbp)
40113f: jg     401172 <main+0x4c>
401141: cmpl   $0x0,-0x4(%rbp)
401145: je     40114f <main+0x29>
401147: cmpl   $0x1,-0x4(%rbp)
40114b: je     40115b <main+0x35>
40114d: jmp    401172 <main+0x4c>
40114f: mov    $0x402010,%edi
401154: callq  401030 <puts>
401159: jmp    401172 <main+0x4c>
40115b: mov    $0x40201e,%edi
401160: callq  401030 <puts>
401165: jmp    401172 <main+0x4c>
401167: mov    $0x40202e,%edi
40116c: callq  401030 <puts>
```



# Reverse Engineering: for

```
for (int i = 0; i < 10; i++) {  
    printf(" Hello ,_world!\n");  
}
```

```
401126: push    %rbp  
401127: mov     %rsp,%rbp  
40112a: sub     $0x10,%rsp  
40112e: movl   $0x0,-0x4(%rbp)  
401135: jmp     401145 <main+0x1f>  
401137: mov     $0x402010,%edi  
40113c: callq  401030 <puts@plt>  
401141: addl   $0x1,-0x4(%rbp)  
401145: cmpl   $0x9,-0x4(%rbp)  
401149: jle     401137 <main+0x11>  
40114b: mov     $0x0,%eax  
401150: leaveq  
401151: retq
```



# Reverse Engineering: while

```
int done = false;
while (!done) {
    printf("Hello ,_world!\n");
    done = true;
}
```

```
401126: push    %rbp
401127: mov     %rsp,%rbp
40112a: sub     $0x10,%rsp
40112e: movl   $0x0,-0x4(%rbp)
401135: jmp    401148 <main+0x22>
401137: mov     $0x402010,%edi
40113c: callq  401030 <puts@plt>
401141: movl   $0x1,-0x4(%rbp)
401148: cmpl   $0x0,-0x4(%rbp)
40114c: je     401137 <main+0x11>
40114e: mov     $0x0,%eax
401153: leaveq
401154: retq
```

```
int done = false;
do {
    printf("Hello ,_world!\n");
    done = true;
} while (!done);
```

```
401126: push    %rbp
401127: mov     %rsp,%rbp
40112a: sub     $0x10,%rsp
40112e: movl   $0x0,-0x4(%rbp)
401135: mov     $0x402010,%edi
40113a: callq  401030 <puts@plt>
40113f: movl   $0x1,-0x4(%rbp)
401146: cmpl   $0x0,-0x4(%rbp)
40114a: je     401135 <main+0xf>
40114c: mov     $0x0,%eax
401151: leaveq
401152: retq
```

# Reverse Engineering: array

```
int buf[] = { 2, 3, 1 };

printf("%d%d%d\n",
       buf[2],
       buf[0],
       buf[1]);
```

```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: sub    $0x10,%rsp
40112e: movl   $0x2,-0xc(%rbp)
401135: movl   $0x3,-0x8(%rbp)
40113c: movl   $0x1,-0x4(%rbp)
401143: mov    -0x8(%rbp),%ecx
401146: mov    -0xc(%rbp),%edx
401149: mov    -0x4(%rbp),%eax
40114c: mov    %eax,%esi
40114e: mov    $0x402010,%edi
401153: mov    $0x0,%eax
401158: callq  401030 <printf>
40115d: mov    $0x0,%eax
401162: leaveq
401163: retq
```

# Reverse Engineering: Work



```
8048535: eb 32          jmp     8048569 <main+0x7c>
8048537: 8d 54 24 1c   lea    0x1c(%esp), %edx  edx ptr to esp + 0x1c / 28 = Start of proposed key
804853b: 8b 44 24 14   mov    0x14(%esp), %eax  esp+14 → eax = i
804853f: 01 d0        add    %edx, %eax      edx = ptr + i
8048541: 0f b6 00     movzbl (%eax), %eax    Place current byte in eax
8048544: 83 e8 20     sub    $0x20, %eax     Subtract 0x20/32 from current byte
8048547: 88 44 24 13   mov    %al, 0x13(%esp) Current char in proposed → esp+13 in proposed key
804854b: 8b 44 24 14   mov    0x14(%esp), %eax  eax = i
804854f: 05 2c a0 04 06 add    $0x804a02c, %eax  eax = solution[i]
8048554: 0f b6 00     movzbl (%eax), %eax    eax = solution[i]
8048557: 3a 44 24 13   cmp    0x13(%esp), %al   } Check if x[i] = y[i]
804855b: 74 07        je     8048564 <main+0x77>
804855d: b8 00 00 00 00 mov    $0x0, %eax
8048562: eb 20        jmp    8048584 <main+0x97>
8048564: 83 44 24 14 01 addl   $0x1, 0x14(%esp)  i++.
8048569: 8b 44 24 14   mov    0x14(%esp), %eax  → i.0 → i.7 } Checked by 7 bytes "yet? AKA processed"
804856d: 3b 44 24 18   cmp    0x18(%esp), %eax
8048571: 7c c4        jl     8048537 <main+0x4a>
8048573: c7 04 24 38 86 04 08 movl   $0x8048638, (%esp)
804857a: e8 21 fe ff ff call   80483a0 <puts@plt> → ("WIN!!! ...")
804857f: b8 00 00 00 00 mov    $0x0, %eax
8048584: 8b 4c 24 5c   mov    0x5c(%esp), %ecx
8048588: 65 33 0d 14 00 00 00 xor    %gs:0x14, %ecx
804858f: 74 05        je     8048596 <main+0xa9>
8048591: e8 fa fd ff ff call   8048390 <__stack_chk_fail@plt> } Stack Canary
8048596: c9          leave
```

# Reverse Engineering: Ghidra



The screenshot displays the Ghidra interface with a code flow graph on the left and an assembly listing on the right.

**Code Flow Graph (Left):**

- A central node (blue box) contains: `PUSH RBP`, `MOV RBP, RSP`, `MOV EDI, 0x402010`, and `CALL 0x00401030`. It is circled in red.
- A left node (pink box) contains: `MOV EAX, 0x0`, `POP RBP`, and `RET`. A blue arrow points from the central node to this node.
- A right node (orange box) contains: `JMP qword ptr [0x00404018]`. An orange arrow points from the central node to this node.
- A bottom node (orange box) contains: `?? ??`. A cyan arrow points from the right node to this node.

**Assembly Listing (Right):**

```
Listing: hello - (14 addresses select...  
00401126 55      PUSH   RBP  
00401127 48 89 e5 MOV    RBP, RSP  
0040112a bf 10 20 MOV    EDI=>s_Hello_world  
          40 00  
0040112f e8 fc fe CALL   <EXTERNAL>.:puts  
          ff ff  
00401134 b8 00 00 MOV    EAX, 0x0  
          00 00  
00401139 5d      POP    RBP  
0040113a c3      RET  
0040113b 0f     ??    0Fh  
0040113c 1f     ??    1Fh  
0040113d 44     ??    44h D
```

**Tool Chest (Bottom):**

- Active Project: Test
- Filter: [ ]
- Tree View | Table View
- Running Tools: Workspace

**Status Bar (Bottom):** Mon Nov 29 20:14:30 CST 2021 Recovery snapshot created: /home/mike...



Graded Homework Aquinas: reveng2, reveng3, and reveng4

<https://www.flyn.org/courses/cs356/schedule>