Reverse Engineering: From Assembly to C

Prepared by Amir (with ChatGPT)

November 12, 2025

Outline

Why reverse engineer?

- Recover high-level intent from binaries (**security auditing**, bug hunting, verifying third-party libraries)
- Understand **calling conventions**, ABI, optimization effects
- Learn how compilers translate C to assembly

Common tools

- **objdump**, readelf, nm, strings
- **Ghidra**, IDA Pro, Binary Ninja (GUI decompilers)
- radare2 / rizin (CLI reversing)
- gcc/clang for compiling small examples, `objdump -d` for disassembly

x86_64 quickly

- Registers: `rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8–r15`
- Typical calling convention (System V AMD64):
  - args: `rdi, rsi, rdx, rcx, r8, r9`
  - return: `rax`
- Stack grows down; `rbp` often used as frame pointer (when enabled)

Reading a small assembly function

Disassembly (Intel-like pseudocode):

```
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], edi
mov eax, DWORD PTR [rbp-4]
add eax, 5
pop rbp
ret
```

Homework: convert this to C.

Solution: mapping to C

Assembly corresponds to:

```
int add5(int x) {
return x + 5;
}
```

Explanation:

- parameter in `edi` (first int arg)
- result returned in `eax`

Common idioms

- Prologue/epilogue: `push rbp; mov rbp, rsp` ... `pop rbp; ret`
- Local variables: stored at **negative offsets from `rbp`**
- Loops: compare + conditional jump (e.g., `cmp; jne/jle/jg`)
- Function calls: `call` and cleanup depends on convention

Example: loop in assembly

Assembly:

```
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], 0    ; i = 0
jmp .L2
.L3:
mov eax, DWORD PTR [rbp-4]
add eax, 1
mov DWORD PTR [rbp-4], eax
.L2:
mov eax, DWORD PTR [rbp-4]
cmp eax, 10
jl .L3
pop rbp
ret
```

Loop -> C

Equivalent C:

```c
void count_to_10() {
int i = 0;
while (i < 10) {
i = i + 1;
}
}
```

Notes: the compiler uses increments and compares; optimizations may inline or unroll.

Calling convention reminders

- Integer args in **registers** (rdi, rsi, rdx, ...)
- Caller-saved vs callee-saved registers
- Watch out for optimizations: **tail calls**, inlining, frame-pointer omission

Assembly: simplified strcpy-like

```
; char *strcpy_simple(char *dst, const char *src)
push rbp
mov rbp, rsp
mov QWORD PTR [rbp-16], rdi   ; dst
mov QWORD PTR [rbp-24], rsi   ; src
mov rax, QWORD PTR [rbp-16]
mov rcx, QWORD PTR [rbp-24]
.Lloop:
mov dl, BYTE PTR [rcx]
mov BYTE PTR [rax], dl
inc rcx
inc rax
test dl, dl
jne .Lloop
mov rax, QWORD PTR [rbp-16]
pop rbp
ret
```

C equivalent: strcpy-like

```c
char *strcpy_simple(char *dst, const char *src) {
char *d = dst;
const char *s = src;
while (1) {
char c = *s;
*d = c;
s++; d++;
if (c == '\0') break;
}
return dst;
}
```

Practical workflow

1. Obtain binary and symbols (if available)
2. Use `objdump -d` or **Ghidra** to get disassembly
3. Identify functions, calling conventions, and constants
4. Rename variables and functions as you infer meaning
5. Reconstruct high-level control flow and types

Quick tips

- Look for **string references** (`strings`, cross-reference in disassembler)
- Identify library calls (e.g., calls to `printf`, `malloc`)
- Use type inference: size of memory ops (byte/word/dword/qword) suggests types
- Control-flow graphs help reconstruct structured code

Exercise 1

Disassembly:

```
push rbp
mov rbp, rsp
mov eax, edi
cdqe
idiv esi
pop rbp
ret
```

Task: propose a C prototype and implementation.

Exercise 1 – solution (one possible)

```c
long div_signed(int a, int b) {
return (long)a / (long)b; // note: idiv uses rax/rdx - sign
    extension
}
```

Explanation: `cdqe` sign-extends `eax` into `rax`; `idiv esi` divides `rax` by `esi`.

Optimizations and pitfalls

- Compiler optimizations can reorder, inline, or eliminate variables
- Omitted frame pointer makes local offsets harder to follow
- Position-independent code (PIC) uses **RIP-relative addressing**

Further resources

- "**Practical Reverse Engineering**" (book)
- Ghidra tutorials and official docs
- x86-64 System V ABI specification

Wrap-up

- Practice by compiling small C examples and inspecting disassembly
- Reconstruct control flow, map registers -> variables, and test hypotheses

Appendix: commands

```
compile
gcc -O0 -g -o ex example.c

disassemble
objdump -d -M intel ex > ex.dis

show strings
strings ex
```