

Cosc 69.16: Reverse Engineering

Prishita Dharampal

Credit Statement: Used perplexity.ai and stackoverflow to look up instructions I wasn't familiar with.

For this lab assignment, you will be compiling short C programs and then reverse engineering them to see what they look like inside. This assignment will be performed individually, first on the Unix command line and then in the GHIDRA reverse engineering framework.

Problem 1. The following is a short C program. Place it in a file named `first.c` and compile it with `cc -o first first.c` to produce an executable. You can run this with `./first`.

```
#include <stdio.h>

int main(int argc, char **argv) {
    if (argc > 1) {
        printf("Hello world. %s\n", argv[1]);
    } else {
        printf("Aloha!\n");
    }
    return 0;
}
```

1. What file format and architecture is your file? You can find this by running the `file` command on your executable.
2. Disassemble your executable with the `objdump -d` command and write down the disassembly of the `main` function.
3. Add comments to the disassembly marking:
 - the return value (0) of `main`,
 - the address of the strings "Hello world argv[1]" and "Aloha!",
 - the registers that store `int argc` and `char **argv`.

This process is typically called *annotation*.

4. objdump -d often does not include strings in its listing, but objdump -D and other tools will. Include the output lines representing the strings, along with the options or actions used to obtain them.
5. Why are the bytes of the string interpreted as machine language?
6. Use an ASCII table to decode the bytes of the strings. C strings end with a null byte (0), so include every byte from the beginning of the string to the null terminator.

Solution.

1. The file format is Mach -O 64-bit and the architecture is x86_64.

2.

```

0000000100000470 <_main>:
100000470: 55          pushq   %rbp
100000471: 48 89 e5    movq    %rsp, %rbp
100000474: 48 83 ec 10 subq    $0x10, %rsp
100000478: c7 45 fc 00 movl    $0x0, -0x4(%rbp)
10000047f: 89 7d f8    movl    %edi, -0x8(%rbp)
100000482: 48 89 75 f0 movq    %rsi, -0x10(%rbp)
100000486: 83 7d f8 01 cmpl    $0x1, -0x8(%rbp)
10000048a: 7e 18        jle     0x1000004a4 <_main+0x34>
10000048c: 48 8b 45 f0 movq    -0x10(%rbp), %rax
100000490: 48 8b 70 08 movq    0x8(%rax), %rsi
100000494: 48 8d 3d 25 leaq    0x25(%rip), %rdi
    ## 0x1000004c0 <_printf+0x1000004c0>
10000049b: b0 00        movb    $0x0, %al
10000049d: e8 18 00 00 00 callq   0x1000004ba
    <_printf+0x1000004ba>
1000004a2: eb 0e        jmp    0x1000004b2 <_main+0x42>
1000004a4: 48 8d 3d 26 leaq    0x26(%rip), %rdi
    ## 0x1000004d1 <_printf+0x1000004d1>
1000004ab: b0 00        movb    $0x0, %al
1000004ad: e8 08 00 00 00 callq   0x1000004ba
    <_printf+0x1000004ba>
1000004b2: 31 c0        xorl    %eax, %eax
1000004b4: 48 83 c4 10  addq    $0x10, %rsp
1000004b8: 5d          popq    %rbp
1000004b9: c3          retq

```

3.

```

0000000100000470 <_main>:
100000470: 55          pushq  %rbp
100000471: 48 89 e5    movq   %rsp, %rbp
100000474: 48 83 ec 10 subq   $0x10, %rsp
100000478: c7 45 fc 00 movl   $0x0, -0x4(%rbp)
10000047f: 89 7d f8    movl   %edi, -0x8(%rbp)
    ##copies argc from edi to [rbp - 8]
100000482: 48 89 75 f0 movq   %rsi, -0x10(%rbp)
    ##copies argv from rsi to [rbp - 10]
100000486: 83 7d f8 01 cmpl   $0x1, -0x8(%rbp)
10000048a: 7e 18        jle    0x1000004a4 <_main+0x34>
10000048c: 48 8b 45 f0 movq   -0x10(%rbp), %rax
100000490: 48 8b 70 08 movq   0x8(%rax), %rsi
100000494: 48 8d 3d 25 leaq   0x25(%rip), %rdi
    ## 0x1000004c0 <_printf+0x1000004c0>
    ##'Hello world.' is stored at 0x25(%rip)
10000049b: b0 00        movb   $0x0, %al
10000049d: e8 18 00 00 00 callq  0x1000004ba
    <_printf+0x1000004ba>
1000004a2: eb 0e        jmp   0x1000004b2 <_main+0x42>
1000004a4: 48 8d 3d 26 leaq   0x26(%rip), %rdi
    ## 0x1000004d1 <_printf+0x1000004d1>
    ##'Aloha!' is stored at 0x26(%rip)
1000004ab: b0 00        movb   $0x0, %al
1000004ad: e8 08 00 00 00 callq  0x1000004ba
    <_printf+0x1000004ba>
1000004b2: 31 c0        xorl   %eax, %eax
    ##return value of main
1000004b4: 48 83 c4 10 addq   $0x10, %rsp
1000004b8: 5d           popq   %rbp
1000004b9: c3           retq

```

4. Used objdump -D to get the entire disassembly including the snippet below.

Disassembly of section __TEXT,__cstring:
<pre> 00000001000004c0 <__cstring>: 1000004c0: 48 65 gs 1000004c2: 6c insb %dx, %es:(%rdi) 1000004c3: 6c insb %dx, %es:(%rdi) 1000004c4: 6f outsl (%rsi), %dx 1000004c5: 20 77 6f andb %dh, 0x6f(%rdi) 1000004c8: 72 6c jb 0x100000536 <_printf+0x100000536> 1000004ca: 64 2e 20 25 73 0a 00 41 andb %ah, %cs:0x41000a73(%rip) 1000004d2: 6c insb %dx, %es:(%rdi) 1000004d3: 6f outsl (%rsi), %dx 1000004d4: 68 61 21 0a 00 pushq \$0xa2161 ## imm = 0xA2161 </pre>

5. Using `-D` with `objdump` assumes that any symbols found in a code section are code, and disassembles them.

6.

```
48 -> 'H'  
65 -> 'e'  
6c -> 'l'  
6c -> 'l'  
6f -> 'o'  
20 -> ' ' (white space)  
77 -> 'w'  
6f -> 'o'  
72 -> 'r'  
6c -> 'l'  
64 -> 'd'  
2e -> '.'  
20 -> ' ' (white space)  
25 -> '%'  
73 -> 's'  
0a -> LF (line feed)  
00 -> NUL  
41 -> 'A'  
6c -> 'l'  
6f -> 'o'  
68 -> 'h'  
61 -> 'a'  
21 -> '!'  
0a -> LF (line feed)  
00 -> NUL
```

Problem 2. Let's try a program that's a little more complicated, which we'll call `second.c`.

```
#include <stdio.h>

int square(int i) {
    return i * i;
}

int main(int argc, char **argv) {
    for (int i = 0; i < 10; i++) {
        printf("The square of %d is %d.\n", i, square(i));
    }
}
```

Compile the program with both `cc -O3 -o second-fast second.c` and `cc -Os -o second-small second.c`. The `-O` flags tell the compiler to optimize the build, with `-O3` producing the fastest code and `-Os` producing the smallest code.

Disassemble both executables with `objdump -d`. What is different about their main functions? Which version is easier to understand, and why?

Solution.

Except for the obvious difference in the number of instructions, the `second-fast` `objdump` has no loops or calls to `square()` and is incredibily unreadable. Whereas `second-small` `objdump` is easier to understand because its structure matches the source code and one can identify the loop structure from the disassembly.

Problem 3. Our first two programs included symbols, so it was easy to locate main and identify variable names. For this program, strip the symbols to create a more realistic executable.

Compile with `cc -o third third.c`, then run `objdump -d` before and after stripping the binary with `strip third`.

What are the differences between these two `objdump` outputs, and why?

```
#include <stdio.h>
#include <stdlib.h>

__attribute__((noinline))
void guess(int g) {
    if (g == 42)
        printf("That's right!\n");
    else
        printf("Nope, that's the wrong number.\n");
}

int main(int argc, char **argv) {
    if (argc == 2) {
        guess(atoi(argv[1]));
    } else {
        printf("Give me a number?\n");
    }
}
```

Instead of `objdump`, use GHIDRA for this assignment.

After importing the binary and allowing GHIDRA to auto-analyze it, complete the following:

1. In the Symbols pane, practice navigating between functions.
2. Observe that after stripping, GHIDRA does not know local function or variable names, though it may identify standard library functions such as `atoi()`.
3. Navigate to the function that calls `atoi()`, rename it `main`.
4. Locate and rename the `guess()` function.
5. Rename local variables in `guess()` to match the original source code.
6. Using the decompiler view as a guide, comment each line of the assembly code for `guess()` and paste the commented disassembly into your report.

Solution.

Function names change from,

```
:
10000047f: 75 10          jne    0x100000491 <_guess+0x21>
:
1000004ca: 75 16          jne    0x1000004e2 <_main+0x32>
:
```

to

```
:
10000047f: 75 10          jne    0x100000491 <__text+0x21>
:
1000004ca: 75 16          jne    0x1000004e2 <__text+0x72>
:
```

Instrestingly however, `printf` stays regardless. This happens because `strip` removes the symbol table, so all names and references are lost. `printf` however stays because it is stored in the procedural linkage table.

GHIDRA

```
                //
// __text
// __TEXT
// ram:100000470-ram:1000004f9
//

***** FUNCTION *****
*

***** undefined guess() *****
undefined             <UNASSIGNED>      <RETURN>
undefined4           Stack[-0xc]:4   g
XREF [2]:           100000478(W),

                    10000047b(R)
                    guess
XREF [2]:           entry:1000004db(c), 1000020a8(*)
100000470 55        PUSH      RBP
                     pushes rbp to the top of the stack
100000471 48 89 e5  MOV       RBP,RSP
                     copies the address of rsp into rbp
100000474 48 83 ec 10 SUB      RSP,0x10
                     allocating 16 bytes for variable
```

```

100000478 89 7d fc      MOV        dword ptr [RBP + g], EDI
                           g is stored in edi
10000047b 83 7d fc 2a    CMP        dword ptr [RBP + g], 0x2a
                           comparing g with 42
10000047f 75 10         JNZ        LAB_100000491
                           begining else block
100000481 48 8d 3d      LEA        RDI, [s_That's_right!_100000506]
                           = "That's right!\n"
                           7e 00 00 00
                           storing 'That's right!' in rdi
100000488 b0 00         MOV        AL, 0x0
                           AL=0 (printf variadic arg count)
10000048a e8 71 00      CALL       _printf
                           int _printf(char * param_1, ...)
                           00 00
                           calls printf with 'That's right!'
10000048f eb 0e         JMP        LAB_10000049f
                           jumps to end
                           LAB_100000491
                           XREF[1]: 10000047f(j)
100000491 48 8d 3d      LEA        RDI, [s_Nope,_that's_the_wrong_number._100000515] = "Nope, that's the
                           wrong number
                           7d 00 00 00
                           else block; stores 'Nope, that's the wrong number.' in
                           rdi
100000498 b0 00         MOV        AL, 0x0
                           AL=0 (printf variadic arg count)
10000049a e8 61 00      CALL       _printf
                           int _printf(char * param_1, ...)
                           00 00
                           calls printf with 'Nope, that's the wrong number.'
                           LAB_10000049f
                           XREF[1]: 10000048f(j)
10000049f 48 83 c4 10    ADD        RSP, 0x10
                           deallocates the stack space
1000004a3 5d             POP       RBP
                           pops the return address
1000004a4 c3             RET
                           returns void

```