Lab 03: Understanding Disassembly

In this lab, you will explore the connection between high- and low-level software.

Purpose

To understand how high-level concepts are implemented in low-level sofware. You will use a development environment to compile sample programs and/or code segments to see what your toolchain outputs. If you have time, I encourage you to explore multiple toolchains to compare how different platforms and compilers work.

Procedure

You will need access to the following materials:

- A functioning C development environment for Windows, Linux, or macOS.
- A corresponding disassembler for your chosen platform(s).

You may choose your preferred environments and tools. If you have previous reverse engineering experience, I encourage you to explore a platform you are less familiar with.

1. Use your development environment to compile, assemble, link to a shared library (.so/.DLL), and examine artifacts containing various C constructs.

   You have been provided with source code and a Makefile for several isolated C constructs. The Makefile is best suited for Linux, but with some effort, the lab can also be completed on Windows.

2. Obtain a disassembly dump of your compiled code.

   Use `dumpbin` or `objdump` on Windows, `objdump` on Linux, or `otool` on macOS. The provided Makefile will do this for you on Linux.

   There are several places where you can obtain (dis)assembly.

   (a) With GCC, you can compile to assembly using the `-S -masm=intel` flags, optionally including debug info using `-g`. The advantage is that there is more context present from the source code. The downside is there's a lot of noise, and this not true disassembly, but assembly, which is not reflective of real-life RE scenarios.

   (b) You can dump the compiled object (`.o` on Unix/`.OBJ` on Windows). The advantage is that the compiled items are better isolated, so less noise. The disadvantage is that on some platforms, placeholders are left with `0`s, and while more likely than assembly, is still unlikely in real-life RE scenarios.

   (c) You can dump the linked executable (/`.EXE`) or library (`.so`/`.DLL`). The advantage is that this is the most true-to-life scenario. The disadvantage is that there is often a lot of extra noise with little context.

   While you are encouraged to explore the individual stages of the build process, for this lab, you are expected to use disassembly by compiling and dumping a linked library.

3. For each of the following constructs compare the source and dump to show how the disassembly implements the source.

   - An arithmetic operation
   - An `if` block (include control-flow graph, CFG)
   - An `if-else` block (CFG)
   - A `while` loop (CFG)
   - A `for` loop (CFG)
   - A `switch` statement (CFG)
   - A function: including prologue, epilogue, and a call site
   - Accessing a primitive variable (global, static, parameter, local)

  - For this one, when examining a global variable, examine it both when compiled as an executable, and when compiled as a shared library. Again, the provided Makefile can do this for you.
- Using a string literal (e.g., in a `printf`)
- Accessing an element of an array (static, pointer) (constant index, variable index)
  - For variable indices, you may find it useful to trace the disassembly by hand, or construct a data-flow graph.
- Accessing a field of a struct (static, pointer)
- Accessing a field of a struct-array field of a struct pointer (e.g., `structPtr->f1[i].f2`)
  - Some constant offsets will likely appear. Please explain how the compiler derived said constants.

4. For the data access constructs, show both the code performing the access (`.text` section), as well as the initialized memory (`.data`, `.rodata` sections), if applicable.
5. Derive plausible source code for the following disassembly:

```
0000000180001000:  45 33 c0       XOR  R8D,R8D
0000000180001003:  ba 01 00 00    MOV  EDX,0x1
                   00
0000000180001008:  3b ca          CMP  ECX,EDX
000000018000100a:  76 29          JBE  180001035
000000018000100c:  ff c9          DEC  ECX
000000018000100e:  85 c9          TEST ECX,ECX
0000000180001010:  7e 1e          JLE  180001030
0000000180001012:  0f 1f 40 00    NOP  dword ptr [RAX]
0000000180001016:  66 66 0f 1f    NOP  dword ptr [RAX + RAX*0x1]
                   84 00 00 00
                   00 00
0000000180001020:  42 8d 04 02    LEA  EAX,[RDX + R8*0x1]
0000000180001024:  44 8d 02       LEA  R8D,[RDX]
0000000180001027:  8b d0          MOV  EDX,EAX
0000000180001029:  48 83 e9 01    SUB  RCX,0x1
000000018000102d:  75 f1          JNZ  180001020
000000018000102f:  c3             RET
0000000180001030:  8b 44 24 08    MOV  EAX,dword ptr [RSP + 0x8]
0000000180001034:  c3             RET
0000000180001035:  8b c1          MOV  EAX,ECX
0000000180001037:  c3             RET
```

   What does the code actually do, or what does it compute?

   Be sure to show your work, including in your report any additional experiments you perform.

Tips and Caveats

The clearest way to show the correlation of control structure implementation is to use a control-flow graph, or CFG. This is required for certain control constructs above. If you have a tool that can already produce a CFG for you, you are welcome to use it, but doing it by hand at least once is strongly recommended. Use the following algorithm to derive a control-flow graph from a disassembled function:

CFG Algorithm	
1.	Draw a line underneath every jump or conditional jump.
2.	Draw a line above the target of every jump or conditional jump.
3.	The lines now break the function into basic blocks. Starting at the top block, draw a box, labeled with the block's starting address, then draw lines based of the possible outcomes of the final instruction of the block.
3a.	For a jump, draw one line from that box to the target box, drawing it new if necessary, representing the block whose starting address is the target address.
3b.	For a non-jump, draw one line from that box to the fall-through box, again drawing it if necessary.
3c.	For a conditional jump, draw two lines: 1 for its fall-through, and 1 for its jump. Some prefer to draw each line for a conditional jump in a different color.
3d.	For a return, do not draw any lines out of the box – it is an exit. Take care not to duplicate an existing box.
4.	Repeat Step 3 for each new box until all blocks have been drawn, and each one's final instruction has been considered. You have a CFG! You may prefer to mark the entry and exit nodes.

* Indirect/variable jumps, common for `switch` statements, are not considered in this algorithm.

You will likely find this algorithm very useful in approaching Task **??**.

Report

Completion of this lab must be substantiated by a report. The report should provide sufficient detail that I can understand your process and repeat portions, if necessary.

You may use the following examples for an `if` block and primitive variable access as a template for your lab report:

An `if` Block

We compare source and disassembly for an `if` block by compiling the source shown below to a shared library, and then invoking `objdump -d` on it. We declare empty functions `before_if`, `inside_if`, and `after_if` as marker functions to quickly identify parts of the disassembly which likely correspond to the call site in the source. Note that `-fPIC` is used for shared libraries on Linux, so we will see some `@plt` suffixes, which we ignore. We identify the portion of disassembly by using the label heading for the `demo_if` function.

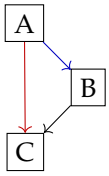
`if` block source

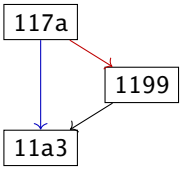
```
1  int demo_if(int c) {
2    before_if(); // A
3    if (c) {
4      inside_if(); // B
5    }
6    after_if(); // C
7  }
```

`if` block disassembly

```
117a:  endbr64
117e:  push   rbp
117f:  mov    rbp,rsp
1182:  sub    rsp,0x10
1186:  mov    DWORD PTR [rbp-0x4],edi
1189:  mov    eax,0x0
118e:  call   1070 <before_if@plt>
1193:  cmp    DWORD PTR [rbp-0x4],0x0
1197:  je     11a3 <demo_if+0x29>
                                        (Step 1)
1199:  mov    eax,0x0
119e:  call   1090 <inside_if@plt>
                                        (Step 2)
11a3:  mov    eax,0x0
11a8:  call   1080 <after_if@plt>
11ad:  nop
11ae:  leave
11af:  ret
```

We then derive the control-flow graph shown below for each. It is apparent from the graphs that Block A corresponds roughly to addresses `117a-1197`, though much of that is the function prologue. We can observe the equivalent behavior in that both blocks call `before_if` on line 2 and address `118e`. We then see the test on line 3 implemented at the end of the block at address `1193` followed by the conditional jump at `1197`, which implements the `if` statement. Note that it tests and jumps for the opposite condition than expressed in the source, because the disassembly *skips* the block whose execution requires the condition to be *true*. Block B in the source clearly corresponds to addresses `1199-119e` in the disassembly. Again, it is apparent in the graphs, but also in that both call `inside_if`. Finally, Block C of the source corresponds roughly to addresses `11a3-11af`, again accounting for the function epilogue. It is apparent in the graphs, and in that both call `after_if`.





Accessing a Primitive Variable

We now compare source and disassembly for accessing a primitive variable of various scopes and allocations: global, static, parameter, and local. We will use one source sample which demonstrates them all. Again, because this is compiled as a library with `-fPIC`, we will observe some indirection, which may require some additional explanation. We include the relevant raw data portions using `objdump -s`, isolating addresses to which the assembly refers.

Primitive variables source

```
 1  int globalVar = 6;
 2
 3  int demo_vars(int paramVar) {
 4    int localVar = 8;
 5    paramVar-;
 6
 7    globalVar = 9;
 8    static int staticVar = 7;
 9    staticVar++;
10  }
```

Primitive variables disassembly

```
1119:  endbr64
111d:  push   rbp
111e:  mov    rbp,rsp
1121:  mov    DWORD PTR [rbp-0x14],edi
1124:  mov    DWORD PTR [rbp-0x4],0x8
112b:  sub    DWORD PTR [rbp-0x14],0x1
112f:  mov    rax,QWORD PTR [rip+0x2eb2]  # 3fe8 <globalVar-0x40>
1136:  mov    DWORD PTR [rax],0x9
113c:  mov    eax,DWORD PTR [rip+0x2eea]  # 402c <staticVar.1719>
1142:  add    eax,0x1
1145:  mov    DWORD PTR [rip+0x2ee1],eax  # 402c <staticVar.1719>
114b:  nop
114c:  pop    rbp
114d:  ret
```

```
Contents of section .got:
 3fd8  00000000 00000000 00000000 00000000  ................
 3fe8  00000000 00000000 00000000 00000000  ................
 3ff8  00000000 00000000                    ........
Contents of section .data:
 4020  20400000 00000000 06000000 07000000   @..............
```

First, at `1121`, `paramVar` (in `edi` by ABI specification) is stored at stack offset −20. Then at `1124`, `localVar` (allocated at stack offset −4) is initialized to 8. `paramVar` is decremented at `112b`.

At `112f`-`1136`, `globalVar` is set to 9. This is done with some redirection: `112f` loads the value at `3fe8` into `rax`. Note `3fe8` is given in PC-relative form: `[rip+0x2eb2]`. Since `rip` points to the *next* instruction, we compute `1136` + `2eb2` = `3fe8`. `3fe8` resides in section `.got`, initialized to all `0`s. "Position-independent" code requires the OS to fix this "global offset table" up using relocations (`objdump -R`). Thus, the address of `globalVar` is written to `3fe8` by the OS. `globalVar` is allocated at `4028` as shown in the symbol table (`objdump -t`). Thus at `1136`, `rax` is `4028`, which is dereferenced as a `DWORD` (4 bytes) to store 9. Note the initial `DWORD` value of `globalVar` in the `.data` section is `06000000`, or 6 in little endian.

Static variable access is similar, but without redirection. Via PC-relative addressing, `113c`-`1145` access `402c`, the address of `staticVar`. `113c` reads its `DWORD` value into `eax`, `1142` increments it, then `1145` writes it back. Note the name in the symbol table has a numerical suffix to avoid a collision, since in C semantics, the variable has local scope.

Grading

At a minimum, each report should describe the purpose of the lab, the procedure, any assumptions, problems, or hypotheses, and the final results. It is more important that you use sound reasoning and have a well-documented process than it is to be without experimental error. Errors happen, and sometimes an experiment yields uninteresting results. If you provide good documentation, even for uninteresting or erroneous results, it will help us locate and address any problems. Please feel free to ask me questions via email. There is no penalty for asking questions, but please ask sooner rather than later, as I cannot guarantee I'll have time for everyone's questions.

This lab is due midnight after 2 weeks.

Rubric

points	Accomplishment
3	Report is well-written (organized, detailed, illustrative)
6	Analyzed the listed constructs
1	Completed Task **??**