Lab 03 : Understanding Disassembly

**Introduction:**

This lab gives us a deeper understanding of the relationship between High level code in C and low level code in assembly. For this lab, we are going to examine the assembly code of given functions and find how the code works.

**Procedure:**

For each of the following constructs we examine, we are going to create a linked library (.so/.DLL) and dump it using objdump to create a .sodump file. This is the most true-to-life scenario to be examined in real life.

1. Arithmetic Operation:

A screenshot of a computer program

Description automatically generated

Here we have simple arithmetic operations in c. The left side shows the dump file of the linked executable which was computed from arithmetic.c

1. The operations start from add block at 10f9.
2. Inside the add block we have simple move, push and add operations to add two integers.
3. The base pointer is pushed into the stack.
4. The integers are stored into edi and esi registers by referencing the rbp base pointer.
5. Both integers are moved into edx and eax registers which are added together using add move.
6. The base pointer is popped at the end of operation and functions is returned.
7. The subtraction operation is also similar to the addition operation, where one integer is stored in eax and other integer is subtracted from eax using sub move.
8. The multiplication operation is similar to subtraction where one integer is moved into eax and other integer is multiplied with it.
9. Inside the division operation, after the first integer is moved into eax, it is converted from double word (DWORD) to signed quad word in the edx register.
10. Division is performed using idiv instruction and rbp is popped.
11. The signed and unsigned operations have a few key differences.
12. The signed integers can represent both positive and negative integers whereas unsigned integers can represent only non-negative integers. This changes the division instruction from idiv to div only since there is no need to worry about signs.
13. A screen shot of a computer code

    Description automatically generatedA computer screen shot of a computer program

    Description automatically generatedIf.c

A diagram of numbers and a red line

Description automatically generated

1. The if program has a conditional block inside the program which switches based on the value of the variable inside it.
2. It starts with demo if which starts with simple push, mov and store instructions.
3. Before if function is called at 118e which calls address 1070 where before if function instructions exist.
4. After the execution of before if function, it compares the stored variable with zero and executes a conditional jump je.
5. The if block compares the integer with zero to deduce whether it is true or false. Based on the result, the conditional jumps are executed.
6. If true, it executes the statements inside if block which start at 1199.
7. In case it is false, it jumps to 11a3, and executes rest of the statements where it calls a block called after if at 1080.
8. If-else block

A computer screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generatedA diagram of a hexagon with white text

Description automatically generated

1. The if else block has two conditional jumps as shown in the conditional flow graph.
2. The program starts with standard push and mov instructions.
3. It called before\_ifelse function on address 1080 and executes before\_ifelse block.
4. Then, it compares stored variable on base pointer with zero, to find true/false and jumps according to the result.
5. Here, it jumps to 11d0 and executes instructions inside the block.
6. Alternate path for the jump would have been 11c4, which would have executed inside\_ifelse block and make another jump using jump command to 11da.
7. Here, both conditional blocks jump to 11da irrespective of the conitions.
8. Rest of the commands are executed which include a call to after\_ifelse block at 10a0.
9. A computer screen shot of a computer program

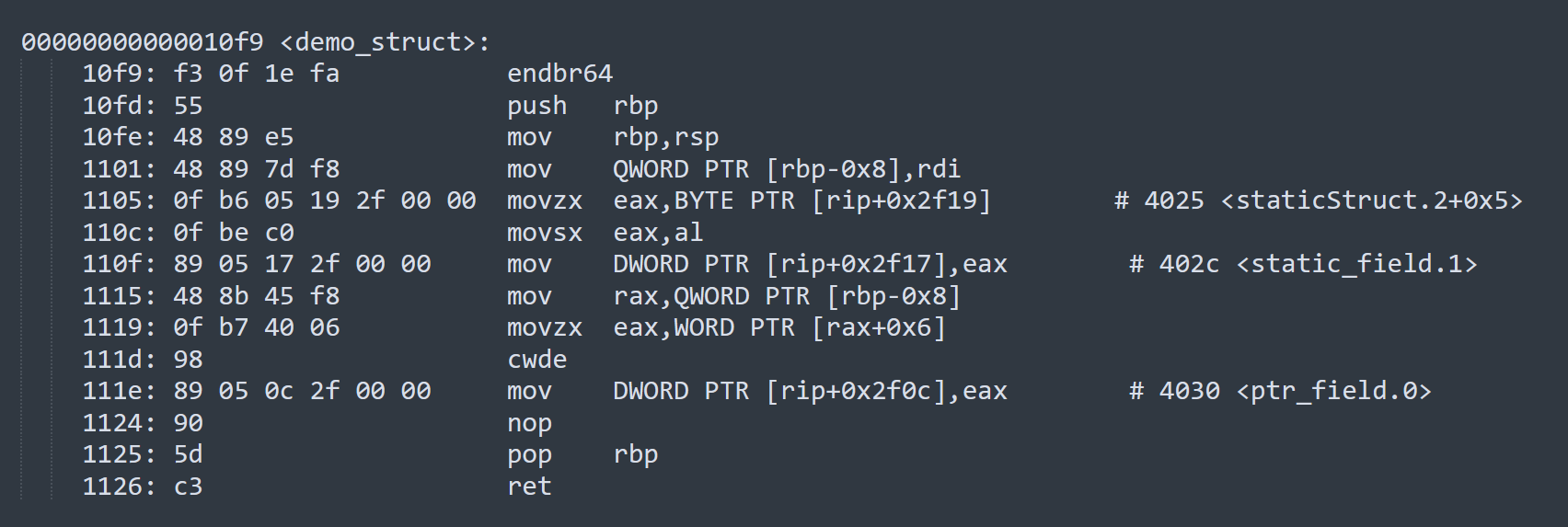
   Description automatically generatedWhile.c

A screen shot of a computer code

Description automatically generatedA screenshot of a computer

Description automatically generated

1. This program demonstrates a loop repiting itself until the condition/comparison is met.
2. It starts with push and mov instructions to store base pointer and variables.
3. It calls before-while function at 1080.
4. It makes a jump to 119f.
5. Inside 119f, we have a comparison of the stored variable with zero. The next instruction is based on the comparison result. Jne is jump if not equal to. If the comparison is fale, it would execute instruction at 1195.
6. 1195 has a mov and call instruction to another function called inside\_while at 1090.
7. It executes this loop until the jne comparison is false and stays in the loop as shown in the conditional flow graph.
8. After it exits loop, it calls after\_while at 1070 and program terminates.
9. Struct.c



A screen shot of a computer

Description automatically generatedA close up of numbers

Description automatically generated

A computer screen shot of a program code

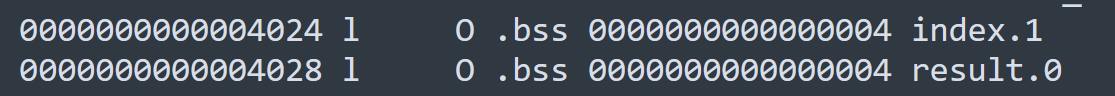
Description automatically generated

1. The struct function starts at 10f9 at demo\_struct function.
2. It starts with push and mov functions to initiate base pointers and stack pointers.
3. At instruction 1105, it accesses the element in struct at 4025 with an offset of 5.
4. That static struct starts at 4020 as shown in diagrams, where each element is stored according to the hexadecimal notation and size. For accessing the third element, it needs to have an offset of (Size of int + Size of char) = 4 + 1 = 5. That is why it refers to the staticStruct with an offset of 5 at 4020. Therefore, the element is accessed at 4025.
5. Since it is an character element, we use BYTE PTR instead of dword ptr.
6. We use movezx to indicate move with zero extension followed by a movesx that is move with sign extension.
7. Similarly, the last element is accessed from rax, with an offset of 6, and stored in eax. Her we use WORD PTR for short element which is of two bytes.
8. The function ends with pop and ret instructions.
9. Struct\_array\_combo.c

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

1. In struct\_array\_combo, we have two structs, inner struct and an outer struct. Outer struct is a struct initialized with inner struct of size 10.
2. As per .data section, there is no data stored in the struct.
3. The demo\_combo starts with push and mov instructions.
4. It loads the value of index from the static memory location at rip+0x2f19.
5. rip+0x2f19 is basically next address added with 0x2f19 which is 0x110b + 0x2f19 = 0x4024 which the compiler calculated for us.
6. It accesses the element at 4024 and stores it in edx.
7. The pointer to outerstruct is loaded into rax.
8. Then it moves edx into rdx using movsdx which takes a 32-bit register or an address to a 32-bit value and moves it sign extended into a 64-bit register.
9. The next instruction is where the access happens. Rax has ptrParamOuterStruct, rdx has index value, so it accesses the value at (rax+rdx\*8+0x9). The offset 0x9 comes with the inner structure of innerstruct where we try to access char c.
10. Movzx is used for zero sign extension and then the result is stored in rip+02f08 which is equal to 4028 as shown in the assembly code.
11. The program pops rbp and terminates.

A screenshot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

This code is computing the fibonacci series of a number using conditional loops at 1010ff and 101121

1. The code first checks if the input value (in EDI) is less than or equal to 1. If so, it immediately returns that value.
2. If the input is greater than 1, it subtracts 1 from EDI (which will be used as a counter later).
3. It then initializes some registers:
   1. EDX = 0 (loop counter)
   2. EAX = 1 (current Fibonacci number)
   3. ECX = 0 (previous Fibonacci number)
4. The main loop starts at address 0x101118:
   1. It moves the current Fibonacci number (EAX) to ESI
   2. Adds the previous Fibonacci number (ECX) to the current one (EAX)
   3. Increments the loop counter (EDX)
   4. Moves the old current number (ESI) to ECX (it becomes the new previous number)
   5. Compares the loop counter (EDX) with the input value (EDI)
   6. If not equal, it jumps back to the start of the loop
5. Once the loop is done, the final Fibonacci number is in EAX, which is then returned.