COMP 40 Laboratory: Closing the Loop on Reverse Engineering

Introduction

In today's lab, you'll develop your bomb defusing skills. The principle behind today's lab is: When reverse engineering something, if you are not certain of what you're seeing, *work the problem from both ends*. That is, create a short C program, compile it, and see if `objdump` explains the results you are seeing. Put another way: If you are unsure what is happening in the disassembled program, you can test hypotheses by coding up candidate programs in C and seeing how they compile.

For each exercise below, you will compile a C program (either one you write or one provided to you), inspect the assembly with `objdump`, and walk through the assembly with GDB to understand how the assembly realizes the C program. These exercises will help you recognize patterns when you're defusing your bomb. If you finish early, please continue working on your bomb assignment!

To get the starter code for this assignment, run

```
  pull-code defusing-lab
```

You do not have to submit anything for this lab; it is purely meant to give you more guided practice in analyzing and reverse engineering assembly.

Exercise 1: `sscanf` and pointers into the stack

This exercise will help reveal how `sscanf` works and give you experience interpreting address arithmetic related to the stack pointer. If you're unfamiliar with `sscanf`, look at the man page! If you get stuck trying to comprehend the stack manipulations, see the last section of this document for more information.

In your starter code, you have been given a header file `tul.h` that declares a struct type and a function called `read_three_unsigned_longs`. You have also been given a program `tuluser.c` that calls that function and simulates a bomb explosion if a certain condition is not met.

Your job is to implement `read_three_unsigned_longs` in a new file called `tul.c`. Make sure your function adheres to the contract specified in the header file. As reference, our implementation of `read_three_unsigned_longs` takes just 2 lines, one of which explodes the bomb.

Compile your program with the provided `Makefile` (`make tul`), run it to make sure it works as you expect, and then examine the assembly with `objdump -d`. Try walking through the assembly with GDB and understanding what components of the assembly are implementing each part of your different C files. When you and your partner are confident you understand the assembly, move on to the next exercise.

Exercise 2: Jump tables to implement `switch` statements

While some `switch` statements require the same control flow as a sequence of `if-else` statements, the compiler can generate more efficient assembly if the values being checked in each case are close together in value. Open up `aspect.c` to see a `switch` that uses the value of an `enum` type to select a case. Each name defined in this type is bound to an integer starting at 0 and counting up (i.e., `NONE` is 0, `ROT90` is 1, etc.). Thus, the `switch` is checking whether the `transformation` variable is equal to any integer between 0 and 6.

Since the values to check against are all close together, the compiler will generate a *jump table* in the resulting assembly. A jump table is an array of code addresses, each of which corresponds to the code that should be executed for a given case of a `switch` statement. The value checked by the switch is either used directly to index into this array, or it is shifted first to a new range of values (if the original range did not start at 0). The assembly then typically uses this index as the target of a `jmp` instruction with a star (`*`), instead of using a label as is typical. IMPORTANT: the star does not do any dereferencing! It is a mnemonic to inform the assembly reader that this is a special jump instruction that is using an address directly, instead of a label.

For example, if you have an instruction like:

```
jmpq *0x401b90(, %rax, 8)
```

then you can view this as first computing the address `(0x401b90 + %rax * 8)`, which is then used as the target of a `jmp` instruction. If instead you see an instruction like:

```
jmpq *%rax
```

then the instruction will take whatever is in %rax, treat it as an address, and jump to that location in the code.

To see this in action, add a `main` function to `aspect.c` that calls the `changes_aspect` function, then compile and load the resulting executable into gdb. Set a breakpoint at the `changes_aspect` function, then run the program in gdb. Gdb should then stop when you are in the function, at which point you can use the `disass` command to see the assembly around your current location in the code. Step through, printing the values of registers to see how the jump table gets you to the correct switch case.

For more explanation, this stuff is covered pretty well by Bryant and O'Hallaron (3rd edition) in Section 3.6.8. The "pidgin C" in Figure 3.22(b) may be helpful, but I would definitely check out the assembly code on line 5 of Figure 3.23, and you can see the jump table at the bottom of the page.

Exercise 3: Working with static, initialized data structures

This exercise will help you get through phase 6 of your bomb. Feel free to pause here and work on the bomb instead if you're nowhere near phase 6, then come back here when you're working on that phase if you need some assistance.

First, write C code to create a binary-search tree in the "initialized data" segment of your process memory. The tree nodes must all be global and have values assigned to them for them to end up in that segment. For example, you could create a "footwear tree", in which Townsend wears `Saucony`, Sheldon wears `Freeds`, and Ramsey wears `Birkenstock`. Follow the below steps to realize this tree and interact with it.

1. *Create a tree.*

   Your first step is to create a tree as *initialized data*. Let's suppose you use this type:

   ```
   struct node {
           const char  *person;         /* search key for binary-search tree */
           const char  *footwear;
           struct node *left, *right;   /* children */
   };
   typedef struct node *Tree;
   ```

   Go ahead and draw a search tree with three nodes. You will *define nodes from the bottom up*. For example, if Townsend's node is a leaf, you'll define (globally)

   ```
   static struct node rt {
   ```

```
        "Townsend",
        "Saucony",
        NULL,
        NULL
};
```

If the Sheldon node is a parent of the Townsend node, then the Townsend node will be a right child, and you can write

```
static struct node ms {
        "Sheldon",
        "Freeds",
        NULL,
        &rt             /* *address* of node rt makes a pointer to that node */
};
```

Add enough of these definitions to finish your global 3-node binary tree.

2. *Count nodes*

   Write a simple recursive function to count the number of nodes in a `Tree`.

   Write a `main()` function that prints the number of nodes in a tree. Pass the address of the root node.

3. *Reverse engineer*

   Compile your code, run `objdump`.

   Run your program and watch it in GDB to see how this structure is traversed.

You should be able to apply what you've learned to Phase 6.

More Information on Stack Manipulation

In class we have talked a bit about the call stack and how it works, but for purposes of analyzing stack code, your best bet is to simulate the execution of the code, keeping track of the state of the stack and the registers. A good example program would be the "three unsigned longs" code in the exercise above. Here's what's useful to know:

- The call stack grows downward, so younger activations are at smaller addresses.
- Just before a call, the stack pointer `%rsp` must be a multiple of 16 (aligned on a 16-byte boundary).
- The call instruction pushes the return address (a pointer to the next instruction of the caller after the call), so just after a call—*and therefore at the beginning of every procedure*—the stack pointer points to the return address, and it is equal to 8 modulo 16. If a function therefore needs to make another call, it has to restore the invariant that the stack pointer before a call is a multiple of 16. This requirement is the source of the mysterious code in some functions that subtracts 8 from the stack pointer without actually using any space on the stack.
- Any `push` instruction subtracts from the stack pointer and writes a word to the stack, all at one go. A `push` instruction is often used to save the value of a register; the value can be restored with a corresponding `pop` instruction.

  Why? Certain registers are *preserved across calls*. A function is allowed to use those registers only if it guarantees to save their values on entry and restore them on exit. The ones you are most likely to see are `%rbx` and `%rbp`.
- Keep in mind that although *the stack pointer may move*, the values on the stack do not move. Your best bet is to draw a picture of the stack and to give a name to each location you find there. You can then "play computer" and execute the code by hand, using the names of the locations to see what the code is doing.