COMP 40 Laboratory: From C to assembler

October 28, 2011

Introduction

In today's lab, you'll have three options:

- If you have not yet solved the early phases, or if you have questions about what you're seeing, the short, focused exercises below should help.
- If you've solved the early phases, try the binary-tree exercise on page 3.
- If you are solving phases with no problems or questions, you can work directly on your bomb with the advice and support of the course staff.

The principle behind today's lab is that when you're reverse engineering, 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.

`sscanf` and pointers into the stack

If you are not sure of how `sscanf` points, or if you are not understanding address arithmetic related to the stack pointer, then this exercise is for you.

Write and compile a function that reads three unsigned long integers:

```
struct three_unsigned_longs {
  unsigned long a, b, c;
};

void read_three_unsigned_longs(const char *inputline,
                               struct three_unsigned_longs *p);
  /* uses sscanf to read three unsigned long integers from
     'inputline', and places the results in the fields of the
     structure pointed to by p */
```

(My implementation of `read_three_unsigned_longs` takes just 3 lines, two of which are used to explode the bomb.) Examine the results with `objdump -d`. Look at your own code and also at the `main` function.

You can get the header file above, a test `main`, and a compile script by

```
  git clone /comp/40/git/code
```

Put your function in file `tul.c` and compile with `compile-code-lab`.

`jmp *something` and the `switch` statement

If you see a `jmp` instruction with a star (`*`), you are seeing a *computed goto*, which most likely has to do with a `switch` statement. This stuff is covered pretty well by Bryant and O'Hallaron in Section 3.6.6.[1] The "pidgin C" in Figure 3.15(b) may be helpful, but I would definitely check out the assembly code on line 4 of Figure 3.16, and you can see the jump table on the next page. Everything in your code is similar except that your machine uses 64 bits instead of 32 bits:

- `%eax` becomes `%rax`.
- a 32-bit pointer (4 bytes) becomes a 64-bit pointer (8 bytes), so the multiplier in the effective address becomes 8, not 4.

If you have an instruction like

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

Then you can view this as indexing into an array of 64-bit pointers located at `a` = `0x401b90`, and you are computing `a[rax]`, which in address arithmetic translates to `a + 8 * %rax`.

It's possible to reconstruct the array of pointers by using

```
objdump -s -j .rodata bomb99
```

but *I recommend against this method*. It is far easier just to get the debugger to show you the target address; put

```
((void *)0x401b90)[$rax]
```

in the DDD window and click the flashlight. Or use the "Data::Memory" tool.

If after consulting the book, you are still not sure what is going on, you might want to compile and objdump the file `aspect.c`:

```
#include <stdbool.h>

enum tx { NONE, ROT90, ROT180, ROT270, FLIPH, FLIPV, TRANS };

bool changes_aspect(enum tx transformation) {
  switch (transformation) {
    case NONE:   return false;
    case ROT90:  return true;
    case ROT180: return false;
    case ROT270: return true;
    case FLIPH:  return false;
    case FLIPV:  return false;
    case TRANS:  return true;
  }
  return false;
}
```

[1] References to Bryant and O'Hallaron are to the *first* edition.

A binary tree, interesting for the last phases

Write C code to create a binary-search tree in initialized data. For example, in the "footwear tree", Brodley wears `aboot`, Hescott wears `aloafer`, and Ramsey wears a `Birkenstock`.

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 Hescott's node is a leaf, you'll define

   ```
   static struct node nh {
     "Hescott",
     "loafer",
     NULL,
     NULL
   };
   ```

   If the Ramsey node is a parent of the Hescott node, then the Hescott node will be a left child, and you can write

   ```
   static struct node nr {
     "Ramsey",
     "Birkenstock",
     &nh,   // *address* of node nh makes a pointer to that node
     NULL
   };
   ```

   Finish the 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 node 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 DDD.

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

Push, pop, and computation on the stack

In Monday's class we'll talk 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 a pointer to the return address, 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`; your overview handout has the complete list.

- 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.