CPSC 213: Assignment 5

Last Modified: October 8, 2:15pm (minor typos).

Due: Sunday, October 15, 2014

After an eight-hour grace period, no late assignments accepted.

Goal

This assignment has two main goals. First, you will examine how programs use the runtime stack to store local variables, arguments and the return address. You will do this using a set of snippets and then you will mount a buffer-overflow, stack-smash attack on an SM213 program.

Second, you will examine a few SM213 programs to determine what they do. Assembly-code reading is actually a bit harder than writing. And so, this is the next step in deepening your understanding of this low-level approach to program. Your goal will be to add high-level comments to the lines of code and to show an equivalent C program. That is, you are the de-compiler this time.

Groups

You may do this assignment in groups of two if you like. If you do, be sure that you both contribute equally to the assignment and that you each work on every part of the assignment. Do ***not*** split up the assignment in such a way that one of you does one part and the other does another part. Keep in mind that the core benefit to you of doing the assignment is the learning that happens while you do it. Each assignment is worth only around 1.8% of your grade, but what you learn while doing the assignment goes a long way to determining the other 85%.

If you choose to do the entire assignment with a partner, submit one solution via **`handin`** and list both of your names, student numbers, and computer-science login ids in the **`README`** file.

Alternatively, you can also choose to collaborate with another student for a portion of the assignment and have your work marked individually. Do do this, simply submit your own version via **`handin`** and list the other student as a collaborator in your **`README`** file. Just don't do this if you and your partner are turning in identical solutions, as we would like to realize marking efficiency where possible. You may have at most one collaborator for the assignment; i.e., you can not collaborate with one student for one part of the assignment and another student for another part.

Provided Code

The file http://www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a5/code.zip contains the code you will use this week. It includes the following files

- **`S7-static-call.{java,c}`** and **`S7-static-call-{reg,stack}.s`**
- **`S8-locals-args.{java,c,s}`**
- **`S9-args.{java,c}`** and **`S9-args-{regs,stack}.s`**
- **`copy.c`**
- **`A5-{a,b}.s`**

Part I: *The Stack*

Here are the requirements for Part I of this week's assignment.

1. Carefully examine the execution of **`S7-static-call-stack`** in the simulator and compare it to **`S7-static-call-regs`**. Unlike previous assignments, do not document what you see. Run the snippets and look carefully at what happens. Then, do the following:
   a) Describe, in high-level, plan English, the difference between the two approaches.
   b) List one benefit of each approach.
2. Carefully examine the execution of **`S8`** in the simulator. Do not document what you see. Do the following with **`S8-locals-args.s`**:
   a) List the line(s) of **`b`** that allocate its stack frame.
   b) List the line(s) of **`b`** that de-allocate its stack frame.
   c) Modify the file to make the following two changes. Change **`b`** so that it has three arguments and change **`foo`** to call **`b (0, 1, 2)`**.
3. Carefully examine both versions of **`S9`** in the simulator. Do not document what you see. Do the following with **`S9-args-*.s`**:
   a) Compare the two versions to list the memory accesses that **`stack`** makes that **`regs`** doesn't make. Group similar instructions and give a high-level, English description of each group.
   b) How many more memory accesses does **`stack`** make, compared to **`regs`**?
4. Write a simple SM213 assembly-language program that copies a 0-terminated array of integers (use Snippets 8 or 9 as a guide). Call this program **`copy.s`**.

   As shown below, the input array should be stored in a global variable called **`src`**. The destination array should be a local variable (i.e., stored on the stack). You need two procedures: one that copies the array, one that initializes the stack pointer and calls the copy procedure. Ensure that the **`copy`** procedure saves r6 (the return address) on the stack in its prologue and restores it from the stack in its epilogue, as shown in class.

   Here is a C template for the program, called **`copy.c`**.

```
int src[2] = {1,0};
```

```
void copy() {
     int dst[2];
     int i = 0;
     while (src[i] != 0) {
          dst[i] = src[i];
          i++;
     }
     dst[i]=0;
}
int main () {
     copy ();
}
```

5. Devise a buffer-overflow attack on this program that sets the value of every register to -1 and then halts. You are suck with a similar set of restrictions that a real attacker confronts. You may not modify the program in any way other than to change its input. In this case the input is stored in the string **`src`**. So, you can change **`src`**. You will need to make it bigger; you can make it as big as you like. This input array must contain the virus program and other values so that the while loop in **`copy`** overwrites the return address on the stack in such a way that **`copy`** jumps to the **stack address** of the virus when it returns (it may not jump to any part of the array **`src`**). Run your attack in the simulator and then describe how it works.

Part II: *Reading Assembly Code*

Here are the requirements for Part II.

1. Examine **`A5-a.s`** and its execution in the simulator. Add a comment to every line that explains what that line does in as high-level a way as possible. Ideally most comments will look like c statements; comments on load and store instructions, for example, must refer to variable names. Assign meaningful names to variables and procedures. Add labels using these names. Use snippet comments as a guideline. Then, write an equivalent C program called **`A5-a.c`** that is the most likely candidate for the file that was compiled into **`A5-a.s`**. Use the same variable and procedure names in this program that you used in the assembly-file comments. Ensure that there is a correspondence between lines in the assembly file and lines of the C program.
2. Do the same for **`A5-b.s`**.

What to Hand In

Use the **`handin`** program. The assignment directory is **a5**.

1. A single file called "**`README.txt`**" that includes your name, student number, four- or five-digit cs-department undergraduate id (e.g., the one that's something like a0b1).

2. If you had a partner for the entire assignment, turn in only one copy of your joint solution under one of your ids and list both student's complete information in the **`README.txt`** file and include the text "**`PARTNER – MARK JOINTLY`**".
3. If, on the other hand, you collaborated with another student for a portion of the assignment, but your solutions are not identical and you want them marked individually, each of you should include the other student's complete information in your **`README.tx`**t file, include the text "**`COLLABORATOR – MARK SEPARATELY`**", and turn in copies separately via **`handin`**.
4. In **`README.txt`**, answer questions 1a, 1b, 2a, 2b, 3a, and 3b.
5. The file **`S8-locals-args.s`**, modified as specified in question 2c.
6. Your buffer-overflow program and the value of **`src`** you used to mount the attack in the file named **`copy.s`**.
7. A plain-English description of your virus and how it works in **`README.txt`**.
8. Your modified **`A5-a.s`** and **`A5-b.s`** your C files **`A5-a.c`** and **`A5-b.c`**.