CPSC 213 – Assignment 7

Stacks and Polymorphism

Due: Friday, October 27th, 2017 at 11:59pm
Grace period extends to Saturday at noon, no late assignments accepted after that.

Goal

The first part of the assignment is about the stack. First you will examine how programs use the run-time stack to store local variables, arguments and the return address. You will do this using a set of snippets and you will answer questions about their execution. Then, you will examine two SM213 programs that contain procedure calls to determine what they do. Finally, you will mount a buffer-overflow, stack-smash attack on a SM213 program.

The second part of the assignment is about dynamic flow control. You'll also do a bit more with dynamic control flow in Assignment 8. First, you will implement the last two instructions in the SM213 ISA. By now this should be quite straight forward. Then, you will examine the code we covered in class that models Java polymorphism in C. You have Java, C and assembly versions of this file. You will use this code as a starting point to implement your own simple "Java program" in C, using our model of polymorphism to handle the polymorphic dispatch contained in the Java program.

As with all assignments, you are encouraged to do this in groups of two. Just be sure that you both do all of the work, helping each other. Don't split up the work so that you both do half … unless, of course, you're shooting for a 50% on the final :).

Part 1 – The Stack

The file *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip* contains the following files.

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

Evaluating Snippets using Simulator

Familiarize yourself with snippets 7-9 by running them in the simulator and asking yourself the following questions. This part is not for marks.

- Carefully examine the execution of `S7-static-call-stack` in the simulator and compare it to `S7-static-call-regs`. Run the snippets and look carefully at what happens. Ask yourself these questions.

    a) What is the difference between the two approaches?

    b) What is one benefit the approach followed in `stack`?

    c) What is one benefit of the approach followed in `reg`?

- Carefully examine the execution of `S8-locals-args.s` in the simulator. Ask yourself these questions.

    d) What lines of `foo` and `b` allocate `b`'s stack frame?

    e) What the lines of `foo` and `b` de-allocate `b`'s stack frame?

  The next two questions ask you to consider changing `b` so that it has 3 arguments and 4 local variables. Ask yourself these questions.

    f) What changes required in `b` to add the arguments and locals; note that you do not actually use these new variables in any way?

    g) What changes required in `foo` to call `b(0,1,2)`?

- Carefully examine both versions of `S9-args-stack.s` and `S9-args-regs` in the simulator. Ask yourself these questions.

    h) What memory accesses does `stack` makes that `regs` doesn't make?

    i) How many more memory accesses does `stack` make, compared to `regs`?

Questions 1 and 2 [30%]

The next two question use these files found in code.zip.

- `q1.s`
- `q2.s`

Answer the next two questions by modifying the `.s` files and by writing `.c` files. The `.c` files must compile and execute. When they execute they must perform the same computation as the `.s` file and print out the value of its static variables, one per line as a decimal number (nothing other than that number on each line). This means that `q1.c` must print 10 lines of numbers and `q2.c` must print 16.

1. [15%] Examine `q1.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.

   Then, write an equivalent C program called `q1.c` that is the most likely candidate for the file that was compiled into `q1.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, but do not include the `start` procedure in `q1.c`; this procedure is added automatically by the c compiler to initialize the stack and call `main`. The end of your `main` procedure should print the value of the program's ten static variables as described above.

   You may notice a register that is used in a way that is best explained by saying that it is a local variable, even if its value is never read from or written to the stack. Avoiding these memory accesses is a common optimization compilers make for local variables.

2. [15%] Do the same for `q2.s`.

Part 1: Stack Smash Attack

The provided code contains an assembly-code template named `copy.c`. Refer to this C program to answer the following two questions.

Questions 3 and 4: The Stack Smash attack [35%]

For the next two questions, refer to the file `copy.c` found in code.zip.

3. [5%] Using `copy.c` as a guide, write a simple SM213 assembly-language program that copies a null-terminated array of integers (use Snippets 8 or 9 as a guide). Call this program `copy.s`.

   In `copy.c`, which is reproduced below, the input array is stored in a global variable named `src` and the destination array is in a local variable (i.e., stored on the stack). Your assembly code must do the same.

   As in `copy.c`, you will need two procedures: one that copies the array and 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.

   Note that this code contains a buffer-overflow bug. That is intentional. Be sure your assembly code has this bug so that you will be able to attack it in Question 4. Another thing you'll want to do is to keep the value of `i` in a register in the body of the loop. If you were to read/write it from/to the stack on every iteration, you'll find that the buffer

overflow will overwrite the value of i and thus change the way the attack string is written to the stack.

4. [30%] Modify `copy.s` to devise a buffer-overflow attack on this program. The attack should set the value of every register to -1 and then halt.

   You are stuck with a similar set of restrictions that a real attacker confronts. **You may not modify the program you have just written in any way other than to change its input (i.e., `src`).** Change `src` to make it bigger to contain virus program and other values as needed so that `copy` executes the virus program when it returns, instead of actually returning to `main`.

   You must specify the attack string (the value of `src`) using a sequence of `.long` directives. Recall that each `.long` specifies the value of 4 bytes of memory. The string will contain the virus program as machine instructions, which are either 2 bytes or 6 bytes. You will thus need to compact multiple instructions into a single `.long` and possibly also split a 6-byte instruction across two `.long`'s.

   Remember that the only change you are permitted to make to the program you wrote for Question 3 is to specify a different value for `src`.

   Run your attack in the simulator to be sure that it works.

Part 2: Polymorphism

Question 5: Implement and Test Double-Indirect Jumps [5%]

There are two remaining instructions to implement in the simulator, described below. Implement them.

Instruction	Assembly	Format	Semantics
dbl ind jmp b+d	`j *o(rt)`	`dtpp`	pc ← m[r[t] + (o == pp*4)]
dbl ind jmp indx	`j *(rb,ri,4)`	`ebi-`	pc ← m[r[b] + r[i]*4]

Then, use the simulator to examine the snippet `SA-dynamic-call` that you will find in this week's code file at *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip*. There you will also find the `CPU.java` solution from Assignment 6, which you can use as a starting point for this question, if you like. Replace the "`TODO`" comments with your code.

Background: Manipulating Strings in C

You will be manipulating strings in your C program in Question 6. Strings in C are more troublesome, by a long way, than strings in Java, due entirely to the dynamic-allocation issues we have been talking about; i.e., deciding what part of code is responsible for freeing something that is malloced from the heap.

A C string is stored in an array of characters. The string itself, which can be smaller than the array that contains it, is terminated by the first `null` (i.e., 0). You'll need to consider a couple of issues.

First, when a procedure receives a string as an input parameter, if it procedure stores that string, it should make its own copy of the string rather than storing the string pointer. Storing the pointer is dangerous because the caller could free the object following the call and thus turn this stored value into a dangling pointer. By copying, the procedure ensures that its copy of the string is safe from whatever its caller does with the string after the call returns.

You will want to use the standard method called `strdup` to do this (see its man page; e.g., via google or by typing "`man strdup`" at a unix command line). You will also need to add "`#include <string.h>`" to the beginning of your file.

For example your code should look like this:

```
void bar (struct X* anX, char* string) {
    anX->string = strdup (string);
}
```

And **not**, as it would in Java, like this:

```
void bar (struct X* anX, char* string) {
    anX->string = string;
}
```

Because if you did this, a caller that does the following creates a dangling pointer:

```
void foo (struct X* anX) {
    char string[] = "Hello World";
    bar (anX, string);
}
```

Similarly, as we have examined, it is often better to avoid writing a C procedure that returns a pointer to an object that it dynamically allocates. Instead, if it can, it should leave it to its caller to perform the dynamic allocation and simply copy its result to that location.

For example, Java code that looks like this:

```
String getString () {…}
```

Would in C look like this:

```
void getString (char* buf, int bufSize) {…}
```

Where `buf` is a pointer to memory provided by that caller into which `getString` copies its result up to the limit of `bufSize` bytes.

Finally, you will need to convert numbers to strings and to concatenate strings. The easiest way to do this is with the standard procedure called `snprintf` that uses `printf` formatting to write to a string. So, for example if you wanted to create the string "`Hello World 42`" from the string "`Hello World`" and the integer `42`, you would do something like this:

```
char buf [1000];
snprintf (buf, sizeof (buf), "%s %d", "Hello World", 42);
```

Question 6: Modelling Polymorphism in C [30%]

In *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip* you will find a file named `Poly.java`. This file contains a simple example of polymorphism using two classes: `Person` and `Student`, which extends `Person`. Write a new C program called `poly.c` that does the same thing in C. Start by copying `SA-dynamic-call.c`. Then make the necessary changes to replace `A` and `B` with `Person` and `Student`.

Follow the approach outlined in class. For example solution should include the following structs: `Person_class`, `Person`, `Student_class`, and `Student`. It should have the following static objects: `Person_class_obj` and `Student_class_obj`. And, it should have the following methods: `new_Person(char*)` and `new_Student(char*,int)`.

The class `Poly` contains two procedures that you need to implement as well, but there will be no `Poly` class or anything like it in your C program. Notice that these are both `static` procedures. One is, of course, `main`. And the other, `print`, is function that contains a polymorphic dispatch to the method `toString`.

The Java code uses the Java class `String`. You must use C strings instead. The Background section has a few pointers on using strings in C. Note that the `toString` procedure should follow the guidelines listed above and that we've discussed in class. It must have the following type signature to pass the handin test.

```
void xxx_toString (void* thisv, char* buf, int bufSize)
```

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs213/a7`, it should contain the following ***plain-text*** files.

1. `README.txt` that contains the name and student number of you and your partner

2. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). Your partner should not submit anything.
3. For Questions 1 and 2: `q1.s`, `q1.c`, `q2.s`, and `q2.c`.
4. For Question 3 and 4: `copy.s`.
5. For Question 5: `CPU.java`.
6. For Question 6: `poly.c`.