CPSC 213 – Assignment 7

Stacks and Polymorphism

Due: Saturday, March 9, 2019 at 11:59pm
After an 8-hour, no-penalty, grace period, no late assignments accepted.

Goal

The first part of the assignment is about the stack. 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 next week. First, you will implement the last two instructions in the SM213 ISA. By now this should be quite straight forward.

Then, you will extend a C program that uses java-style polymorphism, but implemented explicitly using C function pointers as we have discussed in class. In doing so, it will be helpful to review the code we covered in class, which is posted to Piazza and included with this assignment as Snippet A.

Part 1: Stack Smash Attack

Questions 1 and 2: The Stack Smash attack [45%]

For the next two questions, refer to the file `copy.c` found in www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip.

1. [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 from Assignment 6 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 2. 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.

2. [40%] 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 1 is to specify a different value for `src`.

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

Part 2: Polymorphism

Question 3: 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.

Question 4: Modelling Polymorphism in C [50%]

In www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip you will find two directories: `polytree-java` and `polytree-c` that implement a binary search tree.

Carefully examine the Java code; compile and run it. You will see that it consists of a base class called `Node` that implements the tree and two subclasses called `IntegerNode` and `StringNode` use `Node` to stored sorted lists of integers and strings respectively. Look at the `main` method to figure out how to call the `PolyTree main` from the command line with lists of either integers or strings.

Now examine the C code. It has complete implementations of `Node` and `StringNode` and templates for the rest of the work you will do. Examine this code and the `Makefile` carefully. Compile it by typing `make` and run it. You will notice that the implementation consists of both `c` (source code) and `h` (header / interface) files. One new thing you'll see in the header files is a variable declared with the `extern` keyword added to its type; this is how you allow multiple files to reference a global variable declared in one of them.

Now you are going to extend this code in various ways. Take this step by step as described below. Get each step working before moving to the next one.

1. Create a new class called `ReverseStringNode` that *extends* `StringNode` by *overriding* only the `compareTo` method to sort strings in reverse. All of `ReverseStringNode`'s functions except for `compareTo` must come from (i.e., delegate to) its super classes: `StringNode` and `Node`. Place your solution in the files `reversestringnode.c` and `reversestringnode.h` and modify `polytree.c` in the `TODO` location to use the `ReverseStringNode` class when the "`r`" option is specified on the command line.

2. Create a new class called `LoggingStringNode` that *extends* `StringNode` by *overriding* the `insert` method to print "`insert x`" each time `insert` is called (where "`x`" is value of the node begin inserted without the quotes). Note that this means that a node may print multiple times in the process of being inserted since `insert` is recursive.

   For example, if the tree contains a single node with value "`D`", inserting a second node with value "`B`" would print "`B`" once: the call from polytree and no recursive call. Inserting a third node with the value "`A`" or "`C`" would print the value twice (one recursive call), but inserting "`E`" would print "`E`" only once.

Your solution should call `Node`'s implementation of `insert`. In Java you would do this by calling `super.insert(node)`. Note that this is actually a *static* call (i.e., not polymorphic). The Java compiler implements this call by locating the nearest ancestor class that implements the target method, which in this case will be `Node_insert`. So, your solution can just call this method directly.

Place your solution in to files provided and update `polytree.c` to use this class with the "`l`" option.

3. Implement the class `IntegerNode` ensuring that every instance method in the Java version of the class is implemented using the C-based polymorphic structure so that, for example, another class could someday extend `IntegerNode` and override or inherit any of these methods.
4. If you examined `polytree.c` very carefully will you have noticed that it has a memory leak. It never frees the tree nodes that it allocates inside of the for loop. Fix this problem by adding a `delete` method to the `Node` class that calls `free` on a node and all of its descendants in the tree. Then call this method at the end of `polytree main`. Your code must be free of all memory leaks (check with `valgrind`).

What to Hand In

Use the `handin` program.

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

5. `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.
6. For Question 1 and 2: `copy.s`.
7. For Question 3: `CPU.java`.
8. For Question 4: all of the `c` and `h` files in `polytree-c`, modified appropriately. **DO NOT PLACE THEM IN A SUBDIRECTORY** as they are in the code pack. Please just place them directly in your handin directory. You handin directory should have no subdirectories (i.e., there is no `polytree-c`).