CPSC 213 – Assignment 7

Stacks and Polymorphism

Due: Saturday, July 27, 2019 at 11:59pm
You may use one of your two late days on this assignment to make it due Sunday.

Learning Objectives

Here are the learning objectives of this assignment, which will be examined in this week's quiz. They are a subset of the unit learning objectives listed on Slide 2 of Unit 1c.

After completing this assignment you should be able to:

1. write assembly code that uses system calls;
2. describe how a buffer-overflow, stack-smash attack occurs;
3. explain the difference between static and dynamic procedure calls in C and assembly language;
4. write C programs that use function pointers;
5. explain how polymorphism can be implemented in C; and
6. convert Java instance-method call into equivalent C code that uses function pointers.

Goal

The first part of the assignment is about the stack. You will mount a buffer-overflow, stack-smash attack on a SM213 program.

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.

You'll need the code handout for this assignment, available here: *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip*. You will also need the system call enabled SM213 simulator, available here: *www.ugrad.cs.ubc.ca/~cs213/cur/resources/SimpleMachine213WithSyscalls.jar*.

Part 1: Stack Smash Attack

Examine System Call Code

The code handout contains a directory called `examples`. Inside this directory, you will find several programs which use system calls, in both C and SM213 assembly form. The C files can be compiled and run on the student servers, and the SM213 programs can be run in the syscall-enabled simulator. Run each example in the simulator and see how the system calls interact with the registers and memory. There is nothing to hand in for this part.

Question 1: Develop Shellcode [20%]

Your first task as an Evil Hacker™ is to develop your very own *shellcode*. Shellcode is binary code that, when executed by a CPU, uses a system call to launch an interactive shell, giving an attacker free rein over the system. Thus, the ultimate goal of an attacker is to inject their shellcode into a program and get it to run. The Morris worm code shown in class is an example of shellcode for the VAX platform.

On UNIX systems, like the Linux student servers, the shell program is named */bin/sh*. Therefore, you will develop a piece of code that will execute this code on demand.

Shellcode, when executing, cannot make any assumptions about the environment (*e.g.* it cannot rely on any data already in memory besides itself). This makes developing shellcode challenging sometimes. Follow the following steps to produce your very own shellcode.

1. Write a piece of assembly code that uses system call #2 (exec) to launch */bin/sh*. Refer to the examples to see how the system call works. Don't use `.pos` in your code, and don't load or use any absolute addresses - your shellcode must work no matter what address it is loaded at.
2. Convert your assembly code into machine code. You can do this with the simulator, by running
   `java -jar SimpleMachine213WithSyscalls.jar -d shellcode.s`
   This will generate a `.machine` file, which lists the contents of memory in hex.
3. You can use the following command to convert your machine code file into actual binary code:
   `cut -d: -f2 shellcode.s.machine | xxd -r -p > shellcode.bin`
   For this to work correctly, your code must not use `.pos` commands, otherwise the file will be padded with a lot of null bytes.
4. The assembly program `q1test.s` reads shellcode using the read system call, then executes it. You can play with this in the simulator (which shows you the instructions changing to new ones), but you won't be able to input raw binary into the GUI input box. Instead, you should use the CLI mode to test your shellcode:

```
cat shellcode.bin | java -jar SimpleMachine213WithSyscalls.jar
-i cli q1test.s
```
If your shellcode worked, this should print "`<<<WOULD EXECUTE /bin/sh>>>`", indicating that your shellcode would have successfully launched a shell.

If you succeeded in step 4, you will submit your machine-code file as **shellcode.machine** for this part of the assignment.

Question 2: Exploit Buffer Overflow [40%]

Now, it has come time to hack a real program. In the code handout, `q2vuln.s` is a program which uses the read system call to read some data. It contains a vulnerability.

1. Reverse-engineer this program into C, and **submit it as q2vuln.c**. Use the examples as a guide for how to reverse the system calls into C.
2. Now, you need to take control over this program's return address using a stack smash attack. Conveniently, there's a function called *proof* that, when called, prints out a secret message. To prove that you have control over the return address, write a string that will cause the program to jump to *proof* when it tries to return from the main function.
   Write the string in hexadecimal, then convert it to binary when feeding it to the program, like so:
   ```
   xxd -r -p q2proof.hex | java -jar
   SimpleMachine213WithSyscalls.jar -i cli q2vuln.s
   ```
   Submit your hexadecimal-encoded attack string as **q2proof.hex**.
3. Finally, combine your shellcode with your controlled return address. Embed your shellcode inside the attack string, and set the return address to point into it - noting that the address of the stack is fixed and known. The simulator can help you figure out the right addresses to use. If you're successful, the simulator should show the same `<<<WOULD EXECUTE /bin/sh>>>` output as before when running `q2vuln.s` with your input. Submit your new, final attack string as **q2exploit.hex**.

BONUS: Exploit Buffer Overflow Remotely [+15%]

The vulnerable program `q2vuln.s` is actually running on a server on the Internet, ready to be exploited by all of you. Your goal is to get a real, functioning remote shell on this server. The simulator is run in a special mode (`SIMPLE_MACHINE_ALLOW_EXEC=1`) which causes exec system calls to be routed to the real OS - so if you can get a shell on this machine, you will actually gain control of a real system!

The server address is **TBA, see Piazza**. To connect, you may use the *nc* command, like so:

```
nc <server> <port>
```

Just like the simulator, you can pipe data into the connection like so:

```
xxd -r -p q2exploit.hex | nc <server> <port>
```

In order to take advantage of the remote shell that opens after your exploit runs, you will need to get a little creative. Since this is a bonus question, this is left as an exercise to the reader.

After you get a functioning remote shell, you will have an opportunity to *capture a flag* from the server, stealing a little piece of secret data from it, by running a special program called *getflag*. This program is only accessible if you have a shell. Run it, figure out how to use it, and obtain your flag.

Submit the flag you got in `flag.txt` for bonus points!

Part 2: Polymorphism

Example the Execution of Assembly Snippets

The handout contains the files:

- `SA_dynamic_call.java`
- `SA-dynamic-call.{c,s}`

You can compile and run the Java and C programs. Run the assembly program through the simulator. Carefully observe and understand what happens as these snippets execute so that you could explain this code to someone if they asked. This step is not for marks.

Note that these implement a more complex polymorphic hierarchy than what your assignment requires, but it's good to know how the complex polymorphism of Java is implemented in a lower-level language like C.

Question 3a: Modelling Polymorphism in C [30%]

In the code handout, under the `q3` directory, you will find an updated version of the list and tree problem from Assignment 5. This new version is supposed to support integer *and* string elements. Carefully examine all the source code you've been given.

We've provided a new version of the reference counting code, as discussed in class. Our new version supports *finalizers* - bits of code that will execute when the reference-counted object is about to be destroyed. You'll need this feature in this assignment.

`element.h` provides the interface for all elements - basically, a base class. It has two concrete implementations: `str_element` and `int_element`. Your job is to provide the code for both

subclasses. You may *only* modify `int_element.c` and `str_element.c`. You can implement these any way you like, so long as you use the provided reference counting library.

Work on your implementation step-by-step:

4. First, comment out all the code in `main()` except for the call to `test_int()`. Implement enough of the `int_element` class such your program compiles, runs, and prints 42. Test your code carefully, and run it through *valgrind* to make sure you have no leaks.
5. Next, comment out everything in `main()` except `test_str()`. Start implementing the string class until it prints out `Hello, World!`. Run *valgrind* again to make sure you have no leaks.
6. Implement the rest of the public interface for both classes, which includes the instance functions from `element` and the static functions listed in each header file.

Comment: In normal sort implementations, comparisons between mixed types would not be allowed, as comparisons between multiple different types aren't usually defined (is a *structure* less than a *string*? How about an *int*?). However, for this problem, since we have only two types we can declare that "integers are smaller than strings".

Question 3b: Using Polymorphism in C [10%]

Under the `q3` directory you will also find a file called `sortmain.c`. You should implement this program such that it sorts the arguments as elements, placing the integers first (as `compare` would do). You must use the C function `qsort` to do so. `qsort` is a built-in C function which sorts an array, given a suitable comparison function. You should use it to sort a *dynamic array of element pointers*, by giving it a comparison function that calls the element's virtual compare function. You may need to do some pointer casting to make it all work out.

You do *not* have to have implemented part 3a to do this part; the autograder will use fully-implemented versions of the `int_element` and `str_element` classes to test your code. As usual, you must ensure that your code is free of memory errors and leaks using valgrind.

What to Hand In

Use the `handin` program.

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

1. (optional) `PARTNER.txt` containing your partner's CWL login id and nothing else. Your partner should not submit anything.
2. For Question 1: `shellcode.machine`
3. For Question 2: `q2vuln.c`, `q2proof.hex`, `q2exploit.hex`

4. (optional) For the bonus, `flag.txt`
5. For Question 3: `int_element.c`, `str_element.c`, and `sortmain.c`.