CPSC 213 – Assignment 7

Stacks and Viruses

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

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 1e.

After completing this assignment you should be able to:

1. write assembly code for procedure call arguments passed on the stack or in registers, with and without a return value
2. write assembly code for a procedure with and without local variables, with arguments pass on the stack or in registers, with and without a return value
3. write assembly code to access a local scalar, local static array, local dynamic array, local static struct, and local dynamic struct;
4. describe how a buffer-overflow, stack-smash attack occurs, and be able to mount one; and
5. describe why this attack would be more difficult if stacks grew in the opposite direction (i.e., with new frames below (at higher addresses) older ones).

Goal

In this assignment you will engineer virus attacks on *SM213* programs. You will start with a simple attack and end by getting an innocent, but buggy, *SM213* program to run a shell that you control remotely.

The code for this assignment is found in *www.students.cs.ubc.ca/~cs-213/cur/assignments/a7/code.zip*.

You may use a partner for this assignment. If you do, you must both contribute significantly to every question. You must not split up the work between the two of you. One of you should submit your combined work with `handin` and the other person should submit nothing.

Part 1: A Simple Attack [40%]

The first two questions, refer to the file `copy.c` in the provided code.

Question 1: *Build a Simple, Vulnerable Program [0%]*

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.

```
# 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();
}
```

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 `copy` 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.

Question 2: *Mount an Attack on that Program [40%]*

Modify your `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 constraints 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 and 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: Launching a Shell [60%]

You are now ready to launch a more dangerous exploit: to get an innocent program to launch a shell that you can use to run arbitrary programs on the infected machine.

The code file contains a directory called `examples`. Inside this directory, you will find several programs that 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 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.

Note that all system calls return a value in `r0`, which for `read` and `write` is the number of bytes read or written respectively, and for `exec` is the status of the executed program (0 for success). All system calls return -1 if they fail (e.g. user canceled input).

Question 3: *Develop your "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 `sys $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. Test your shellcode in the simulator to make sure it works.
2. Convert your assembly code into hexadecimal machine code by following the instruction formats for each instruction. For example, you would translate the instruction `mov r1, r2` as `6012`.
3. The provided assembly program `q3test.s` reads input using the read system call (`sys $0`), then executes it as code. You can play with this in the simulator, inputting your hexadecimal machine code by checking the "Hex Input" checkbox.
4. Once you're satisfied with your shellcode, try it against the CLI simulator. Save your hexadecimal machine code in a plain text file called `shellcode.hex`, use `xxd` to convert the hex into binary, and send it into the simulator's input as follows:

   ```
   xxd -r -p shellcode.hex | java -jar SimpleMachine213.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, submit your original assembly code as **`shellcode.s`** and your hexadecimal machine code as **`shellcode.hex`** for this part of the assignment.

Question 4: *Exploit the Buffer Overflow [40%]*

Now, it's time to hack a real program. In the code handout, `q4vuln.s` is a program that uses the read system call to read some data. It contains a vulnerability.

1. [10%] Reverse-engineer this program into C, and submit it as **`q4vuln.c`**. Use the examples as a guide for how to reverse the system calls into C.
2. [10%] 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 this:

```
xxd -r -p q4vuln.hex | java -jar SimpleMachine213.jar -i cli q4vuln.s
```

Submit your hexadecimal-encoded attack string as **`q4proof.hex`**.

3. [20%] 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 `q4vuln.s` with your input. Submit your new, final attack string as **`q4exploit.hex`**.

BONUS: *Launch a Remote Attack [+10%]*

The vulnerable program `q4vuln.s` is actually running on a Linux server on the Internet (in an AWS datacentre), 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 java -jar …`) that 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 `35.166.142.125`, port `1337`. To connect, you may use the *nc* command:

```
nc 35.166.142.125 1337
```

Similar to using the simulator directly, you can pipe data into the connection like so (the `cat` keeps the connection open so that you can type commands that are sent to the infected program).

```
(xxd -r -p q4exploit.hex; cat) | nc 35.166.142.125 1337
```

The bonus has two parts:

1. Take control over the program's return address, as in Step 2 above. The server will print a different secret message from the secret message in the provided `q4vuln.s`; submit the server's secret message in **`secret.txt`**.
2. Send over your shellcode exploit to obtain a shell on the server. In order to take full advantage of the remote shell that opens after your exploit runs, you will need to get a little creative. 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 the special program **`./getflag`**. Run it, figure out how to use it, and obtain your flag. Submit the flag you got in **`flag.txt`**.

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs-213/a7`, it should contain the following ***plain-text*** files.

1. **`PARTNER.txt`** containing your partner's CWL login id. Your partner should not submit anything.
2. Question 1 and 2: **`copy.s`**.
3. Question 3: **`shellcode.s`** and **`shellcode.machine.`**
4. Question 4: **`q4vuln.c`**, **`q4proof.hex`**, and **`q2exploit.hex`**.
5. BONUS: **`secret.txt`** and **`flag.txt`**.