Info-I533/CSCI B-649 Assignment #1

Prof. Steven Myers
Due Date: Nov 12th

February 15, 2016

1 The Purpose

The point of these assignments is to do two things: 1) make you think over the material you've been reading and we've been discussing in class, to make sure you really understand it; and 2) Push your thinking just a little beyond that of what we have discussed in class and what was covered in the readings. This is to make you actively think of the material in new ways, so that you're not just memorizing what has been taught in the book, but actively thinking about how the material applies to other problems.

2 Expectations

The expectation is that you write the solutions to this assignment by yourself, as the goal is to demonstrate you have learned the material. You can discuss the project with colleagues, friends and AIs, but if you do you should ensure you do the following things. First, you need to list on the first page of your assignment all people with whom you've discussed the project. This is everyone including faculty, AIs, classmates, friends working on a secret bitcoin startup, etc... Second, you should not take any notes from any such discussions, not should you write up your solutions immediately after the discussion. Make sure a few hours of leisure time, where you're actively thinking of something else, between your last discussion and when you write up your solution. If you have truly learned the material, this will not prevent you from writing up solutions. If you didn't really understand, you'll have difficulty writing it up, and should probably take this as a cue to revisit the material before writing your solutions. If in doubt, please reference IU's code of student conduct (`http://www.indiana.edu/~code/`)

The expectation is that the writing is clear, well written and you've used proper grammar and spelling. Answers should be typed, with the exception of diagrams or mathematical formulas which are difficult to typeset. In these cases you can include hand-written diagrams or formulas, but they need to be clear and easy to read. In short, be professional.

3 The Goal

The goal is for you to write some basic buffer overflow attacks on some very simple programs. One goal should be to understand that the attack you'll write is so simple that by today's standards it is completely useless as an attack in practical manner on modern operating systems. However, the ideas of even the most advanced attacks are founded on the principles of this basic attack, so it is still very useful information. Further, many embedded systems (think Internet of Things) have code that has significantly less protection than modern operating systems, either because they run dated code on old systems, or because they are embedded and have little to operating system functioning. However, because of the modern defenses the first thing we'll do is disable a number of Linux protections, so that the attack will actually work. You should be able to use the Aleph One text "Smashing the stack for fun and profit" posted on the website to great advantage for this assignment. However, you will be implementing the attack on a 64-bit system.

4 Setup

Because buffer overflows are such a big security concern, and there is so much code that has these types of attacks, there has been significant effort put into stopping them. While none of these processes is foolproof, they are sufficient to stop the simple types of attacks we will be considering in this class. Therefore, we need to turn off several of the stack and memory protections:

4.1 Non-Executable Stacks

Modern linux systems actually mark the stack as non-executable except in certain situations. This means that if the program counter finds itself in the stack's memory space, if will immediately throw a system error and crash the program.

To get around this problem there are two approaches. We can run the program with a special program called `execstack` which will ensure your program has an executable stack. To get this program run:
`sudo apt-get install execstack` The run `your-program` with the command:
`execstack -s your-program`
where here the -s flag ensures that the stack is executable.

The other option is to compile the program we're *attacking* with the flag to denote that it has an executable stack: `-z execstack`

4.2 Address Space Layout Randomization

We need to turn off ASLR protection with the following command. Note that every time one reboots a system this command must be entered by a user who has a root password.
`echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`

We will discuss ASLR later in the course.

4.3 Canaries

In order to turn off stack canaries in the program being attacked we need to compile it with the following flag: `-fno-stack-protector`

Therefore, to compile the programs we will be *attacking* (i.e. *not the attack programs*), we will compile our program `victim.c` as follows:
`gcc -z execstack -fno-stack-protector -o victim victim.c`

We will discuss exactly what canaries are later in the course.

5 The Vulnerability

The theory is that you are writing code to attack the following program which is running on a machine with root permissions, but while you have access to the machine and the program that is running, you don't have root permissions on it. You are a guest, with limited permissions. In practice we're making it easier by giving you the code that is being attacked. This can be useful for debugging your attack code, and making it work. The goal of your attack code is going to be to have it open a shell.

We are going to break down the assignment into parts, slowly building towards our goal.

6 Submission

Create a directory in your github repository entitled HWA2, where you will keep all your work files. Create a sub-directory entitled submitted, where you will put your final work.

7 Part I

Your first goal is to simply write an assembly language program that will open a shell (please use /bin/sh) through a call to Linux system call execve, and then exit the program properly with a call to Linux system call exit. (Technically, the exit code is not really necessary, but as I found out the hard way, it makes debugging easier). It should do this without using any external C libraries, etc...Recall that system code call numbers are located in the file `/usr/include/x86_64-linux-gnu/asm/uninstd_64.h`.

Submission Once you have this code compiling and executing save it to a file entitled part1.s in the submitted directory. Also include a typescript, using the script tool, called Part1Demo.txt which shows you compiling the assembly program, linking it, and running it. Note you can exit the shell you started with your program by typing 'exit'.

8 Part II

Your second goal is to modify the assembly program from Part I, so that it is more suitable for use in shell code. Namely, we need to ensure that there is only one NULL (0x00) values that appear in it, the NULL that should remain is at the very end of the string "/bin/sh" that should be at the end of you code. Recall we have to take out the NULLs, as NULLs represent the end of a string in many typical input sequences, due to the C end-of-string convention. The program should still open a shell, but there should be no NULL values in the program binary code. The last NULL will be removed later. We don't remove it now, as to so properly requires the last NULL to be placed in by the program itself, making the code self-modifying. This requires certain memory protection features in Linux to be shut off, and there is no trivial way to do this.

Submission Once you have this code compiling and executing save it to a file entitled part2.s in the submitted directory. Also include a typescript, using the script tool, called Part2Demo.txt which shows you compiling the assembly program, linking it, and running it. Now use gdb to show that you have actually removed all the NULL characters (You may have to use 'apt-get install gdb' to install gdb if it is not already installed on your machine). To do this show a disassembly of your program and then display the memory (in hex) where your program is located. Recall that you can show the memory contents of the program by using the 'x/[count]x address' command, where count is the number of bytes of memory you would like to see, and address is the beginning of the memory you wish to eXamine.

9 Part III

Next we will take the memory dump of our binary (hex) code in Part II and ensure it executes correctly, by redirecting the execution of a simple C program to ensure that this works. We are also going to demonstrate the benefits of several security mechanisms that are currently deployed in linux.

9.1 Part IIIa

Use the file "BufferRedirect.c" and put your binary hex code in the array shellcode. Remember how memory is mapped when using gdb to provide a memory map (that is where the least significant bit of a word is, or use alternate formatting commands to provide all the bytes in the correct order). Name the resulting file Part3a.c. When you execute the program, the code should result in a shell being opened. Now comment out the line of code `mprotect(..., ...., PROT_EXEC|PROT_WRITE|PROT_READ);` and compile and execute the code again. What happens? Look up the `mprotect` command online, and explain the difference in execution between the two pieces of code. Here we are using `mprotect` to specifically turn off a security functionality of Linux, namely that the heap is declared non-executable

Submission Place part3a.c in the submitted directory. Also include two typescripts called Part3aDemo1.txt and Part3aDemo2.txt which shows you compiling the program, and running it in the two ways described above. Include a txt file called Part3aExplanation.txt which gives your answer to the above question.

9.2 Part IIIb

Use the file "BufferRedirectViaStack.c" and put your binary hex code in the array shellcode. This program is similar to the previous one, but instead of just redirecting execution to the array. It copies the array onto the stack, as would be done in buffer overflow attack. At this point you should modify the code so that the last remaining NULL in the hex-code can be removed. Note it is insufficient to actually just remove the NULL at the end of the string, as the unix command execve will no longer know where the end of your string is. Instead, you need to place a 0 at the end of the string *after the program starts running.* This is not difficult, of course, treating a string like an array of characters, we just place a '0x00' at the appropriate location via our code. Name the resulting file Part3b.c. When you execute the program, the shell will not execute. There are two barriers to execution. The first is that the stack is not currently executable and the second is that there is a canary stack protection. *Note that when you are trying to get this code to run it is imperative that you compile with stack protection off and stack execution turned on (see earlier in the assignment). Once you have it running, you can should try compiling with all different settings of these options as described below in the submission directions.*

Submission Place part3b.c in the submitted directory. Also include a typescript Part3bDemo.txt where you compile and execute the program in four ways. 1) With no compile time options, 2) with stack protection turned off, 3) with an executable stack, and 4) with both an executable stack and stack protection turned off (see the beginning of the assignment for guidance on how to turn on and off the various features). Based on your readings and research of these features on the internet, create a file called Part3bExplanation.txt that explains the different executions.

9.3 Part IV

Place you buffer exploit code that you developed and tested in part IIIb into the attack.c code provided. Use this to launch an attack against the program vuln. The attack code itself is a significant part of the whole attack execution. The attack code takes in two variables. The first variable is the size of the attack string (which needs to be at least as big as both i) your program, and ii) the buffer you're overflowing), but it is normally a bit bigger, say a few 10s or 100s of bytes.

The second variable is a proxy for the address that is our guess for the beginning of the buffer on the stack that we're overflowing. The literal definition of the second variable it that it is the amount that will be subtracted from the current stack pointer value when the attack program is run . The *attack.c* program adds this value to the end of your shell code, in the hopes of overwriting the return address on the stack. What you need to do is align these values so that an appropriate amount is subtracted off in order for this value to align exactly (or closely if you have created a NOP sled in your code (see the readings)) with the beginning of the buffer.

To make this process easier we've done two things. The attack program tells you exactly which address it is appending to your shell code, and the vuln program tells you exactly what the memory address is at the top of the array being attacked. So once you run it, you can use the δ between the two values, to adjust your 'guess' on the next iteration.

Note that in the real world, no program is going to tell you what the address for the buffer your overflowing is, and in practice you write a program to search for it. We have given you an executable of vuln, but have also provided the c source, as it may help you in debugging. If you compile it again, you must compile it with the no stack protection flags and the executable stack commands given earlier, or your new version of vuln will not be susceptible to attack.

Submission Place your modified attack.c in the submitted directory. Also include a typescript Part4Demo.txt where you compile and execute the program with parameters that result in the shell being executed (i.e. vuln is successfully attacked). Also run it with a buffer size that is too small to cause a buffer overrun, and one where the buffer is sufficiently big to cause a stack overflow, but where the attack is not successful. **NOTE:** Make sure that ALSR is turned off while trying to attack the vuln program, or you will have a nearly impossible time of it. You will know it is off if each time you run the attack program with the same buffer size and different deltas, the address in vuln of the buffer being attacked does not change.

10 Collaborators

Finally include a file called collaborators.txt, and include in it the names of everyone you have collaborated with, and in what capacity you collaborated. Note all the code you write, and the scripts you run must be done by you. However, you can, and are expected, to discuss the problems with your classmates. You just shouldn't take notes from those meetings, nor should you copy code, etc..., and of course, you should give credit where credit is due.