Exploits Lab
CSC 9010/5930 - Offensive Security
Grading: 10 points
Due Date: Jan 30th, 2019 at 11:59 PM

Description: In this lab you will learn about basic binary exploitation in C. Follow along with the instructions, write your answers to all questions in a separate `answers.txt` file, and submit on Blackboard when finished.

The Narnia Exercises: The exercises for today's lab are from a series called "narnia", hosted by Over The Wire. This site hosts a number of wargames and practice environments to teach a variety of different security and penetration testing concepts.

1. Begin the lab by going to `http://overthewire.org/wargames/narnia/`. Here you will find a summary of the exercise and instructions for logging into the Wargames server.

2. Begin by logging into the narnia0 username on the server narnia.labs.overthewire.org. You will need to specify the correct port number for ssh using the `-p 2226` flag.

3. As specified on the home page, the exercise source and executables can all be found in the `/narnia` directory. Note that there are not many places where you can write data to the file system, but `/tmp` is one. As user narniaX (for X = 0,. . . ,9), your goal is to exploit the binary named `narniaX` to allow you to read the password file for the next level. All of the password files are in the `/etc/narnia_pass/` directory.

Part A:

1. The first thing you will want to do is examine the source code for `narnia0` and run the executable a few times. The program accepts input read from standard input and stores it to a buffer, but it is configured to read in too many bytes! Let's exploit this.

2. Since buf is declared after val in the main method, it will rest above val on the stack. So, the 24 bytes read in by scanf will fill up the buffer and overflow to change val as well. The program is configured to launch a shell if val is overwritten with the proper value.

3. To exploit this, you will need an exploit string of 20 bytes to fill up buf, with four additional bytes representing the target value of "0xdeadbeef" (to trigger the if statement further down in the source). You could type this by hand, but it's faster to use python. Try the following command:
   `python -c 'print "A"*10 + "\x45\x45\x45\x45"'`
   You will see the character "A" printed ten times, concatenated with the character "E" four times. This shows Python's ability to print out ASCII characters or direct hex values (which will be interpreted as ASCII if they are in the correct range).

4. To pipe the value printed by python into the program, simply enter the python command followed by `| ./narnia0`. This will pipe the output of python into standard input for narnia0.

5. From here, you should be able to modify the above python code to produce an appropriate attack string. Remember that the bytes piped into the program will be written to memory in the order they are typed, but since the x86 architecture is little-endian, they will be interpreted as starting with the least-significant byte. For our target value, the least significant byte is `\xef`.

6. At this point, you should be getting the vulnerability to trigger, but nothing seems to happen. This is because the program is running the shell internally, where it tries to run any instruction piped to standard input. Since there is nothing after the attack string, the shell and the enclosing program are simply terminating.

7. To finish the attack, we need to pipe in a shell command that will print out the password stored in `/etc/narnia_pass/narnia1`. We can combine shell commands using a semicolon, so we will use the "echo" command to print a call to the "cat" command, which will print the contents of a file to standard output:
   `(python -c 'print <YOURATTACKSTRING>' ; echo 'cat /etc/narnia_pass/narnia1') | ./narnia0`
   This should print a string of characters password that will allow you to ssh in as narnia1.

8. QUESTION: write the password in your answer file.

Part B:

1. After you log out and log back in as narnia1, you can see a new source file and run a new binary in the `/narnia` directory. You'll immediately notice that this source tries to execute whatever is in the "EGG" environment variable. Let's put something good there!

2. We will be filling the variable with shellcode, which is a specially crafted set of machine instructions designed to launch a new bash shell (with new user privileges). Learning to write your own shellcode takes a lot of time and practice, but there are plenty of examples we can borrow for now. To export the correct value to EGG, enter:
   ```
   export EGG=‘python -c 'print
   "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f
   \x62\x69\x6e\x89\xe3\x89\xc1\x89
   \xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"'‘
   ```
   Note the use of the apostrophe (') and the tick (‘). The apostrophe specifies the command to be fed to the python interpreter, the tick tells bash to execute the command inside the ticks before proceeding. You can also cut and paste the shellcode from here:
   `http://shell-storm.org/shellcode/files/shellcode-811.php`

3. Now, when you run narnia1, a new shell prompt will open (thanks to our shellcode). Simply use "cat" to print the password file for narnia2!

4. QUESTION: write the password in your answer file.

Part C: Explore

1. See how many more programs you can exploit. Some tips for the third exercise:

(a) You will need to try several different lengths for your attack string. A string that is too long will cause a SEGFAULT, and a string that is too short will allow the program to exit normally. Use something like a binary search to figure out exactly how long your attack string needs to be.

(b) You will need to use a NOP sled for this attack.

(c) gdb is your friend! After loading the program into gdb, use: `run ‘python -c ’print "<YOURATTACKSTRING>"’‘`
to start the program with command line parameters. Remember, the debugger is useful for stepping through execution *as well as* inspecting the machine state after a crash (or SEGFAULT).

(d) Remember this program accepts input through command line args, not standard input. Use the tick notation in your terminal to craft your input strings as arguments to the binary executable.

(e) Once you have refined your attack string in gdb, you will have to execute the overflow outside of gdb.

2. **QUESTION:** How helpful did you find today's lab exercise? What helped your learning the most? What would you change?

Rubric:
(10 points) submit your answer sheet on Blackboard.

Deliverables: Submit the answer sheet on Blackboard.