Paxson
Spring 2011

CS 161
Computer Security

Project 1

Due: February 22, 11:59pm

Background

It is a time of rebellion. The evil empire of Caltopia oppresses its people with relentless surveillance, and the emperor has recently unveiled his latest grim weapon: a supremely powerful botnet, called *Calnet*, that aims to pervasively observe the citizenry and squash their cherished Internet freedoms.

Yet in the enlightened city of Birkland, a flicker of hope remains. The brilliant University of Caltopia alumnus Neo, famed for not only his hacking skills but also the excellent YouTube videos he produces illustrating his techniques, has infiltrated the empire's byzantine networks and hacked his way to the very heart of the Calnet source code repository. As the emperor's dark lieutenant, Prof. Evil of Junior University, attempts to hunt him down, Neo feverishly scours the Calnet source code hunting for weaknesses. He's in luck! He realizes that Prof. Evil enlisted ill-trained CS students from Junior University in writing Calnet, and unbeknownst to the empire, the code is assuredly not memory-safe.

Alas, just as Neo begins to code up some righteous exploits to pwn Calnet's components, a barista at the coffeeshop where Neo gets his free WiFi betrays him to Prof. Evil, who brutally cancels Neo's YouTube account and swoops in with a SWAT team to make an arrest. As the thugs smash through the coffeeshop's doors, Neo gets off one final tweet for help. Such are his hacking skillz that he crams a veritable boatload of key information into his final 140 characters, exhorting the University of Birkland's virtuous computer security students to carry forth the flame of knowledge, seize control of Calnet, and let freedom ring once more throughout Caltopia . . .

Getting Started

In this project you will write exploits for 4 vulnerable Calnet components. Each program forms one part of the nefarious botnet. All you have to go by are your wits, your grit, and Neo's legacy: guidelines on how to proceed, and, most precious, a virtual machine (VM) image that contains code samples from the main Calnet components.

VMware Setup

Neo placed the image at http://cs.berkeley.edu/~mavam/teaching/cs161-sp11/vm.zip. Download the image and extract it on your local machine. You need VMware to launch the VM. VMware Player is installed on the instructional machines and is also freely available for Windows and Linux. The Mac version of VMware (VMware Fusion) is available as a free 30-day trial. To use the image, start VMware, select *Open a Virtual Machine*, and browse to where you've stored the image. If it asks whether the VM was moved or copied, select *I copied it*.

You will run the vulnerable programs and their exploits in the VM. The image is a bare-bones Linux Ubuntu installation on a 32-bit Intel architecture. Valid logins are `root` and `calnet`, both having the password `cs161`. The relevant files are located in the home directory of the user `calnet`. We recommend to work as user `calnet` and use the `root` login only to modify the VM, e.g., to install additional packages. Further, a convenient way to use the VM is to launch it, find out its IP address via `ifconfig eth0`, and then work with it remotely via SSH.

The GNU debugger

The GNU debugger `gdb` will prove useful for this project, and worth some time spent becoming comfortable with it. A basic `gdb` workflow begins with loading the executable in the debugger:

```
gdb executable
```

You can then start running the program with:

```
$ run [arguments-to-the-executable]
```

(Note, here we have changed gdb's default prompt of `(gdb)` to `$`).

In order to stop the execution at a specific line, set a breakpoint before issuing the "`run`" command. When execution halts at that line, you can then execute step-wise (commands `next` and `step`) or continue (command `continue`) until the next breakpoint or the program terminates.

```
$ break line-number or function-name
$ run [arguments-to-the-executable]
$ step        # branch into function calls
$ next        # step over function calls
$ continue    # execute until next breakpoint or program termination
```

Once execution stops, you will find it useful to look at the stack backtrace and the layout of the current stack frame:

```
$ backtrace
$ info frame 0
$ info registers
```

You can navigate between stack frames using the `up` and `down` commands. To inspect memory at a particular location, you can use the `x/`*FMT* command

```
$ x/16 $esp
$ x/32i 0xdeadbeef
$ x/64s &buf
```

where the `FMT` suffix after the slash indicates the output format. Other helpful commands are `disassemble` and `info symbol`. You can get a short description of each command via

```
$ help command
```

In addition, Neo left a concise summary of all `gdb` commands at:

http://cs.berkeley.edu/~mavam/teaching/cs161-sp11/gdb-refcard.pdf

You may find it very helpful to dump the memory image ("*core*") of a program that crashes. The core captures the process state at the time of the crash, providing a snapshot of the virtual address space, stack frames, etc., at that time. You can activate core dumping with the shell command:

```
% ulimit -c unlimited
```

A crashing program then leaves a file `core` in the current directory, which you can then hand to the debugger together with the executable:

```
gdb executable core
$ bt      # same as backtrace
$ up      # move up the call stack
$ i f 1   # same as "info frame 1"
$ ...
```

Lastly, here is how you step into a second program `bar` that is launched by a first program `foo`:

```
gdb -e foo -s bar             # load executable foo and symbol table of bar
$ set follow-fork-mode child # enable debugging across programs
$ b bar:f                     # breakpoint at function f in program bar
$ r                           # run foo and break at f in bar
```

Problems

Problem 1 ***Buffer Overflow*** **(30 points)**

Neo's tweet assures you that given its hasty development by poorly educated programmers, Calnet's components contain a number of memory-safety vulnerabilities. In the VM that Neo provided, you will find the first code piece located in the directory `/home/calnet/q1`. The executable you will exploit is named `target1`. Neo also placed a copy of the corresponding source code in `target1.c`. You can use it to locate the vulnerability.[1] You also discover the in-progress exploit that Neo had started working on in the file `exploit1.c`. You are to continue his work and write an exploit for `target1` that spawns a root shell. To this end, you can use shellcode provided in `shellcode.h`. To compile your exploit, type `make`. After successfully diverting the control flow to the shellcode, you will see a root shell prompt:

```
calnet@cs161:~/q1$ ./exploit1
#
```

To get started, read "Smashing The Stack For Fun And Profit" by AlephOne [1]. Neo recommended that you try to absorb the high-level concepts of exploiting stack overflows rather than every single line of assembly. He also warned you that some of the example codes are outdated and may not work as-is.

Submission and Grading. For this problem you will submit `exploit1.c`. Our grading tool will log into a clean VM image as user `calnet` and put your submission into the directory `q1`. A script will then (1) compile your exploit by running `make`, (2) execute `exploit1` via `ssh calnet@VM:q1/exploit1`, and (3) check for the existence of the root shell prompt. We encourage you to test your exploit also via SSH to make sure that it will work with our grading tool.

You must also submit a file, `exploit1.txt`, that includes a brief description of the vulnerability, how it could be exploited, how you determined which address to jump to, and a sketch of your solution. This document should be no more than one page. We will use it to verify that your understanding of the problem matches your exploit code. Moreover, we will use it to award you partial credit in the event that your exploit does not work with our automated grading system.

[1] Do not recompile the target. It is has the *setuid* bit set and is owned by root, meaning, it will run with effective root privileges even when launched as a regular user. These properties are lost upon recompilation, which could cause subtle problems.

Problem 2 ***Arithmetic Overflow*** **(20 points)**

In another related component of Calnet, the inept Junior University programmers actually attempted a half-hearted fix to address the buffer overflow vulnerability in Problem 1. In this problem you must bypass these mediocre security measures and again inject code that spawns a root shell.

You can find the vulnerable binary in `/home/calnet/q2`, named `target2`, and the corresponding source code in `target2.c`. As in the previous problem, you discover an exploit stub that Neo started to hack up in `exploit2.c`. You must write an exploit for `target2` that spawns a root shell. You can again find a copy of the shellcode for this purpose in `shellcode.h`.

Submission and Grading. For this problem you will submit two files, `exploit2.c` and `exploit2.txt`, which have the same meaning as in Problem 1. The grading procedure is identical.

Problem 3 ***BSS Overflow*** **(20 points)**

Calnet is a pernicious and invasive piece of malcode. One of the many features that Prof. Evil designed into it extracts credit card numbers and other personally identifiable information (PII) from the machines of innocent citizens. The malware saves the gathered data in a file `/tmp/pii.txt` before sending it off to Prof. Evil's headquarters.

Neo's tweet includes an example of this text file:

```
42
John Doe - CC:5450207771681322
43
L. User - CC:378407724076386
44
Rick Astley - CC:4716817342323957
```

You can find the Calnet code that writes into this file in `/home/calnet/q3`, named `target3`. It takes input from *stdin* and writes it to the file `/tmp/pii.txt`. To produce entries as shown above, `target3` splits each line of input at the first whitespace and then writes to two separate lines in the file. For example, it splits the string

`"42 John Doe - CC:1234567890"`

into the substrings `"42"` and `"John Doe - CC:1234567890"`, each of which it then writes into a line of its own.

In this problem, your must write an exploit that adds a new user `rut` to the system.

You will craft an input string to change the location of the PII file from `/tmp/pii.txt` to `/etc/passwd`. To add the new user `rut` to the system, the input string has to begin with:

```
rut::0:0::/:/bin/sh #...
```

Submission and Grading. For this problem you will submit two files, `exploit3.c` and `exploit3.txt`, which have the same meaning as in Problem 1. The grading procedure is almost the same as in Problem 1, except that our grading tool does not check for the existence of a root shell prompt, but instead tries to log in as user `rut` after having executed your exploit.

Problem 4 *Format String Vulnerability* (30 points)

When Prof. Evil first developed Calnet, he had little faith in the loyalty of his subordinates, and accordingly built a *kill switch* into the botnet so he could disable it lest one of his minions attempted to seize it for their own use. Once Calnet went out of beta, however, he removed the kill-switch functionality. But Prof. Evil made this change in a hurry, and himself never had all that lucid of an understanding of basic software security. Is there some way that the rebels could perhaps re-enable the kill switch, and use it to disable the entire botnet? That had been Neo's unceasing dream … and you discover it had been tantalizingly close within his grasp before his sorry bust at the coffeeshop.

The vulnerable program, `target4`, takes multiple arguments. The first two, `delay` and `action`, specify the time until a specific action is executed. The program converts both arguments to the type `unsigned long`.[2] Neo's final tweet did not clarify the use of the final argument, which perhaps only exists for legacy reasons, but did convey that Neo believed it holds the key to deactivating Calnet.

In this problem you must find a combination of arguments that cause the function `disarm` to execute. To keep his dream alive, Neo provided a suggestion that you read "Exploiting Format String Vulnerabilities" by scut / team teso [2].

Submission and Grading For this problem you will submit two files, `exploit4.args` and `exploit4.txt`. The first file `exploit4.args` contains the first three arguments to `target4`. Our grading tool will log into a clean VM image and invoke `target4` with the arguments in `exploit4.args`. It deems the exploit successful if the supplied arguments cause the function `disarm` to execute.

[2]**Hint:** addresses are of type `unsigned long` as well.

The second file, `exploit4.txt`, includes the same type of description as in Problem 1.

Problem 5 ***Feedback (optional)*** **(0 points)**

If you wish, submit a text file, `feedback.txt`, with any feedback you may have about this project. What was the hardest part of this project in terms of understanding? In terms of effort? (We also, as always, welcome feedback about other aspects of the class.) Your comments will not in any way affect your grade.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996. `http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf`.

[2] scut / team teso. Exploiting Format Strings Vulnerabilities, September 2001. `http://badcoded.blogspot.com/2007/12/user-supplied-format-string.html`.