CS 161

Computer Security

Dawn Song

Fall 2012

Lab 1

Buffer Overflows v.01

Due Date: September 17, 2012 by 11:59pm

Introduction

In this lab, you will get a hands-on approach to circumventing user permissions and access control through three buffer overflow exploits. We have provided you and your partner with a virtual machine that contains 3 executables and source files which contain software vulnerabilities.

Note: we will not be giving any details of the different exploits you will be using to acquire a shell. This will be a real-world experience where you have to discover what will work! ;)

Overview and Getting Started

Please note: No collaboration beyond your lab partner is allowed! Max group size is 2. No exceptions unless there are an odd number of students.

To get started, read "Smashing The Stack For Fun And Profit" by AlephOne [1], "Basic Integer Overflows" by Blexim [2], and "The Frame Pointer Overwrite" by klog [3]. These documents are fairly detailed, and we therefore recommend that you try to absorb the high-level concepts of exploiting stack overflows. Also, some example code may not work as-is from these documents.

In this project you will write exploits for 3 different programs, each is owned by a different user on the virtual machine. Each exploit, when run in the virtual machine on target programs installed with setuid[1] will yield a shell as the owner of that program (calnet1, calnet2, calnet3). By acquiring a shell by exploiting each program, you will be able to view files owned by that user. Each exploit has a respective secret code that we will use for submission.

All of the targets are found in the `/calnetsrc` directory. After exploiting a target, in that shell, extract the secret code by typing `cat /calnetN/code` (N is the current target). If your exploit worked, you will gain a shell with the proper permissions and be able to output the secret code. SAVE THIS FOR YOUR SUBMISSION.

The sploits/ directory in the assignment tarball contains skeleton source for the exploits which you are to write, along with a Makefile for building them. We also included shellcode.h, which has Aleph One's shellcode.

[1] setuid allows a program to run with privilege as a specific user as opposed to the privilege of the user who launched it.

In summary, our goal is for every student to gain a hands-on experience with the effect of buffer overflows. Your goal is to:

1. Write three exploit programs (sploit 1, sploit2, sploit3).
2. Execute each exploit (it will feed in your input that results in a priveleged shell of user calnet1, calnet2, or calnet3).
3. Save the secret code for each user to prove you were able to gain access.

Additional libraries are not needed to solve this lab. All you need are your wits and the VM. Good luck! For any questions, please post on Piazza!

Beginning Notes

You may download the lab image at `http://tinyurl.com/ucbcs161lab1-2`. To simplify your task, everything you need is contained within this VM. You may use VMware Player (Windows), VMware Fusion (Mac), or other compatible software to run the vm. If your software asks whether the VM was moved or copied, select *I copied it*.

Important! In order for each group to have unique solutions, there is a file "setup" in user's home directory. Before attempting the exercises, run `./setup` and enter a berkeley email address of you or your partner. This sets up the secret code files whose contents you will discover.

Login Info: user: `user` , password: `user`.

The relevant files are located in the home directory of the user. Also, we highly recommend that, instead of coding in the VM itself, each student launches the VM and SSH into it. This will remove the pains of working within a small inconvenient window on the VM. (You can find out the IP address via `/sbin/ifconfig eth0`)

Initial Setup

After starting up the VM, you will notice there are several setup files. Follow these instructions to get ready to go! Remember: login is `user`, with password `user`.

Files included in this exercise

- `/sploits`, folder containing exploit stubs.
- `/targets`, folder containing target application sources and compiled executables.

```
user@box:~/$ tar xzvf lab1.tgz
user@box:~$ cd targets
....look around and understand the folder structure....
user@box:~targets$ cd sploits
....edit, test, exploit....
user@box:~/sploits$ make
user@box:~/sploits$ ./sploit1
#cat /calnet1/code
Write down the code that shows up here!!
```

Submission and Grading

Each partner will submit their own version of the lab. Each will submit two files for final grading: `submission.txt` and an archive of the exploits. You will upload these files to bSpace.

Creating the submission archive:

1. For every exploit, please write an additional file `explanationN.txt` which details your thought process in solving the problem, an explanation of your exploit strategy, and any other relevant information (Please keep this less than 500 Words!). This will be used for partial credit.

   Hint: Try to imagine you are describing how to reproduce the attack to someone with knowledge of the subject, but lacks complete expertise.

2. Use the submit script found on the course website to create a "submission file" which will contain your information and secret codes. (You can either copy and paste the script into the VM, or use scp submit.sh user@192.168.173.128:, where 192.168.173.128 is your VM's respective IP address).

   Example output after executing `./submit.sh`

   ```
   Enter your name:
   Dylan Jackson
   Enter your email address:
   dylanj@berkeley.edu
   Enter your partner's name:
   Kunal Agarwal
   Enter your partner's email address:
   kunala@berkeley.edu
   Enter secret codes below found in /calnet*/code.
   Enter code1:
   d.r.jackson@berkeley.edu:secretcodehere
   Enter code2:
   d.r.jackson@berkeley.edu:secretcodehere
   ```

```
Enter code3:
d.r.jacksonberkeley.edu:secretcodehere

Done writing to submission.txt
```

3. Move the submission file, the explanation files, and *EVERYTHING* required to build your exploits into a tarball: Makefile, source files, header files, etc. **Please name the tar file firstname_lastname.tgz**.

TEST YOUR SUBMISSION Always double check your submission to make sure that you get the points that you deserve. Try the tar file on a fresh copy of the VM! If there is any manual intervention by the graders, you will be deducted points.

Questions and Problems

As per usual, please post any questions to Piazza.

Acknowledgments

Kunal Agarwal and Dylan Jackson - Credits to Steve Hanna and Matthias Vallentin (GDB Tutorial).

References

[1] Aleph One. Smashing The Stack For Fun And Profit. `http://www.phrack.com/issues.html?issue=49&id=14`.

[2] Blexim. Basic Integer Overflows. `http://www.phrack.org/issues.html?issue=60&id=10`.

[3] klog. The Frame Pointer Overwrite. `http://www.phrack.com/issues.html?issue=55&id=8`.

Appendix

GNU Debugger

The GNU debugger `gdb` is a very powerful tool that is extremely useful all around computer science, and will be especially essential for this project. 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://vividmachines.com/gdbrefcard.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
```