CSE 484 / M584 Lab 1: Binary Exploitation

1 Due Dates

- **Part A Due (Exploits 1,2)**: Wednesday April 9th, 11:59pm
- **Part B Due (Exploits 3,4,6)**: Wednesday April 16th, 11:59pm
- **Extra Credit (Exploits 5,7,8)**: Wednesday April 16th, 11:59pm

Handin All parts on Gradescope, see Deliverables.

Groups

- **Exploits:** Individual or partners.
- **Writeups:** Individual.

Grading

Points

- 5 per exploit (Submit as partners)
- 5 per writeup (Submit individually)
- Extra Credit worth 1 per exploit, 1 per writeup.

Writeups MUST be done individually: do not collaborate on the writeups.
Exploits are graded as all-or-nothing: each submitted exploit gets 0 or 5 points.
Writeups are graded normally.

2 Formatting Notes for this Handout

- Programs, files, or tools are stylized like `this`.
- Commands for programs are stylized like `this`.
- Code, assembly, and strings are stylized like `this`.

3 Before you start

Make sure you can SSH into the CSE Linux server we use for this course: `umnak.cs.washington.edu`.
This uses the same credentials as `attu.cs.washington.edu`.

4 Overview

4.1 Goal

- The goal of this assignment is to gain hands-on experience with the effects of buffer overflow bugs and similar problems. We strongly suggest doing all work on `umnak.cs.washington.edu`.
- You are given the source code for seven exploitable programs (`targets/targetN`).
- Your goal is to write seven exploit programs (`exploit1`, ..., `exploit7`). Program `exploit[i]` will execute program `../targets/target[i]`, giving it an input you construct that results in shellcode running.
- Each exploit, when run including `checkcode.h` (instead of `shellcode.h`) should print a success message.
- When run including `shellcode.h` it should start a new (nested) shell.

4.2 Setup

First, you will need to make a fork of the lab1 gitlab:
https://gitlab.cs.washington.edu/dkohlbre/buffer-lab-25sp
Please make this fork *private* so that other students don't find it! Share this fork with your partner if you have one. Then clone your fork to wherever you are working (we encourage umnak.)

4.3 Code

4.3.1 The Targets `targets/`

- Read the source files carefully, along with any header files they include.
- Note that the extra bits included (strlcpy and tmalloc) are standard implementations: they don't have bugs we care about, but you will want to read them and understand them.
- Make sure to build the targets.
- Do not modify the targets, even for debugging purposes.
- You should examine the target binaries, and will find value in using `objdump` to get the assembly of the targets.

4.3.2 The Exploits `exploits/`

The `exploits/` directory contains skeletons for each of the exploits you will write, a `Makefile` for building them, and two options for shellcode: `shellcode.h` and `checkcode.h`. `checkcode.h` will try to run the check script, rather than start a shell. Build them with `make`, do not build manually.

Each skeleton is structured in the same way: it sets up an environment in which to call the right target program, then passes arguments to it and and starts it. This ensures that the target program executes in a very controlled and dependable way. The argument currently passed is: `''hi there''` which, it turns out, is not a very effective adversarial input! You will need to replace this argument with your adversarial input.

4.4 Extra Credit

Some targets are extra credit.

Target 5 is an "unlink" exploit, which will require doing the optional assigned readings. Section may also be helpful.

Target 7 will require you to carefully step through program execution, and learn something new (not taught in class!) about how programs are loaded and linked.

Target 8 requires a different exploit technique! For 8, you can see that the source code is exactly the same as target0, except this time, the stack is *not* executable. You might want to try a return2libc attack.

4.5 Optional Exercises

4.5.1 Basic Buffer exercises (`buffer-exercises/`)

These are a few small exploitable programs that use `strcpy` unsafely, and your goal is only to change the value in a local stack variable determining if the user is an admin.
You may find using `python` a useful way to convert between decimal and hexidecimal, and you may also find using the `ascii` Linux command useful as well.
Toy1 and toy2 are great warmup problems, feel free to edit them or make alternatives when trying to work through the graded exploits.

Toy1 Your goal is to give an input to the program that will cause it to say you are the admin. Read the check for admin carefully to determine what input to give the program.

Toy2 Your goal is to give an input to the program that will cause it to say you are the admin. You will need to do a bit more work to ensure you pass an extra safety check on this one.

ToyX Your goal is to give an input to the program that will cause it to say you are the admin. The check from toy2 looks the same, but this one is *much* trickier. We don't have a reference solution; it may not be possible! Understanding *why* it is hard may be useful, and understanding why it may be impossible may help you think about solving exploit7!

4.5.2 `printf` exercises (`printf-exercises/`)

These are a few small exploitable programs that use `printf` in bad ways.
These are a great way to get some experience working with `printf` exploits before you try exploit6.

Toy1 Your goal is to give an input to the program that will cause it to print out the secret string.

Toy2 Your goal is to give an input to the program that will change the value of a variable such that the toy prints out a success message.

Toy3 Your goal is the same as before, but now it is setup to run in the same way as exploit6 (with a harness that starts the target and gives it an input.)

5 Deliverables

5.1 Lab 1a

There are two gradescope assignments: One for your code (this should be turned in as partners if you are partners), one for your writeups (this one is individual.)

- Your `exploit1.c` and `exploit2.c` files.
- Individually done writeups for your exploit approach for exploit1 and 2.

5.2 Lab 1b

- Your `exploit3.c`, `exploit4.c` and `exploit6.c` files.
- Individually done writeups for your exploit approach for exploits 3,4 and 6.

6 Exploit Writeups

You should produce, entirely on your own, a brief writeup for each of the exploits you solved. The goal here is that if teammate B discovered the key insight for exploit3, teammate A needs to really understand that insight to do the writeup. **Your writeups should be in your own words, and written solely by you. If your whole team submits copies of the same writeups, you won't get full credit for this.**

A writeup should explain what your exploit does, and what goals it accomplishes along the way. This writeup should be relative to the complexity of the exploit, and should be at most 2 paragraphs long for the most complex ones. **Maximum length of 500 words per-exploit.**
If you aren't sure how to start, consider how you would walk someone who hasn't done the lab through the problem. Or how you'd respond to a TA if they asked you "how does your exploit work? What are the steps it takes and why do they work?"
An exploit 0 writeup is quite simple, and might say: "Unfortunately, strcpy does not have a bounds check, so the copy is able to go past the end of the stack buffer in foo if our input is larger than the buffer. We use this to write arbitrary values to anything above the buffer on the stack. Specifically, we write over the return pointer on the stack for foo (e.g. the one that points back into main) with the address of the stack buffer buf. This buffer was first filled with our shellcode, so when foo returns it does not return to main, and instead executes our shellcode."

Important: For exploit1, you should not simply copy/lightly reword the exploit0 example above. That is one of many, many reasonable ways to explain exploit0 or 1. Write a new one, in your own words, to demonstrate that you really understand what's going on! We do understand that you and your partner will have similar writeups for many exploits, but make sure you do them independently. We do compare them using plagarism tools.

7 Miscellaneous

7.1 gdb

You will want to use gdb. We recommend learning an extension to gdb.

gef (https://hugsy.github.io/gef) is an exploitation-focused set of gdb extensions. To use it, download the gef.py file to wherever you are using gdb and then add loading gef.py to your .gdbinit. In gef, if you lost the original nice display, you can use the command context to tell it to reprint it. We highly recommend learning gef.
There's lots of online documentation for gdb. Here's one you might start with: GDB Notes (formerly hosted at CMU)

- Be familiar with commands like: disassemble, stepi, break, examine.
- Nearly all gdb commands have shorthands, e.g. si is stepi, and x is examine.
- The layout command is helpful for seeing multiple things, e.g. layout src will show the source code of the target and the location of breakpoints.
- The info register command is helpful in printing out the contents of registers such as ebp and esp.
- The info frame command also tells you useful information, such as where the return eip is saved.
- You will find the x (eXamine) command useful to examine memory (and the different ways you can print the contents such as /a /i after x).
- Since you are running exploit but want to debug target you will need to catch the switch from exploit to target. This can be done with catch exec which will break once exec is called. Now you can set your breakpoints for the target. Note that if you try to set breakpoints in target before you have hit exec, it won't work as expected.
- The catch exec component can be automated with a script! Spending time reading up on gdb commands, scripts, and such can be very helpful for debugging.

7.2 Hints

- Read the Lab1 FAQ!
- Remember 351's bomblab? This is similar, but there are many points of difference. Notably this is 32-bit, not 64-bit.
- Read Aleph One's "Smashing the Stack for Fun and Profit" carefully! Another classic useful reading is Chien and Szor's "Blended Attacks" paper. These readings will help you have a good understanding of what happens to the stack, program counter, and relevant registers before and after a function call, but you may wish to experiment as well. It will be helpful to have a solid understanding of the basic buffer overflow exploits before reading the more advanced exploits.
- Read "once upon a free()" for exploit5.
- Read scut's "format strings" paper.
- You may also wish to read http://seclists.org/bugtraq/2000/Sep/214.
- gdb. Really. Before you ask a TA, try walking through before and after your exploit triggers any state corruption using gdb.

- `objdump` is a great tool as well, it will let you print out the assembly of a program for further reference.
- Make sure that your exploits work on the server if you did some work locally.
- Start early!!! Theoretical knowledge of exploits does not readily translate into the ability to write working exploits. Target1 is relatively simple and the other problems are quite a bit more complicated.

7.3 Warnings

Aleph One gives code that calculates addresses on the target's stack based on addresses on the exploit's stack. Addresses on the exploit's stack can change based on how the exploit is executed (working directory, arguments, environment, etc.); in our testing, we do not guarantee to execute your exploits as bash does. You must therefore hard-code target stack locations in your exploits. You should not use a function such as `get_sp()` in the exploits you hand in.

7.4 Credits

This project was originally designed for Dan Boneh and John Mitchell's CS155 course at Stanford, and was then also extended by Hovav Shacham at UCSD. Thanks Dan, John, and Hovav! Previous UW security instructors also contributed significantly to the UW version of this lab.