CSE 484 / M 584: Lab 1

Buffer Overflows

Part A Due (Sploits 1-4): Monday, October 14, 11:59pm
Part B Due (Sploits 5-7): Wednesday, October 23, 11:59pm
Turn in: All parts on Gradescope, see Deliverables
Individual or group: Individual or partners (pairs of 2)
Points:

10 per sploit
5 per writeup (*individually* done)
EC worth 4 for the sploit, 2 for the writeup.

Before you start:

- Make sure you can SSH into the CSE Linux servers: `umnak.cs.washington.edu`
  - This uses the same credentials as `attu.cs.washington.edu`
- If you are unfamiliar or rusty with Linux, SSH/SCP, or git, review these resources:
  - Linux tips from CSE 351
  - Our SSH and SCP and Git guide

Overview

Goals

- 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**. Other setups may work but are untested and won't be officially supported/debugged.
- You are given the source code for seven exploitable programs (`targets/targetN`)
- Your goal is to write seven exploit programs (`sploit1`, ..., `sploit7`). Program `sploit[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` (the default setup), should print a success message. When run including `shellcode.h` instead, it should start a new (nested) shell. (Note: That nested shell will still have your own user privileges.)
- Sploits 1-7 are required. Sploit 8 is extra credit. Sploit 0 is similar to Sploit 1, and will be demonstrated/solved in Section.

Contents

First, you will need to make a fork of the lab1 gitlab:
https://gitlab.cs.washington.edu/dkohlbre/buffer-lab-24au

- 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 (probably `umnak`)
- Stuck on these instructions? Return to the top and see the resources under "Before you start"!

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 what they do.
- 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.

The Exploits (`sploits/`)

The `sploits/` directory contains skeletons for each of the sploits 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.

Printf exercises (`printf-exercises/`)

These are a few small exploitable programs that use `printf` in bad ways.

- 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.

These are **optional**, but they are a great way to get some experience working with simpler printf exploits before you try sploit6.

Extra Credit

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. Here's a good tutorial for it: RET2LIBC (starting from page 52). Since it's extra credit, we won't give further hints about it.

Deliverables (See Gradescope)

Lab 1a:

- Your `sploit1.c`, `sploit2.c`, `sploit3.c`, `sploit4.c` files.
- An *individual* writeup explaining your exploit strategies for each of sploits 1-4

Lab 1b:

- Your `sploit5.c`, `sploit6.c`, and `sploit7.c` files.
- An *individual* writeup explaining your exploit strategies for each of sploits 5-7

Writeups

You should produce a brief writeup for each of the sploits you solved. Writeups are *individual*, even if you worked in a pair. The goal here is that if teammate B discovered the key insight for a sploit, teammate A needs to really understand that insight to do their own writeup. **Your writeups should be in your own words, and written solely by you. If both partners submit copies of the same writeup, you won't get full credit for this. Do not use ChatGPT or similar.**

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 350 words per exploit.** If you aren't sure, consider what you'd tell a TA if they asked you "how did you exploit this?"

A sploit 0 writeup is quite simple, and might say:
"We overflow the stack buffer in foo, allowing us to write arbitrary values to anything above the buffer on the stack. We then specifically write over the return pointer on the stack for foo (part of main's frame) 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 sploit1, you should **not** simply copy/lightly reword the sploit0 example above. That is one of many, many reasonable ways to explain sploit0 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 describe the same strategies, but make sure you do them independently and in your own words.

Miscellaneous

gdb

You will want to use `gdb`. We recommend using extensions to gdb if you are comfortable learning them:

- `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 `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)

`gdb` is your best friend in this assignment, particularly to understand what's going on. Specifically, note the `disassemble` and `stepi` commands.

- 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 may 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 `sploit` but want to debug `target` you will need to catch the switch from `sploit` 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.

Hints and Notes

- 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! We also recommend reading 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 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 using gdb to walk through your exploit's execution and read relevant areas of memory before and after your exploit triggers any state corruption. Did the memory region you intended get corrupted in the way you intended?
- `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 (i.e., umnak) 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. In general, Target N+1 is not necessarily harder than Target N.

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.

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 David Kohlbrenner and Yoshi Kohno also contributed significantly to the UW version of this lab.