

# Buffer Overflows

**Part A Due (Sloits 1-3):** Friday, October 13, 11:59pm

**Part B Due (Sloits 4-7):** Wednesday, October 25, 11:59pm

**Turn in:** Group part on Canvas, Individual part on Gradescope, see Deliverables

**Individual or group:** Groups of up to 3

**Points:**

10 per exploit (Extra credit is worth 5)

5 per writeup

## Before you start:

- Read the CSE484 SSH+SCP Guide!
- Join a Canvas Group under the “**Lab 1 Groups**” group set. In Canvas, go to the “People” tab and search for “Lab 1”, and make sure that you and your teammates join the same group.
- **Individually** fill out the Google form with your ssh public key:
  - [https://docs.google.com/forms/d/1KJfkSBupZVmu9Po0Fo4\\_16hbZTsfwLWoVWf23dD-0c](https://docs.google.com/forms/d/1KJfkSBupZVmu9Po0Fo4_16hbZTsfwLWoVWf23dD-0c) [Make sure you are signed into your @uw.edu Google account]

After we have created your account (should happen automatically within 5-10 minutes), you should ssh into the lab server:

```
ssh -i <path-to-private-key> <username>@cse484.cs.washington.edu
```

Your username is **cse484-au23-lab1-n** where n is your Canvas lab 1 group number.

## Overview

### Goal:

- The goal of this assignment is to gain hands-on experience with the effects of buffer overflow bugs and similar problems. All of the work must be done on the machine **cse484.cs.washington.edu**
- You are given the source code for seven exploitable programs (`~/sources/targetN`), whose binaries are stored in the `/lab1/bin` directory (`/lab1/bin/targetN`). Each target program [i] is installed as `setuid hax0red[i]`.

- Your goal is to write seven exploit programs (sploit1, ..., exploit7). Program `sploit[i]` will execute program `/lab1/bin/target[i]`, giving it an input you construct that should result in a shell run with the same permissions as user `hax0red[i]`.
- Each exploit, when run on the `cse484` machine, should yield a `hax0red[i]` shell (`/bin/sh`). To confirm this is working, run the command `whoami` in the shell, and you should see the `hax0red[i]` user.
- Sploits 1-7 are required. Sploit 8 is extra credit.

## Contents:

By default, everything that matters is committed to a git repository in your homedir. We encourage you to use this while developing your exploits to keep track of your work.

## The Targets

- The targets are stored in `/lab1/bin/` and their corresponding sources in your home directory under `~/sources/`. You should carefully study the source code of each target.
- Your exploits assume that the compiled target programs are installed in `/lab1/bin/`. You cannot modify the targets.
- You can read the target binaries, and may find some value in using `objdump` to get the assembly of the targets.
- Each `target[i]` is `setuid hax0red[i]`, which means that they run as `hax0red[i]` regardless of who runs it. The one exception is when they're run under a debugger. Allowing users to debug a `setuid` executable would present interesting security problems, so `setuid` programs temporarily lose their `setuid`-ness under a debugger. This means that you can only get a `hax0red[i]` shell when your sploits are run outside of `gdb`. However, if you get a user shell inside `gdb`, you should get a `hax0red[i]` shell outside of `gdb`.

## The Exploits

The `~/sploits/` directory will contain the source for the exploits which you write, along with a `Makefile` for building them. Also included is `shellcode.h`, which gives Aleph One's shellcode. Build them with `make`, do not build manually.

## Extra Credit

Target 8 is extra credit! 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).

# Deliverables

## Lab 1a:

- A file `<netid member 1>_<netid member 2>_<netid member 3>.txt` With the output of the `handin.sh` script, unmodified.
- A writeup (PDF) explaining your exploit strategies for spoils 1-3

## Lab 1b:

- A file `<netid member 1>_<netid member 2>_<netid member 3>.txt` With the output of the `handin.sh` script, unmodified.
- A writeup (PDF) explaining your exploit strategies for spoils 4-7

## Using handin.sh:

We provide a `handin.sh` script that will make a copy of your current sploit code, and print out a list of hashes for those spoils. `handin.sh` must be run on `cse484`. You will hand in those hashes, and since we have access to your remote home directory, you won't need to submit any code. However, to let us know when you're done, please submit a text file named

```
<netid member 1>_<netid member 2>_<netid member 3>.txt
```

With the output of the `handin.sh` script, which is automatically put in a log file as indicated by the script. It should have a bunch of lines like this:

```
/homes/students/cse484-au23-lab1/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit0.c:11f5fbce21e8b67baf9abfdabf0a726e16cbfdef424d640946b7dcc1fff45a82
```

- Please just turn in that file, don't edit it/trim it/copy-paste the text out.
- Get the file from the server using `scp`, see the SSH+SCP guide.
- Test your code right before you hand in!
- Turn in your **group** text file via the Canvas assignment. (If you submit multiple times, canvas will add a suffix to your filename, this is fine.)
- Turn in your **individual** writeup to the Lab1a or Lab1b writeup assignment on Gradescope.

## Writeups

You should produce a brief writeup for each of the spoils you solved. Writeups are *individual*. The goal here is that if teammate B discovered the key insight for exploit3, teammate C 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. If you aren't sure, consider what you'd tell a TA if they asked you "how did you exploit this?"

A `spl0it0` 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 `spl0it1`, you should **not** simply copy/lightly reword the `spl0it0` example above. That is one of many, many reasonable ways to explain `spl0it0` or 1. Write a new one, in your own words, to demonstrate that you really understand what's going on!

## Miscellaneous

### `gdb`

You will want to use `gdb`. We have several `gdb` options available:

- `gdb`, it's `gdb`.
- `cgdb`, which gives you a view of the source code you're debugging
- `gef` (<https://hugsy.github.io/gef/>) for `gdb` is installed. To use it add it to your `.gdbinit` with 

```
echo "source /usr/share/gef.py" >> ~/.gdbinit
```

 If you use `gef`, `telescope` is a really useful command to check the information stored on stack. In `gef`, if you lost the original nice display, you can use `context` to tell it to reprint it.)

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 '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' command useful to examine memory (and the different ways you can print the contents such as `/a /i` after `x`).
- A useful way to run `gdb` is to use the `-e` and `-s` command line flags; for example, the command `gdb -e spl0it3 -s /lab1/bin/target3 -d ~/sources` tells `gdb` to execute `spl0it3`, use the symbol file in `target3`, and the `-d` shows you the source code of the target as you step through it. These flags let you trace the execution of the `target3` after the `spl0it` has forked off the `execve` process. When running `gdb` using these command line flags, be

sure to first issue 'catch exec' then 'run' the program before you set any breakpoints; the command 'run' naturally breaks the execution at the first execve call before the target is actually exec-ed, so you can set your breakpoints when gdb catches the execve. Note that if you try to set breakpoints before entering the command 'run', you'll get a segmentation fault.

## Hints

- 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 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 within the remote environment we provided.
- **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.
- Find more FAQs answered in the FAQ doc linked on the assignment page.

## 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.