7

Lab 5: Stack Smashing, Part 1

In this assignment, you will construct and carry out a stack-based buffer overflow attack. A stack overflow attack targets the integrity of a program's control flow, which is why it belongs to a class of attacks called "control flow integrity attacks." The ultimate purpose of the attack in this lab is to force the program to divulge secret information without the use of the correct password. Each part of the assignment guides you through systematically building up a buffer overflow exploit that bypasses the program's authentication mechanism. In this part of the lab, we use GDB and some test inputs to systematically probe the program's vulnerabilities. In Lab 7, we will write code that extracts the secret value.

For each question, be sure to follow the instructions carefully, supplying all of the parts mentioned. Be sure to supply a Makefile that produces whatever artifacts you submit. Please make sure that your Makefile includes updated `all` *and* `clean` *targets.*

7.1 *Learning Goals*

In this lab, you will learn:

- How to debug assembly code using GDB.
- How to analyze a binary for stack overflow vulnerabilities.
- How to deploy stack smashing exploit code.

7.2 *Required Reading*

- *Assembly-level debugging with GDB*
- *Finding a return address on the stack using GDB* (video)
- *Creating a shellcode file*

7.3 *Requirements*

Language. In order to carry out the attack you will primarily write assembly code. You may also need to write small utilities in C in order to prepare your attack. Hand in all of the utility programs you write along the way.

Common environment. Your code must be developed for and work on the Raspberry Pi machines we use for class.

Special note about SSH. If you plan to work on your assignment by `ssh`'ing to your RPi, please be aware that SSH changes the environment[1] of your user's shell. This means that you will very likely need to alter the offsets in your attack before you submit your assignment. All attack code must be checked against the console environment we set up in the first lab. If you do not understand what I mean, this would be a good question to ask me!

[1] A *shell environment* is the set of local variables and other shell settings, including `tty` settings, for your user. You can view the contents of your environment by typing `env` at a shell prompt. When a program is started, the entire shell environment is copied into the memory for the new process. Because the bottom of the call stack is placed *after* the environment, the size of the environment changes the starting offset of the shell.

Stack Overflow and the honor code. You are permitted to refer to Stack Overflow for help, but you must not under any circumstances copy the code you see there. If you find a helpful Stack Overflow post, you *must attribute the source of your inspiration in a comment at the appropriate location of your code*. You must also provide the URL of the post. Unattributed code will be considered an honor code violation.

Instructions for compiling and running. You must supply a file called `BUILDING.md` with your submission explaining how to:

- compile your program using your `Makefile`, and
- how to run your programs on the command line.

Reflection questions. This assignment asks you to answer a few questions. You must supply the answers to these questions in a `PROBLEMS.md` file.

Starter code. For this assignment, your repository includes the program you need to exploit, some sample attack code, and a `Makefile`.

7.4 *Application code*

You are supplied with a program in source code form, `prog.c`, however significant parts of the rest of the program are obscured: you are given a compiled binary and a header file only. Nevertheless, the function of the program should be clear.

You should compile the program with the supplied `Makefile` and try running it. For demonstration purposes, you should use the following login and password:

username: `W1234567`
password: `demodemo`

7.5 *Environment set-up*

Although variants of this attack are still possible on modern computers and operating systems, this particular attack is no longer feasible because of three security countermeasures: stack smashing protection (aka "stack canaries"), the non-executable stack, and address space layout randomization (ASLR). We need to disable all of these features to perform our attack.

This lab must be performed using the class Raspberry Pi computer. Your personal computer has both important architectural differences from the class hardware and likely incorporates additional countermeasures against control flow integrity attacks.

Figure 7.1: The idiom *canary in a coal mine* refers to the practice of using canaries to detect hazardous gasses like carbon monoxide. Due to their small size and faster metabolism, canaries are more sensitive to many toxins than humans. If a canary became ill or died, it signaled that miners should immediately evacuate.

7.5.1 *Disabling SSP and NX*

Any code you compile using `gcc` must disable *stack smashing protection* (SSP) and the non-executable stack (NX). It is also much easier to read generated assembly when *call frame information* (CFI) and *exception handling* directives are disabled. The supplied `Makefile` already has these flags, but here they are for posterity:

```
-fno-dwarf2-cfi-asm             # don't emit CFI
-fno-asynchronous-unwind-tables # really don't emit CFI
-fno-exceptions                 # disable exceptions
-fno-delete-null-pointer-checks # don't optimize nulls!
-z execstack                    # disable NX
-fno-stack-protector            # disable SSP
```

SSP works by inserting a guard value, known as a *canary*, between

the return address and the rest of the stack frame. When the function epilogue is run, this canary value is compared against the same canary stored in the program's read-only `DATA` segment. If the values are different, the return address has been tampered with, and the program is terminated by the C runtime.

NX ("no execute") is a hardware feature now present on all modern computers. Every virtually-addressed[2] memory page has an entry in a data structure called a *page table*. A page table maps the virtual address of a page to a physical address. They enable the operating system to translate between virtual memory requests made by a program and the physical management of memory performed by a hardware memory management unit (MMU). Page tables are maintained by the operating system. Other page-related information is stored in a page table, including the *NX bit*, which stands for "no execute." When the NX bit is set (`== 1`), the computer will refuse to execute any instructions found in that page. Modern compilers set the NX bit for `DATA`, stack, and heap segments, because valid code should reside only in the `TEXT` segment.

[2] Recall that *virtual memory* is an abstraction provided by the operating system to provide the illusion to programmers that a program has complete and exclusive access to all of the computer's memory. In reality, memory is shared among many programs. Virtual memory dramatically simplifies the programming of a computer. Without it, programmers would have to anticipate when memory might be shared, for any possible set of programs that *might* run on that computer. A difficult task to say the least!

The above set of `gcc` flags also disables a few things that will complicate stack frames for your program, like exception handler support.

7.5.2 *Disabling ASLR*

Your operating system also includes a feature called *address space layout randomization* (ASLR). ASLR is a security feature that changes the layout of your program from run to run. In particular, it randomly alters the starting offsets of the call stack, heap, and library functions. This has the effect of making it difficult to determine where things like return addresses are unless an attacker can run a program many times.

On your Raspberry Pi, run:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Your machine will prompt you to enter the password for your user.

You can verify that ASLR is off by running:

```
$ cat /proc/sys/kernel/randomize_va_space
```

`0` means that ASLR is disabled. `1` or `2` means that ASLR is enabled.

The setting you changed above does not persist after reboots. To disable ASLR permanently, run

```
$ sudo emacs /etc/sysctl.d/01-disable-aslr.conf
```

and when in `emacs`, add the following line:

```
kernel.randomize_va_space = 0
```

To test that you configured ASLR correctly, reboot your Raspberry Pi:

```
$ sudo shutdown -r now
```

and when it comes back up, in your terminal, run the same command we used to check ASLR above:

```
$ cat /proc/sys/kernel/randomize_va_space
```

where `0` means that ASLR is off, and `1` or `2` means that ASLR is on.

7.5.3 *Shell Environment*

Lastly, your shell's environment may make finding your addresses difficult, because if the environment changes, the program will be in a different location in memory. Both `ssh` and `gdb` load variables into the environment, so you may discover that when you get your exploit running in `gdb`, running the program outside of `gdb` (and/or `ssh`) may not work. When I developed my exploit, I found that the address of the exploitable buffer in `gdb` was 64 bytes lower than the address I needed without `gdb`. There are a couple ways to address this:

- Adjust your attack's new return address for use outside `gdb` so that you jump to the right place, or
- Add a *NOP sled*[3] to the beginning of your buffer and then adjust the return address so that both attacks will jump inside the sled.

How did I know how much `gdb` altered the offset? I temporarily inserted the following handy function

```
void print_stack_pointer() {
  void* p = NULL;
  printf("%p", (void*)&p);
}
```

into `prog.c`. Generally speaking, your exploit should work on an unmodified `prog.c`, but you may change it temporarily to figure out how to exploit it. Remember to ensure that your exploit works on an unmodified `prog.c` before you hand it in!

[3] A *NOP sled* is a sequence of valid instructions that allow an attacker leeway when trying to jump to the starting point of attack code. Typically, a NOP sled is made up of long chains of repeated instructions that cause no meaningful effect on the state of a computer, like `nop` or `mov r0, r0`. For example, supposing that `mov r3, sp` is the start of attack code, a NOP sled might look like:

```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
mov r3, sp
```

7.6 *Step 1: Find the vulnerability*

The supplied application contains two weaknesses, which together form a vulnerability. The first weakness is that the only thing guarding the program's sensitive data, obtained by calling the `decrypt` function, is a `student_id`, which is public knowledge.

Reflection Q1. Identify the second weakness and explain how the two weaknesses combine to constitute a vulnerability. Be sure to explain, in

general terms, how an attacker might exploit this vulnerability. Record your answer in `PROBLEMS.md`.

7.7 *Step 2: Jump to a different function*

After identifying the vulnerability, use `gdb` or simulate the program in order to find your point of attack. You will need to craft an input that overwrites a return address left on the stack. The function you should call is called `test2`.

Supply your input (which is likely to contain binary characters) in two forms:

1. as a string of escaped hexadecimal literals in a file called `input1.hex`, and
2. in binary form in a file called `input1`.

I should be able to run your exploit like this:

```
$ ./prog < input1
```

Reflection Q2. Crafting an input requires that you answer the following questions. If you're having trouble with the above, note that the answers to these questions are a recipe for crafting a buffer overflow exploit. It will probably help to answer them first!

1. What is the location of the return address stored in the stack frame for the function you plan to exploit?
2. What is the location of the buffer located that you plan to exploit?
3. How many bytes do you need to write in order to overwrite the return address?
4. What is the address you plan to put in the overwritten return address slot?
5. What order should your overwritten return address be written?
6. What bytes should you write into the buffer?
7. Since you cannot type in certain bytes, how will you write those bytes to an attack input file?
8. How does one feed an attack input file into a program?

Record answers to all of the above questions in `PROBLEMS.md`.

Reflection Q3. How does your attack work? Answer in detail in `PROBLEMS.md`.

7.8 *Step 3: Filling a buffer with shellcode and executing it*

Shellcode is attack code that launches a shell.

Your second attack should first fill a buffer with shellcode and then, after overwriting a function return address, transfer control to the shellcode in the buffer. To make this step easier, you are supplied with sample shellcode in assembly form.

You are given two sample shellcode files, `shellcode.s` and `shellcode-test.s`. `shellcode.s` has been written in such a way as to allow it to pass through C string-handling functions unmolested. Recall that C string functions are sensitive to `NULL` characters, because in C, `NULL` signifies the end of a string. To ensure that a shellcode attack is successful, no assembly mnemonic may generate an opcode containing `NULL` bytes. In other words, the byte `0x00` must never occur in the code. In *Part 2* of this lab, you will learn how to write shellcode and to remove the `NULL` bytes yourself.

Unfortunately, one trick `shellcode.s` utilizes is to modify itself. Were we to compile `shellcode.s` and try running it, the self-modification would segfault.[4] Therefore, you are also supplied with another, slightly modified version, called `shellcode-test.s`. This version cannot be used in a buffer overflow attack, because the self-modification is necessary to make the attack work. However, you can run it independently to ensure that you've set up your environment properly.

[4] Recall that program code is stored in the `TEXT` segment of memory. `TEXT` pages are marked as read-only, so self-modifying code will cause the program to abort. This is an important security feature!

Finally, it is sometimes difficult to tell when you've successfully started a shell. To make this crystal clear, this lab comes with a shell wrapper program called `qh` that prints `"Starting sh!"` when it starts up.

Do the following steps:

1. Compile `qh`

   ```
   $ make qh
   ```

   and install it in `/bin`.

   ```
   $ sudo mv qh /bin/qh
   ```

2. Make sure that `qh` works when called directly.

   ```
   $ qh
   Starting sh!
   $ exit
   exit
   ```

   Typing `exit` returns to your original shell.

3. Compile the shellcode:

```
$ make shellcode-test
```

   If you've done everything correctly, running `shellcode-test` should start `qh`.

```
$ ./shellcode-test
Starting sh!
$ exit
exit
```

Now that you know `shellcode-test.s` works, crafting an exploit using `shellcode.s` should also work.

1. Compile `shellcode.o`.

```
$ make shellcode.o
```

2. Extract all the machine code from `shellcode.o` associated with the `main` and `shell` labels.

3. Now, craft an input that exploits the program in the following way:

   (a) When the program runs, input is fed into the vulnerable buffer.

   (b) That input is made of extracted object code, padded with `nop` instructions where necessary.

   (c) That input is crafted so as to ensure that it overwrites the return address of the function containing the vulnerable buffer.

   (d) The new return address that you create points into the buffer you just overflowed, so that when the function returns, it jumps into your attack shellcode.

   Once you have crafted an exploit, you should be able to run it like this:

```
$ ./prog < input2
```

   Be sure to supply `input2` in two forms:

1. as a string of escaped hexadecimal literals in a file called `input2.hex`, and

2. in binary form in a file called `input2`.

Reflection Q4. How does your attack work? Answer in detail in `PROBLEMS.md`.

7.9 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab05-07_stack_smashing-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

7.10 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?

2. Did this assignment help you to understand buffer overflow attacks?

3. Is there is one skill/technique that you struggled to develop during this lab?

4. Your name, for the bonus point (if you want it).

7.11 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass the auxiliary sensor" you should have
    written "bypass the primary sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too many to count" as you
     state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.