Buffer Overflows

This project will give you hands-on experience with buffer overflow vulnerabilities, allowing you to attack a simple web server called zookws. The zookws web server is running a simple python web application, zoobar, in which users transfer "zoobars" (credits) between each other. You will find buffer overflows in the zookws web server code, write exploits for those buffer overflows, and look for other potential problems in the web server implementation.

Project Setup

- Click on the provided GitHub Classroom assignment link, login via GitHub if necessary, and click "Accept assignment".
- Wait for the repository to be created, then create a fork.
- Login to the VM.
- `cd` to your home directory and run `git clone <fork_url> lab/` to clone your fork.
- Run `cd lab/` to enter the project directory.

Refer to Project 0's writeup for elaboration on any of these steps.

> **Caution!**
>
> Before starting, remember the warning from Project 0:
>
> It is important that you **do not** modify the original repository created by GitHub Classroom (the one with `harvard-cs263` in the URL). Only ever modify the fork. This applies to **all** projects in this course.

> **Caution!**
>
> As mentioned in Project 0's writeup, **do not** use `apt-get` and related commands yet. Installing, removing, or upgrading packages (esp. `lib*`) has the potential to change various things in your address space, so your code will probably fail tests on our grading machines.

Specification

Caution!

For all projects, trying to modify or otherwise game the test cases will result in a grade of zero and academic dishonesty sanctions. Contact the course staff if you encounter issues with the tests.

Introduction

The files that you will need for this project should now reside in `~/lab`. Before you proceed with the project, compile the zookws webserver via `make`.

The zookws web server consists of the following components:

- `zookld`, a launcher daemon that launches services configured in the file `zook.conf`.
- `zookd`, a dispatcher that routes HTTP requests to corresponding services.
- `zookfs` and other services that may serve static files or execute dynamic scripts.

After `zookld` launches the configured services, `zookd` listens on a port (8080 by default) for incoming HTTP requests and reads the first line of each request for dispatching. In this lab, `zookd` is configured to dispatch every request to the `zookfs` service, which reads the rest of the request and generates a response from the requested file. Most HTTP-related code is in `http.c`. Here is a tutorial of the HTTP protocol.

You will be using two versions of the web server:

- `zookld`, `zookd-exstack`, `zookfs-exstack`, as configured in the file `zook-exstack.conf`
- `zookld`, `zookd-nxstack`, `zookfs-nxstack`, as configured in the file `zook-nxstack.conf`

In the first version, the `*-exstack` binaries have an executable stack, which makes it easier to inject executable code using a stack buffer overflow. The `*-nxstack` binaries have a non-executable stack; you will write exploits that bypass non-executable stacks later in this lab assignment.

In order to run the web server in a predictable fashion---so that its stack and memory layout is the same every time---you will use the `clean-env.sh` script. This is the same way in which we will run the web server during grading, so make sure that all of your exploits work on this configuration!

The reference binaries of zookws are provided in `bin.tar.gz`. We will use those binaries for grading, so make sure that your exploits work on those binaries.

Now, make sure that you can run the zookws web server and start the application via `./clean-env.sh ./zookld zook-exstack.conf`. You should then be able to open your browser and go to http://192.168.26.3:8080/. If something doesn't seem to be working, try to figure out what went wrong before proceeding further.

Part 1: Finding Buffer Overflows

In the first part of this lab assignment, you will find buffer overflows in the provided web server. Read Aleph One's article, "Smashing the Stack for Fun and Profit", as well as "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" (`articles/stack_smashing.txt` and `articles/buffer_overflows.pdf` in the repository, respectively), to figure out how buffer overflows work.

> **Tip**
>
> Since the repository lives on your VM and `scp` is a pain, it's probably easiest to download all files in `articles/` from your fork's GitHub page.

Study the web server's code, and find examples of code that is vulnerable to memory corruption via buffer overflows. Write down a description of each vulnerability in the file `bugs.txt` ; use the format described in that file. For each vulnerability, describe the buffer which may overflow, how you would structure the input to the web server (i.e., the HTTP request) to overflow the buffer, and whether the vulnerability can be prevented using stack canaries. Locate at least 5 different vulnerabilities.

Testing: `make test_bugs` will check if your `bugs.txt` file matches the required format. However, the command will not check whether the bugs that you listed are actual bugs (or whether your analysis of them is correct).

Part 2: Memory Corruption

Now, you will start developing exploits to take advantage of the buffer overflows that you have found. We have provided template Python code for an exploit in `exploit_template.py` . That script issues an HTTP request, and takes two arguments, the server name and port number. So, you might run it as follows to issue a request to zookws running on localhost:

```
httpd@vm263:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf

(in another terminal session)
httpd@vm263:~/lab$ ./exploit-template.py localhost 8080
HTTP request:
GET / HTTP/1.0

...
```

You are free to use this template, or write your own exploit code from scratch. Note, however, that if you choose to write your own exploit, the exploit must run correctly inside the provided virtual machine.

> **Important**

The exploit template is **Python 3** code. You may use Python 2.7 (if you do, be sure to change the first line's `python3` to `python2`), but you are on your own as we only officially support Python 3.

Pick two buffer overflows from the ones you put in `bugs.txt`. The first must overwrite a return address on the stack, and the second must overwrite some other data structure that you will use to take over the control flow of the program.

Then, write exploits that trigger them. For now, you do not need to inject code or do anything other than corrupt memory past the end of the buffer. Verify that your exploit actually corrupts memory, by either checking the last few lines of `dmesg | tail`, using `gdb`, or observing that the web server crashes.

Name these exploits `crash_1.py` and `crash_2.py`. In addition, answer the written questions in `crash.txt` (save your answers directly in the file).

If you believe that a vulnerability in `bugs.txt` is too difficult to exploit, choose a different one.

Testing: `make test_crash_1` and `make test_crash_2` will check that your exploits crash the server via memory corruption (namely, a `SIGSEGV`). Note that this does **not** check `crash.txt`.

Tip

You will find `gdb` useful in building your exploits. As zookws forks off many processes, it can be difficult to debug the correct one. The easiest way to do this is to run the web server ahead of time with `clean-env.sh` and then attaching `gdb` to an already-running process with the `-p` flag. To help find the right process for debugging, zookld prints out the process IDs of the child processes that it spawns. You can also find the PID of a process by using pgrep; for example, to attach to `zookd-exstack`, start the server and, in another shell, run:

```
httpd@vm-CS263:~/lab$ gdb -p $(pgrep zookd-exstack)
...
0x4001d422 in __kernel_vsyscall ()
(gdb) break your-breakpoint
Breakpoint 1 at 0x1234567: file zookd.c, line 999.
(gdb) continue
Continuing.
```

Keep in mind that a process being debugged by `gdb` will not get killed even if you terminate the parent zookld process using `Ctrl-C`. If you are having trouble restarting the web server, check for leftover processes from the previous run, or be sure to exit `gdb` before restarting zookld.

When a process being debugged by `gdb` forks, by default `gdb` continues to debug the parent process and does not attach to the child. Since zookfs forks a child process to service each request, you may find it helpful to have `gdb` attach to the child on fork, using the command `set follow-fork-mode child`. We have added

that command to `.gdbinit`, which will take effect if you start `gdb` in that directory.

Tip

For this and subsequent tasks, you may need to encode your attack payload in different ways, depending on which vulnerability you are exploiting. In some cases, you may need to make sure that your attack payload is URL-encoded, i.e.,, using `+` instead of space and `%2b` instead of `+`. Here is a URL encoding reference. You can also use the quoting functions in the Python urllib module to URL-encode strings (see the exploit template for an example).

In other cases, you may need to include binary values into your payload. The Python struct module can help you do that. For example, `struct.pack(b'<I', x)` will produce a 4-byte (32-bit) little-endian binary encoding of the integer `x`.

Part 3: Code Injection via Buffer Overflow

In this part, you will use your buffer overflows to inject code into the web server. The goal of the injected code will be to `unlink` (i.e., remove) a sensitive file on the server, namely `/home/httpd/grades.txt`. Use the `*-exstack` binaries (via configuration files, as discussed before), since they have an executable stack that makes it easier to inject code. The zookws web server should be started via `./clean-env.sh ./zookld zook-exstack.conf`.

Shell Code

We have provided Aleph One's shell code for you to use in `shellcode.S`, along with `Makefile` rules that produce `shellcode.bin`, a compiled version of the shell code, when you run make. Aleph One's exploit is intended to exploit `setuid-root` binaries, and thus it runs a shell. You will need to modify this shell code to instead unlink `/home/httpd/grades.txt`. This part is ungraded, but you will most likely need `shellcode.bin` for your injection attack.

Tip

To help you develop your shell code for this task, we have provided a program called `run-shellcode` that will run your binary shell code, as if you correctly jumped to its starting point. For example, running it on Aleph One's shell code will cause the program to `execve("/bin/sh")`, thereby giving you another shell prompt:

```
httpd@vm263:~/lab$ ./run-shellcode shellcode.bin
$
```

Injection Attack

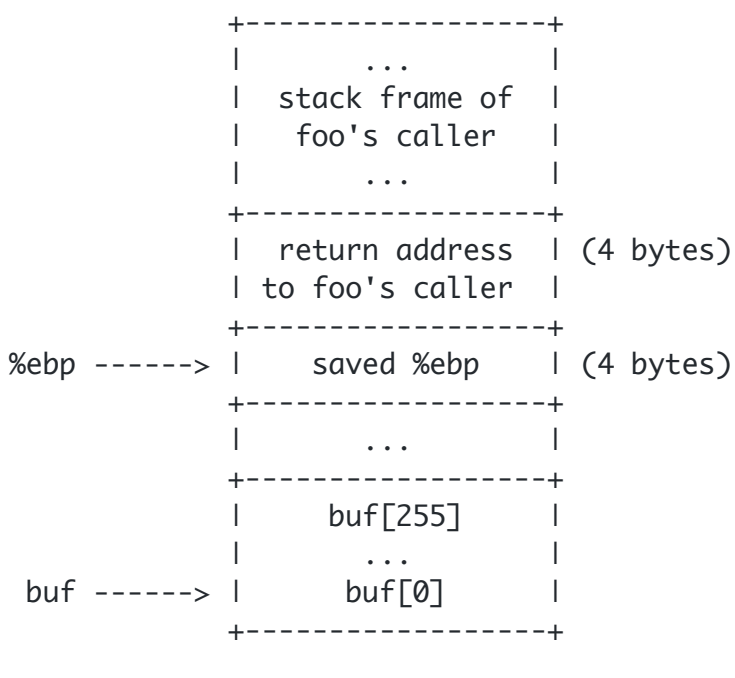
Starting from one of your memory corruption exploits, construct an exploit that hijacks control flow of the web server to unlink `/home/httpd/grades.txt`. Save this exploit in a file called `unlink_exstack.py`. In addition, answer the written questions in `unlink_exstack.txt` (save your answers directly in the file).

Verify that your exploit works; you will need to re-create `/home/httpd/grades.txt` after each successful exploit run.

> **Important**
>
> It's OK to hardcode things -- however, you should be careful that your hardcoded things won't break when we're testing your code. This means, among other things, that your repository's root directory should be `/home/httpd/lab/`.

Testing: `make test_unlink_exstack` will check that your exploit unlinks `/home/httpd/grades.txt` with an executable stack. Note that this does **not** check `unlink_exstack.txt`.

> **Tip**
>
> When developing an exploit, you will have to think about what values are on the stack, so that you can modify them accordingly. For your reference, here is what the stack frame of some function `foo()` looks like; here, `foo()` has a local variable `char buf[256]`:



Note that the stack grows down in this figure, and memory addresses are increasing up.

Tip

When developing an exploit, you will often need to know the addresses of specific stack locations, or specific functions, in a particular program. The easiest way to do this is to use `gdb`. For example, suppose you want to know the stack address of the `pn[]` array in the `http_serve()` function in `zookfs-exstack`, and the address of its saved `%ebp` register on the stack. You can obtain them using `gdb` as follows:

```
httpd@vm-CS263:~/lab$ gdb -p $(pgrep zookfs-exstack)
...
0x40022416 in __kernel_vsyscall ()
(gdb) break http_serve
Breakpoint 1 at 0x8049415: file http.c, line 248.
(gdb) continue
Continuing.
```

Be sure to run `gdb` from the `~/lab` directory, so that it picks up the `set follow-fork-mode child` command from `~/lab/.gdbinit`. Now you can issue an HTTP request to the web server, so that it triggers the breakpoint, and so that you can examine the stack of `http_serve()`:

```
[New process 1339]
[Switching to process 1339]

Breakpoint 1, http_serve (fd=3, name=0x8051014 "/") at http.c:248
248     void (*handler)(int, const char *) = http_serve_none;
(gdb) print &pn
$1 = (char (*)[1024]) 0xbfffd10c
(gdb) info registers
eax            0x3  3
ecx            0x400bdec0 1074519744
edx            0x6c6d74 7105908
ebx            0x804a38e  134521742
esp            0xbfffd0a0 0xbfffd0a0
ebp            0xbfffd518 0xbfffd518
esi            0x0  0
edi            0x0  0
eip            0x8049415  0x8049415 <http_serve+9>
eflags         0x200286 [ PF SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0  0
```

```
gs             0x33 51
(gdb)
```

From this, you can tell that, at least for this invocation of `http_serve()`, the `pn[]` buffer on the stack lives at address `0xbfffd10c`, and the value of `%ebp` (which points at the saved `%ebp` on the stack) is `0xbfffd518`.

Hint

Here's a suggested plan of attack for this task:

First, focus on obtaining control of the program counter. Sketch out the stack layout that you expect the program to have at the point when you overflow the buffer, and use `gdb` to verify that your overflow data ends up where you expect it to. Step through the execution of the function to the return instruction to make sure you can control what address the program returns to. The `next`, `stepi`, `info reg`, and `disassemble` commands in `gdb` should prove helpful.

Once you can reliably hijack the control flow of the program, find a suitable address that will contain the code you want to execute, and focus on placing the correct code at that address (perhaps from `shellcode.bin`).

Note that `SYS_unlink`, the number of the unlink syscall, is 10 or `'\n'` (newline). Why does this complicate matters? How can you get around it?

Part 4: Return-to-libc Attacks

Many modern operating systems mark the stack as non-executable in an attempt to make it more difficult to exploit buffer overflows. In this part, you will explore how this protection mechanism can be circumvented. You'll need to run the web server configured with binaries that have a non-executable stack via `./clean-env.sh ./zookld zook-nxstack.conf`.

Starting from your two memory corruption exploits, construct two additional exploits that unlink `/home/httpd/grades.txt` when run on the binaries that have a non-executable stack. Name these new exploits `unlink_libc_1.py` and `unlink_libc_2.py`. In addition, answer the written questions in `unlink_libc.txt` (save your answers directly in the file).

Verify that your exploits work; you will need to re-create `/home/httpd/grades.txt` after each successful exploit run.

Testing: `make test_unlink_libc_1` and `make test_unlink_libc_2` will check that your exploits unlink `/home/httpd/grades.txt` with a non-executable stack. Note that this does **not** check `unlink_libc.txt`.

Important

Although in principle you could use shellcode that's not located on the stack, for this task you should not inject any shellcode into the vulnerable process. You should use a return-to-libc (or at least a call-to-libc) attack where you divert control flow directly into `libc` code that existed before your attack.

Tip

The key observation to exploiting buffer overflows with a non-executable stack is that you still control the program counter, after a `RET` instruction jumps to an address that you placed on the stack. Even though you cannot jump to the address of the overflowed buffer (it will not be executable), there's usually enough code in the vulnerable server's address space to perform the operation you want.

Thus, to bypass a non-executable stack, you need to first find the code you want to execute. This is often a function in the standard library, called `libc`; examples of functions which are often useful are `execl`, `system`, or `unlink`. Then, you need to arrange for the stack to look like a call to that function with the desired arguments, such as `system("/bin/sh")`. Finally, you need to arrange for the `RET` instruction to jump to the function you found in the first step. This attack is often called a return-to-libc attack. The file `articles/return_to_libc.txt` contains a more detailed description of this style of attack.

Tip

You will need to understand the calling convention for C functions. For your reference, consider the following simple C program:

```
void
foo(int x, char *msg, int y)
{
     /* ... */
}

void
bar(void)
{
    int a = 3;
    foo(5, "Hello, world!", 7);
}
```

The stack layout when `bar()` invokes `foo()`, just after the program counter has switched to the beginning of `foo()`, looks like this:

```
                +------------------+
   %ebp ------> |    saved %ebp    | (4 bytes)
```

```
                +------------------+
                |       ...        |
                +------------------+
bar's a ------> |        3         | (4 bytes)
                +------------------+
                |       ...        |
                +------------------+
                |        7         | (4 bytes)
                +------------------+
                |    pointer to    | ------>  "Hello, world!", somewhere
                |      string      | (4 bytes)                  in memory
                +------------------+
                |        5         | (4 bytes)
                +------------------+
                |  return address  | (4 bytes)
   %esp ------> |     into bar     |
                +------------------+
                |                  |
```

When foo starts running, the first thing it will do is save the `%ebp` register on the stack, and set the `%ebp` register to point at this saved value on the stack, so the stack frame will look like the one shown for foo.

Part 5: Finding Other Vulnerabilities

Now that you have figured out how to exploit buffer overflows, you will try to find other kinds of vulnerabilities in the same code. As with many real-world applications, the "security" of our web server is not well-defined. Thus, you will need to use your imagination to think of a plausible threat model and policy for the web server.

Look through the source code and try to find more vulnerabilities that can allow an attacker to compromise the security of the web server. Describe the attacks you have found in `attacks.txt`, along with an explanation of the limitations of the attack, what an attacker can accomplish, why it works, and how you might go about fixing or preventing it. You should ignore bugs in `zoobar/`.

You should find at least two vulnerabilities for this exercise.

Testing: on your own.

> **Tip**
>
> One approach for finding vulnerabilities is to trace the flow of inputs controlled by the attacker through the server code. At each point that the attacker's input is used, consider all the possible values that the attacker might have provided at that point; consider what the attacker can achieve.

Part 6: Fixing Buffer Overflows

Finally, you will explore fixing some of the vulnerabilities that you have found in this lab assignment. For each buffer overflow vulnerability you have found in `bugs.txt`, fix the web server's code to prevent the vulnerability in the first place. Above each modified code block, add a comment stating which bug from `bugs.txt` is fixed.

Commit these fixes directly to your fork. Don't worry about your fixes breaking your previous exploits, as the test scripts will always use the original (buggy) binaries.

Testing: on your own.

> **Caution!**
>
> Do not rely on compile-time or runtime mechanisms such as stack canaries, removing `-fno-stack-protector`, baggy bounds checking, etc.

Submitting

> **Important**
>
> Before submitting, make sure all your work is committed and pushed to the master branch of your fork.

> **Caution!**
>
> Due to the specificity of the VM environment, there is no Travis build for this project. Thus, you need to be especially careful in making sure that `make test` succeeds locally (with all files committed and pushed).
>
> If you want to make sure your submission is OK, here's how we're going to grade your code:
>
> - Create a clean VM from the vm263 image.
> - Login and clone your repository.
> - `make -k test`

On the fork's GitHub page. click on "New pull request". The base fork should be the original repository (prefixed with `harvard-cs263`), and the head fork should be your fork (prefixed with

your GitHub username). Then, click on "Create pull request" to submit your work! The title can be whatever, and the comment can be left blank (or non-blank if you have a note for the grader).

If you need to edit your submission before the deadline, just commit and push your new changes to the master branch of your fork. The original pull request will be automatically updated with those commits (of course, be sure to check the GitHub pull request page to verify).

> **Caution!**
>
> Do **not** click "Merge pull request" after submitting, as this will modify the original repository. We will merge your pull request when grading.

> **Caution!**
>
> The deadlines for all assignments are on Canvas. Deadlines are enforced to the minute (based on pull request/push times, not commit times), and the course late policy is a 10% deduction per 8 hours of lateness.

Deliverables and Rubric

"Script" grading means we will assign points based on the result of the relevant `make test_blah` command.

Criteria	Points	Grading method
`bugs.txt`	15	Manual
`crash_1.py` and `crash_2.py`	16	Script
`crash.txt`	2	Manual
`unlink_exstack.py`	16	Script
`unlink_exstack.txt`	2	Manual
`unlink_libc_1.py` and `unlink_libc_2.py`	26	Script
`unlink_libc.txt`	4	Manual
`attacks.txt`	10	Manual
Buffer overflow fixes	9	Manual

Acknowledgements

This project was derived from one offered by MIT's 6.858 class.