Lab 1 - Memory Vulnerabilities

Lab 1 will introduce you to buffer overflow vulnerabilities, in the context of a web server called `zookws`. The `zookws` web server runs a simple python web application, `zoobar`, with which users transfer "**zoobars**" (credits) between each other. You will find buffer overflows in the `zookws` web server code, write exploits for the buffer overflows to inject code into the server over the network, and figure out how to bypass non-executable stack protection. Later labs look at other security aspects of the `zoobar` and `zookws` infrastructure.

Getting started

The files you will need for this and subsequent labs are distributed using the Gitoverview of GitGit user's manual

The course Git repository is available at **https://web.mit.edu/6858/2019/lab.git**. To get the lab code, log into the VM using the `httpd` account and clone the source code for lab 1 as follows:

```
httpd@istd:~$ git clone https://web.mit.edu/6858/2019/lab.git
Cloning into 'lab'...
httpd@istd:~$ cd lab
httpd@istd:~/lab$
```

Before you proceed with this lab assignment, make sure you can compile the `zookws` web server:

```
httpd@istd:~/labs$ make
cc zookd.c -c -o zookd.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-
stack-protector
cc http.c -c -o http.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-
protector
cc -m64  zookd.o http.o  -lcrypto -o zookd
cc -m64 zookd.o http.o  -lcrypto -o zookd-exstack -z execstack
cc -m64 zookd.o http.o  -lcrypto -o zookd-nxstack
cc zookd.c -c -o zookd-withssp.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static
cc http.c -c -o http-withssp.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static
cc -m64  zookd-withssp.o http-withssp.o  -lcrypto -o zookd-withssp
cc -m64   -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
cc run-shellcode.c -c -o run-shellcode.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -
static -fno-stack-protector
cc -m64  run-shellcode.o  -lcrypto -o run-shellcode
rm shellcode.o
httpd@istd:~/labs$
```

The component of `zookws` that receives HTTP requests is `zookd`. It is written in C and serves static files and executes dynamic scripts. For this lab you don't have to understand the dynamic scripts; they are written in Python and the exploits in this lab apply only to C code. The HTTP-related code is in `http.c`. Here is a tutorial about the HTTP protocol.

There are two versions of `zookd` you will be using:

- `zookd-exstack`

- `zookd-nxstack`

`zookd-exstack` has an executable stack, which makes it easy to inject executable code given a stack buffer overflow vulnerability. has a non-executable stack, and requires a more sophisticated attack to exploit stack buffer overflows.

The reference binaries of `zookd` are provided in `bin.tar.gz`, which we will use for grading. Make sure your exploits work on those binaries. The **make check** command will always use both `clean-env.sh` and `bin.tar.gz` to check your submission.

Now, make sure you can run the `zookws` web server and access the `zoobar` web application from a browser running on your machine, as follows:

```
httpd@istd:~/lab$ ./clean-env.sh ./zookd 8080
```

The `./clean-env.sh` commands starts `zookd` on port 8080. To open the zoobar application, open your browser and point it at the URL `http://IPADDRESS:8080/`, where `IPADDRESS` is the VM's IP address we found above. If something doesn't seem to be working, try to figure out what went wrong, or contact the course staff, before proceeding further.

A) Finding buffer overflows

In the first part of this lab assignment, you will find buffer overflows in the provided web server. To do this lab, you will need to understand the basics of buffer overflows. To help you get started with this, you should read Smashing the Stack in the 21st Century, which goes through the details of how buffer overflows work, and how they can be exploited.

Now, you will start developing exploits to take advantage of the buffer overflows you have found above. We have provided template Python code for an exploit in `/home/httpd/lab/exploit-template.py`, which issues an HTTP request. The exploit template 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@istd:~/lab$ ./clean-env.sh ./zookd-exstack 8080 &
[1] 2676
httpd@istd:~/lab$ ./exploit-template.py localhost 8080
HTTP request:
GET / HTTP/1.0

...
httpd@istd:~/lab$
```

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.

You may find `gdb` useful in building your exploits (though it is not required for you to do so). As `zookd` forks off many processes (one for each client), 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. You can 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@istd:~/lab$ gdb -p $(pgrep zookd-)
...
(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 `zookd` process using `^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 `zookd`. You can also save yourself some typing by using `b` instead of `break`, and `c` instead of `continue`.

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 `zookd` 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 `/home/httpd/lab/.gdbinit`, which will take effect if you start `gdb` in that directory.

For this and subsequent exercises, 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; that is, use `+` instead of space and `%2b` instead of `+`. Here is a URL encoding reference and a handy conversion tool. You can also use quoting functions in the python `urllib` module to URL encode strings. 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("<Q", x)` will produce an 8-byte (64-bit) binary encoding of the integer `x`.

You can check whether your exploits crash the server as follows:

```
httpd@istd:~/lab$ make check-crash
```

B) Code injection

In this part, you will use your buffer overflow exploits to inject code into the web server. The goal of the injected code will be to unlink (remove) a sensitive file on the server, namely `/home/httpd/grades.txt`. Use `zookd-exstack`, since it has an executable stack that makes it easier to inject code. The `zookws` web server should be started as follows.

```
httpd@istd:~/lab$ ./clean-env.sh ./zookd-exstack 8080
```

You can build the exploit in two steps. First, write the shell code that unlinks the sensitive file, namely `/home/httpd/grades.txt`. Second, embed the compiled shell code in an HTTP request that triggers the buffer overflow in the web server.

When writing shell code, it is often easier to use assembly language rather than higher-level languages, such as C. This is because the exploit usually needs fine control over the stack layout, register values and code size. The C compiler will generate additional function preludes and perform various optimizations, which makes the compiled binary code unpredictable.

We have provided shell code for you to use in `/home/httpd/lab/shellcode.S`, along with `Makefile` rules that produce `/home/httpd/lab/shellcode.bin`, a compiled version of the shell code, when you run **make**. The provided shell code 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`.

To help you develop your shell code for this exercise, 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 the provided shell code will cause the program to `execve("/bin/sh")`, thereby giving you another shell prompt:

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

To test whether the shell code does its job, run the following commands:

```
httpd@istd:~/lab$ make
httpd@istd:~/lab$ touch ~/grades.txt
httpd@istd:~/lab$ ./run-shellcode shellcode.bin
# Make sure /home/httpd/grades.txt is gone
httpd@istd:~/lab$ ls ~/grades.txt
ls: cannot access /home/httpd/grades.txt: No such file or directory
```

You may find strace useful when trying to figure out what system calls your shellcode is making. Much like with `gdb`, you attach `strace` to a running program:

```
httpd@istd:~/lab$ strace -f -p $(pgrep zookd-)
```

It will then print all of the system calls that program makes. If your shell code isn't working, try looking for the system call you think your shell code should be executing (i.e., `unlink`

Next, we construct a malicious HTTP request that injects the compiled byte code to the web server, and hijack the server's control flow to run the injected code. When developing an exploit, you will have to think about what values are on the stack, so that you can modify them accordingly.

When you're constructing an exploit, you will often need to know the addresses of specific stack locations, or specific functions, in a particular program. One way to do this is to add `printf()` statements to the function in question. For example, you can use `printf("Pointer: %p\n", &x);` to print the address of variable `x` or function `x`. However, this approach requires some care: you need to make sure that your added statements are not themselves changing the stack layout or code layout. We (and **make check**) will be grading the lab without any `printf` statements you may have added.

A more fool-proof approach to determine addresses is to use `gdb`. For example, suppose you want to know the stack address of the `pn[]` array in the `http_serve` function in `zookd-exstack`, and the address of its saved return pointer. You can obtain them using `gdb` by first starting the web server (remember `clean-evn`!), and then attaching `gdb` to it:

```
httpd@istd:~/lab$ gdb -p $(pgrep zookd-)
...
(gdb) break http_serve
Breakpoint 1 at 0x5555555561d2: file http.c, line 275.
(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`.

```
httpd@istd:~/lab$ curl localhost:8080
```

This will cause `gdb``gdb`

```
Thread 2.1 "zookd-exstack" hit Breakpoint 1, http_serve (fd=4,
name=0x55555575f80c "/") at http.c:275
275         void (*handler)(int, const char *) = http_serve_none;
(gdb) print &pn
$1 = (char (*)[2048]) 0x7fffffffd4b0
(gdb) info frame
Stack level 0, frame at 0x7fffffffdce0:
 rip = 0x55555555622e in http_serve (http.c:275); saved rip = 0x5555555558e5
 called by frame at 0x7fffffffed10
 source language c.
 Arglist at 0x7fffffffdcd0, args: fd=4, name=0x55555575f80c "/"
 Locals at 0x7fffffffdcd0, Previous frame's sp is 0x7fffffffdce0
 Saved registers:
  rbx at 0x7fffffffdcc8, rbp at 0x7fffffffdcd0, rip at 0x7fffffffdcd8
(gdb)
```

From this, you can tell that, at least for this invocation of `http_serve`, the `pn[]` buffer on the stack lives at address `0x7fffffffd4b0`, and the saved value of `%rip` (the return address in other words) is at `0x7fffffffdcd8`. If you want to see register contents, you can also use **info registers**.

:::tip

Exercise 3.2 - Develop an exploit

Now it's your turn to develop an exploit.

You can check whether your exploit works as follows:

```
httpd@istd:~/lab$ make check-exstack
```

The test either prints "PASS" or "FAIL". We will grade your exploits in this way. Do not change the `Makefile`.

The standard C compiler used on Linux, gcc, implements a version of stack canaries (called SSP). You can explore whether GCC's version of stack canaries would or would not prevent a given vulnerability by using the SSP-enabled versions of `zookd`: `zookd-withssp`.

Submit your answers to the first two parts of this lab assignment by running **make submit-a**. Alternatively, run **make prepare-submit-a** and upload the resulting `lab1a-handin.tar.gz` file to the submission web site.

:::

C) Return-to-libc attacks

Many modern operating systems mark the stack 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. Run the web server configured with binaries that have a non-executable stack, as follows.

```
httpd@istd:~/lab$ ./clean-env.sh ./zookd-nxstack 8080
```

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, such as `execve`, `system`, or `unlink`. Then, you need to arrange for the stack and registers to be in a state consistent with calling that function with the desired arguments. 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.

One challenge with return-to-libc attacks is that you need to pass arguments to the libc function that you want to invoke. The x86-64 calling conventions make this a challenge because the first 6 arguments are passed in registers. For example, the first argument must be in register `%rdi` (see `man 2 syscall`, which documents the calling convention). So, you need an instruction that loads the first argument into `%rdi`. In Exercise 3, you could have put that instruction in the buffer that your exploit overflows. But, in this part of the lab, the stack is marked non-executable, so executing the instruction would crash the server, but wouldn't execute the instruction.

The solution to this problem is to find a piece of code in the server that loads an address into `%rdi`. Such a piece of code is referred to as a "borrowed code chunk", or more generally as a *rop gadget*, because it is a tool for return-oriented programming (rop). Luckily, `zookd.c` accidentally has a useful gadget: see the function `accidentally`.

You can test your exploits as follows:

```
httpd@istd:~/lab$ make check-libc
```

D) Fixing buffer overflows and other bugs

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.

Finally, you will fix the vulnerabilities that you have exploited in this lab assignment.

:::tip

Exercise 6 - Fixing vulnerabilities

For each buffer overflow vulnerability you have exploited in Exercises 2, 3, and 4, fix the web server's code to prevent the vulnerability in the first place. Do not rely on compile-time or runtime mechanisms such as stack canaries, removing `-fno-stack-protector`, baggy bounds checking, etc.

Make sure that your code actually stops your exploits from working. Use **make check-fixed** to run your exploits against your modified source code (as opposed to the staff reference binaries from `bin.tar.gz`). These checks should report FAIL (i.e., exploit no longer works). If they report PASS, this means the exploit still works, and you did not correctly fix the vulnerability.

Note that your submission should *not* make changes to the `Makefile` and other grading scripts. We will use our unmodified version during grading.

You should also make sure your code still passes all tests using **make check**, which uses the unmodified lab binaries.

:::