ECE455
Lab #1: Buffer Overflows

This lab is an adaptation of an assignment developed by MIT Course 6.858.

Overview
Goals

- Become familiar with Buffer Overflow attacks
- Gain practical experience with C, x86 assembly, and gdb

Grading & Submission

- This lab is graded holistically, best efforts will be considered. This is a learning experience, not a test!
- Submit the generated .tar.gz files via Teams.
- **There are two submission windows:**
    - **Parts 1 & 2** Wednesday, October 13 2020 by 11:59pm
    - **Parts 3 & 4** Wednesday, October 20 2020 by 11:59pm

Introduction
This lab will give you practical experience with common attacks and countermeasures. To make security issues concrete, you will explore the attacks and countermeasures in the context of the zoobar web application. In this lab, you will explore the base structure of the zoobar web application (zookws), and use buffer overrun attacks to break its security properties.

The zookws web server is running a simple python web application, zoobar, where 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, figure out how to bypass non-executable stack protection, and finally look for other potential problems in the web server implementation.

Security weaknesses often hide in corner cases, and so you need to understand the details to craft exploits and design defenses for those corner cases. This can make the lab time consuming. You should start early on the labs and work on them daily for some limited time, instead of trying to do all exercises in a single shot before the deadline. You should also try to understand the necessary details, instead of muddling your way through the exercises. If you get stuck on a detail, you may work with another student or ask the instructor.

This lab will ask you to design exploits. These exploits are realistic enough that you might be able to use them for a real attacks, but you should not do so. The point of the designing exploits is to teach you how to defend against them, not how to use them. Attacking computer systems is illegal and can get you into serious trouble. Don't do it.

Lab infrastructure
Exploiting buffer overflows requires precise control over the execution environment. A small change in the compiler, environment variables, or the way the program is executed can result in slightly different memory layout and code structure, thus requiring a different exploit. For this reason, this lab uses a VMware virtual machine to run the vulnerable web server code.

To start working on this lab assignment, you should download the VMware Player, which can run virtual machines on Linux and Windows systems. For Mac users, the equivalent program is VMware Fusion. You can get an individual student license from the VMware website. Once you have VMware installed on your machine, you should download the course VM image, and unpack it on your computer.

This virtual machine contains an installation of Ubuntu 14.04.1 Linux, and the following accounts have been created inside the VM.

Username	Password	Description
root	6858	You can use the root account to install new software packages into the VM, if you find something missing, using **`apt-get install <package>`**.
httpd	6858	The httpd account is used to execute the web server, and contains the source code you will need for this lab assignment, in /home/httpd/lab.

For Linux users, MIT also tested running the course VM on KVM, which is built into the Linux kernel and should be much easier to get working than VMware. KVM should be available through your distribution. On Debian or Ubuntu, try **`apt-get install qemu-kvm`**. Once installed, you should be able to run a command like:

```
kvm -m 512 -net nic -net user,hostfwd=tcp:127.0.0.1:2222-:22,hostfwd=tcp:127.0.0.1:8080-:8080 vm-
6858.vmdk
```

to run the VM and forward the relevant ports.

You can either log into the virtual machine using its console, or you can use ssh to log into the virtual machine over the (virtual)network. To determine the virtual machine's IP address, log in as root on the console and run:

```
/sbin/ifconfig eth0
```

(If using KVMwith the command above, then **`ssh -p 2222 httpd@localhost`** should work.)

Download lab1.zip from on the MIT OpenCourseWare site or use the zip attached to this assignment. To begin, log into the VM using the httpd account and copy over or download the zip.

First, make sure you can compile the zookws web server:

```
httpd@vm-6858:~/lab1$ make clean
rm -f *.o *.pyc *.bin zookld zookfs zookd zookfs-exstack zookd-exstack zookfs-nxstack zookd-nxstack
zookfs-withssp zookd-withssp shellcode.bin run-shellcode
httpd@vm-6858:~/lab1$ make clean all
rm -f *.o *.pyc *.bin zookld zookfs zookd zookfs-exstack zookd-exstack zookfs-nxstack zookd-nxstack
zookfs-withssp zookd-withssp shellcode.bin run-shellcode
cc zookld.c -c -o zookld.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc http.c -c -o http.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  zookld.o http.o  -lcrypto -o zookld
cc zookfs.c -c -o zookfs.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  zookfs.o http.o  -lcrypto -o zookfs
cc zookd.c -c -o zookd.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  zookd.o http.o  -lcrypto -o zookd
cp zookfs zookfs-exstack
execstack -s zookfs-exstack
cp zookd zookd-exstack
execstack -s zookd-exstack
cp zookfs zookfs-nxstack
cp zookd zookd-nxstack
cc zookfs.c -c -o zookfs-withssp.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
cc http.c -c -o http-withssp.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
cc -m32  zookfs-withssp.o http-withssp.o  -lcrypto -o zookfs-withssp
cc zookd.c -c -o zookd-withssp.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
cc -m32  zookd-withssp.o http-withssp.o  -lcrypto -o zookd-withssp
cc -m32   -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 -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-
```

```
protector
cc -m32  run-shellcode.o  -lcrypto -o run-shellcode
rm shellcode.o
```

The 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 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.

There are two versions of the web server you will be using:

- 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 one, the *-exstack binaries have an executable stack, which makes it easier to inject executable code with a stack overflow attack. The *-nxstack binaries in the second version have a non-executable stack, and you will write exploits that bypass non-executable stacks later in this lab. 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 all of your exploits work on this configuration!

The reference binaries of zookws are provided in bin.tar.gz, which we will use for grading. Make sure your exploits work on those binaries.

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@vm-6858:~/lab1$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          inet6 addr: fec0::682a:55b6:636d:cfd7/64 Scope:Site
          inet6 addr: fec0::5054:ff:fe12:3456/64 Scope:Site
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6704 errors:854 dropped:0 overruns:0 frame:854
          TX packets:1968 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8274865 (8.2 MB)  TX bytes:179403 (179.4 KB)

httpd@vm-6858:~/lab1$ ./clean-env.sh ./zookld zook-exstack.conf
```

The `/sbin/ifconfig` command will give you the virtual machine's IP address. In this particular example, you would want to open your browser and go to the URL `http://10.0.2.15:8080/`. (If you're using KVM with the command above, just access `http://localhost:8080/` on your host.) If something doesn't seem to be working, try to figure out what went wrong with a fellow student or me, before proceeding.

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*, for more background information on this type of attack.

Exercise 1

Study the web server's code, and find examples of code vulnerable to memory corruption through a buffer overflow. Write down a description of each vulnerability in the file `/home/httpd/lab/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.

You can use the command `make check-bugs` to check if your `bugs.txt` file matches the required format, although the command will not check whether the bugs you listed are actual bugs or whether your analysis of them is correct.

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@vm-6858:~/lab1$ ./clean-env.sh ./zookld zook-exstack.conf &
[1] 1451
httpd@vm-6858:~/lab1$ ./exploit-template.py localhost 808
HTTP request:
GET / HTTP/1.0
…
httpd@vm-6858:~/lab1$
```

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 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-6858:~/lab$ gdb -p $(pgrep zookd-exstack)
...
Loaded symbols for /lib/ld-linux.so.2
0x40022424 in __kernel_vsyscall ()
(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 ^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`. That command can be added 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("<I", x)` will produce a 4-byte (32-bit) binary encoding of the integer `x`.

Exercise 2

Pick two buffer overflows out of what you have found for later exercises (although you can change your mind later, if you find your choices are particularly difficult to exploit). 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.

Write exploits that trigger them. You do not need to inject code or do anything other than corrupt memory past the end of the buffer, at this point. 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.

Provide the code for the exploits in files called `exploit-2a.py` and `exploit-2b.py`, and indicate in `answers.txt` which buffer overflow each exploit triggers. If you believe some of the vulnerabilities you have identified in Exercise 1 cannot be exploited, choose a different vulnerability.

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

```
httpd@vm-6858:~/lab$ make check-crash
```

Part 2: 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 the *-exstack binaries, since they have an executable stack that makes it easier to inject code. The zookws web server should be started as follows.

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf
```

We have provided Aleph One's 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`. 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`.

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

```
httpd@vm-6858:~/lab$ ./run-shellcode shellcode.bin
$
```

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]`

```
                       HIGH ADDR
                  +------------------+
                  |                  |
                  |  stack frame of  |
                  |   foo's caller   |
                  |       ...        |
                  +------------------+
                  |  return address  | (4 bytes)
                  |  to foo's caller |
                  +------------------+
%ebp ------>      |    saved %ebp    | (4 bytes)
                  +------------------+
                  |       ...        |
                  +------------------+
                  |      buf[255]    |
                  |        ...       |
buf ------>       |      buf[0]      |
                  +------------------+
                        LOW ADDR
```

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

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. 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-6858:~/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 0edi 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, andthe value of %ebp (which points at the saved %ebp on the stack) is 0xbfffd518.

Now it's your turn to develop an exploit.

Exercise 3

Starting from one of your exploits from Exercise 2, construct an exploit that hijacks control flow of the web server and `unlinks /home/httpd/grades.txt`. Save this exploit in a file called exploit-3.py.

Explain in `answers.txt` whether or not the other buffer overflow vulnerabilities you found in Exercise 1 can be exploited in this manner.

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

Suggestion: 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, such as a derivative of Aleph One's shell code.

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

You can check whether your exploit works as follows:

```
httpd@vm-6858:~/lab$ make check-exstack
```

The test either prints "PASS" or fails. We will grade your exploits in this way. If you use another name for the exploit script, change Makefile accordingly.

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 the web server binaries (zookd-withssp and zookfs-withssp), by using the `zook-withssp.conf` config file when starting zookld.

Submission
Submit your answers to the first two parts of this lab assignment by running `make prepare-submit-a` and submit the resulting lab1a-handin.tar.gz file.

Part 3: 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. Inthis 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@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-nxstack.conf
```

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 and execute (it will not be executable), there's usually enough code in the vulnerable server's address space to perform the operation you want.

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 `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. This article contains a more detailed description of this style of attack.

In the next exercise, 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 in memory
                |      string      | (4 bytes)
                +------------------+
                |        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 just above Exercise 3.

Exercise 4

Starting from your two exploits in Exercise 2, construct two exploits that take advantage of those vulnerabilities to `unlink /home/httpd/grades.txt` when run on the binaries that have a non-executable stack. Name these new exploits `exploit-4a.py` and `exploit-4b.py`.

Although in principle you could use shellcode that's not located on the stack, for this exercise 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 vector control flow directly into code that existed before your attack.

In `answers.txt`, explain whether or not the other buffer overflow vulnerabilities you found in Exercise 1 can be exploited in this same manner.

You can test your exploits as follows:

```
httpd@vm-6858:~/lab$ make check-libc
```

The test either prints two "PASS" messages or fails. We will grade your exploits in this way. If you use other names for the exploit scripts, change `Makefile` accordingly.

Part 4: 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.

Exercise 5

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 `answers.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 orpreventing it. You should ignore bugs in zoobar's code. They will be addressed in future labs.

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 the attacker might have provided at that point, and what the attacker can achieve in that manner.

You should find at least two vulnerabilities for this exercise.

Finally, you will explore fixing some of the vulnerabilities you have found in this lab assignment.

Exercise 6

For each buffer overflow vulnerability you have found in Exercise 1, fix the web server's code to prevent the vulnerability in the first place. Do not rely on compile-time or run-time mechanisms such as stack canaries,removing `-fno-stack-protector`, baggy bounds checking, etc.

You are done!
Submit your answers to the first two parts of this lab assignment by running `make prepare-submit-a` and submit the resulting lab1a-handin.tar.gz file.

Adapted from:

MIT OpenCourseWare
http://ocw.mit.edu
6.858 Computer Systems Security
Fall 2014
Non-commercial Use Only