Computer System Security (INGI2347)
Project 1: Buffer Overflows

Due 2nd March 2016 at 11:59pm

Project setup

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 project uses a virtual machine to run the vulnerable web server code.

To work on this project assignment, you will use some well-known tool for managing the virtual machine environment, namely Vagrant. We supply you with pre-provisioned VM that will serve as the *gold standard* for grading the project.

Note: These instructions assume you are working on a machine of your own, where you have permission to install new software. If you can only access the machine in the computer rooms, we can supply you with a pre-made VM to be run there. Please let us know if you need one such pre-provisioned VM.

1. Install Vagrant from https://www.vagrantup.com/downloads

2. Install VirtualBox from https://www.virtualbox.org/wiki/Downloads

3. Clone the course repository by running
   `git clone https://github.com/mcanini/INGI2347-2016.git`
   If you don't have git, you can install it from http://git-scm.com/downloads.

4. Download the VM (INGI2347-vm) from
   http://perso.uclouvain.be/marco.canini/ingi2347/INGI2347-vm.box and put it in the `INGI2347-2016/INGI2347-vm` folder.

5. Start the VM by running `vagrant up` in the `INGI2347-vm` directory (You can terminate the VM by running `vagrant halt`).

6. Log into the VM by executing `vagrant ssh` from the host console.

Note that the `INGI2347-2016` serves as a shared directory between the VM and your machine. On the VM, this directory is mounted under `/vagrant`.

The project directory is available inside the VM at the path `/project1` and you should only work inside that path:

```
vagrant@INGI2347-vm:~$ cd /project1
```

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

```
vagrant@INGI2347-vm:/project1$ make
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
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
vagrant@INGI2347-vm:/project1$
```

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 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 project, `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 this web server: `zookld, zookd-exstack, zookfs-exstack`, as configured in the file `zook-exstack.conf`. In this one, the `*-exstack` binaries have an executable stack, which makes it easier to inject executable code given a stack buffer overflow vulnerability.

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:

```
vagrant@INGI2347-vm:/project1$ ./clean-env.sh ./zookld
zook-exstack.conf
```

From your host machine, you would want to open your browser and go to the URL http://localhost:8080/. If something doesn't seem to be working, try to figure out what went wrong, or contact the course staff, before proceeding further.

Assignment

Exercise 1: In the first step, you are expected to find buffer overflow from the provided web server. 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 `/project1/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 **five** different vulnerabilities*.

Note: we are only interested in vulnerabilities in the web server code (coded in C) and not in the web application (coded in Python and residing in the `zoobar` directory). We do not accept answers of vulnerabilities related to files inside the `zoobar` directory.

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.

Tips before excercise 2:

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 `/project1/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:

```
vagrant@INGI2347-vm:/project1$ ./clean-env.sh ./zookld
zook-exstack.conf &
[1] 2676
vagrant@INGI2347-vm:/project1$ ./exploit-template.py localhost 8080
HTTP request:
GET / HTTP/1.0
...
vagrant@INGI2347-vm:/project1$
```

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

```
vagrant@INGI2347-vm:/project1$ gdb -p $(pgrep zookfs-exstack)
...
0x40022424 in __kernel_vsyscall ()
(gdb) break your-breakpoint
Breakpoint 1 at 0x1234567: file http.c, line 9999.
(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`. We have added that command to `/home/vagrant/.gdbinit`, which will take effect if you start `gdb` in that directory.

Exercise 2: *Pick **two** buffer overflows* out of what you have found for later exercises (Note: 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:

```
vagrant@INGI2347-vm:/project1$ make check-crash
```

Tips before excercise 3:

In excercise 3, 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/vagrant/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.

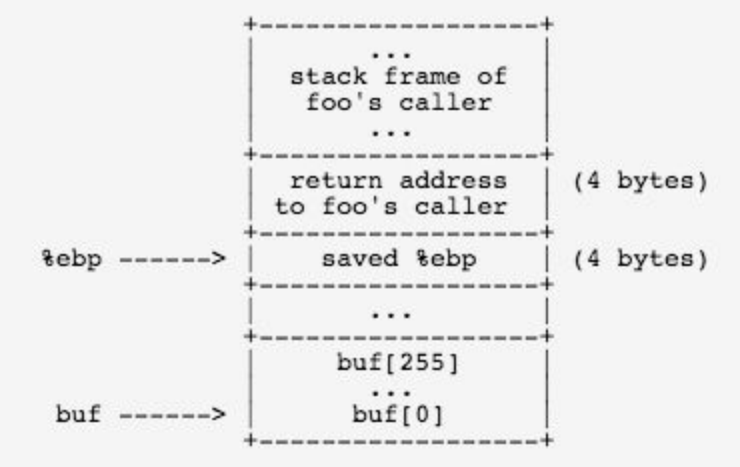
```
vagrant@INGI2347-vm:/project1$ ./clean-env.sh ./zookld
zook-exstack.conf
```

We have provided Aleph One's shell code for you to use in `/project1/shellcode.S`, along with Makefile rules that produce `/project1/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 execute: `unlink /home/vagrant/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:

```
vagrant@INGI2347-vm:/project1$ ./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]`:



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

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:

```
vagrant@INGI2347-vm:/project1$ 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 `/project1` directory, so that it picks up the set `follow-fork-mode child` command from `/project1/.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`.

Exercise 3: Now it's your turn to develop an exploit. Starting from one of your exploits from Exercise 2, construct an exploit that hijacks control flow of the web server and unlinks `/home/vagrant/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/vagrant/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 -- e.g. 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:

```
vagrant@INGI2347-vm:/project1$ 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.

Exercise 4: 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 `/project1/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 or preventing it. Note: you need not take care of the bugs in `zoobar`'s code.

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.

Exercise 5: Finally, you will explore fixing some of the vulnerabilities you have found in this project assignment. 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 runtime mechanisms such as stack canaries, removing `-fno-stack-protector`, etc.

Submission

You are done! Submit your answers to the project assignment by running `make prepare-submit` and upload the resulting `project1-handin.tgz` file to the INGInious web site: https://inginious.info.ucl.ac.be/course/INGI2347. Access to INGInious requires a valid INGI or UCL account.