Assignment 2
CS628A: Computer Systems Security
2018-19 - Semester II
Computer Science and Engineering Department
Indian Institute of Technology Kanpur
Due Date: Part A: 18th March 2019. 11:55pm
Part B: 25th March 2019. 11:55pm

Mohd Arsalaan Hameed

March 11, 2019

Acknowledgment: This project is based on a homework used at MIT for MIT-6.858 course in 2014

Introduction

This assignment will give you practical experience with common attacks and counter measures. You will be exploring attacks and counter measures in the context of the zoobar web application. This assignment consist of two parts.

Part A will introduce you to buffer overflow vulnerabilities, in the context of a web server called zookws. 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. Part B of the assignment will introduce you to privilege separation in zoobar web application.

This assignment may require you to learn a new programming language or some other piece of infrastructure. For example, in this project you must become familiar with certain aspects of the C language, x86 assembly language, gdb, etc. The assignment do so because that allows you to understand attacks and defenses in realistic situations. Often you need to understand certain parts of corner cases, and so you need to understand the details to craft exploits and design defenses for those corner cases. These two factors (new infrastructure and details) can make the projects time consuming. You should start early on the projects and work on them daily for some limited time (both part of assignment has several exercises), 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

Getting Started

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.

Once you have VirtualBox installed on your machine, you should download the course VM image, provided with this project assignment 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. The name of the VM is vm-cs628. The file containing the VM is provided along with this document via `https://web.cse.iitk.ac.in/users/spramod/courses/cs628-2019/hw/vm-628.ova`

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 and find the iNET address. For the rest of the document we assume this IP address is 192.168.134.128 for example. Each of you will get a different IP address of course.

Username	Password	Description
root	CS628A	You can use the root account to install new software packages into the VM, if you find something missing, using apt-get install pkg-name.
httpd	CyberSecurity	The httpd account is used to execute the web server, and contains the source code you will need for this assignment, in /home/httpd

The files you will need for this assignment is already preloaded in the VM as /home/httpd/lab (for part A), /home/httpd/lab2 (for part B). Proceed with this assignment, make sure you can compile the zookws web server as

```
httpd@vm-CS628:~$ cd lab
httpd@vm-CS628:~/lab$ 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
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-protect
cc -m32  run-shellcode.o  -lcrypto -o run-shellcode
rm shellcode.o
httpd@vm-CS628:~/lab$
```

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.

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

- **zookld, zookd-exstack, zookfs-exstack**, as configuredin 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 given a stack buffer overflow vulnerability. 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 project 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 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-CS628:~/lab$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 08:00:27:85:cc:0f
          inet addr:192.168.56.101  Bcast:192.168.56.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe85:cc0f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:925 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1108 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:287827 (287.8 KB)  TX bytes:156517 (156.5 KB)
httpd@vm-CS628:~/lab$ ./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 browser on your host machine and go to the URL http://192.168.56.101:8080/. If something doesn't seem to be working, try to figure out what went wrong, or contact the course TAs or the instructor before proceeding further.

Part - 1 Buffer overflow, Code injection

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/project/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/project/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-CS628:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf &
[1] 1960
httpd@vm-CS628:~/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. 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-CS628:~/lab$ gdb -p $(pgrep zookd-exstack)
-------
0x40022424 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 /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-CS628:~/lab$ make check-crash
```

For the next exercise, 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-CS628:~/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 **runshellcode** 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-628:~/lab$ ./run-shellcode shellcode.bin
$
```

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—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:

```
httpd@vm-CS628:~/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.

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@vm-CS628:~/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 (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 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.

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

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-CS628:~/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.

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 or preventing it. You should ignore bugs in zoobar's code. They will be addressed in future exercise. You should find at least two vulnerabilities for this exercise.

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.

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 runtime mechanisms such as stack canaries, removing -fno-stack-protector, baggy bounds checking, etc.

Run make prepare-submit. The resulting lab1-handin.tar.gz file will be graded. So submit it via moodle.

Part - B Privilege Separation

This part of the assignment will introduce you to privilege separation, in the context of a simple python web application called zoobar , where users transfer "zoobars" (credits) between each other. The main goal of privilege separation is to ensure that if an adversary compromises one part of an application, the adversary doesn't compromise the other parts too. To help you privilege-separate this application, the zookws web server used in the previous project is a clone of the OKWS web server, as discussed in the accompanying paper on OKWS. In this assignment, you will set up a privilege-separated web server, examine possible vulnerabilities, and break up the application code into less-privileged components to minimize the effects of any single vulnerability.

Zoobar web application also support executable profiles , which allow users to use Python code as their profiles. To make a profile, a user saves a Python program in their profile on their Zoobar home page. (To indicate that the profile contains Python code, the first line must be !python .) Whenever another user views the user's Python profile, the server will execute the Python code in that user's profile to generate the resulting profile output. This will allow users to implement a variety of features in their profiles, such as:

- A profile that greets visitors by their user name.
- A profile that keeps track of the last several visitors to that profile.
- A profile that gives a zoobar to every visitor (limit 1 per minute).

Supporting this safely requires sandboxing the profile code on the server, so that it cannot perform arbitrary operations or access arbitrary files. On the other hand, this code may need to keep track of persistent data in some files, or to access existing zoobar databases, to function properly. You will use the RPC library and some shim code (look up "shim code" on google) that is provided with this project to securely sandbox executable profiles.

For this part of the assignment the code is already present in /home/httpd/lab2 directory in the provided virtual box image file.

You may need to patch your python2.7 as well using the following

```
httpd@vm-CS628:~/lab2$ sudo make fix-flask
[sudo] password for httpd:
./fix-flask.sh
patching file /usr/lib/python2.7/dist-packages/werkzeug/routing.py
Done
httpd@vm-CS628:~/lab2$
```

Once your source code is in place, make sure that you can compile and install the web server and the zoobar application:

```
httpd@vm-CS628:~/lab2$ make
cc -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE   -c -o zookld.o zookld.c
cc -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE   -c -o http.o http.c
cc -m32  zookld.o http.o  -lcrypto -o zookld
cc -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE   -c -o zookfs.o zookfs.c
cc -m32  zookfs.o http.o  -lcrypto -o zookfs
cc -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE   -c -o zookd.o zookd.c
cc -m32  zookd.o http.o  -lcrypto -o zookd
httpd@vm-CS628:~/lab2$ sudo make setup
./chroot-setup.sh
+ grep -qv uid=0
+ id
------------------
------------------
+ python /jail/zoobar/zoodb.py init-person
+ python /jail/zoobar/zoodb.py init-transfer
httpd@vm-CS628:~/lab2$
```

What's a zoobar?

To understand the zoobar application itself, we will first examine the zoobar web application code.

One of the key features of the zoobar application is the ability to transfer credits between users. This feature is implemented by the script transfer.py . To get a sense what transfer does, start the zoobar Web site:

```
httpd@vm-CS628:~/lab2$ sudo make setup
./chroot-setup.sh
+ grep -qv uid=0
------------------------
------------------------
+ python /jail/zoobar/zoodb.py init-person
+ python /jail/zoobar/zoodb.py init-transfer
httpd@vm-CS628:~/lab2$ sudo ./zookld zook.conf
zookld: Listening on port 8080
zookld: Launching zookd
------------------------
```

Now, make sure you can run the web server, and access the web site from your host browser as http://192.168.56.101:8080/zoobar /index.cgi, You should see the zoobar web site

Exercise - 1 In your browser, connect to the zoobar Web site, and create two user accounts. Login in as one of the users, and transfer zoobars from one user to another by clicking on the transfer link and filling out the form. Play around with the other features too to get a feel for what it allows users to do. In short, a registered user can update his/her profile, transfer "zoobars" (credits) to another user, and look up the zoobar balance, profile, and transactions of other users in the system.

Read through the code of zoobar and see how transfer.py gets invoked when a user sends a transfer on the transfer page. A good place to start for this part of the lab is templates/transfer.html, _init_.py, transfer.py, and bank.py

Note: You don't need to turn in anything for this exercise, but make sure that you understand the structure of the zoobar application–it will save you time in the future!

Privilege separation

Having surveyed the zoobar application code, it is worth starting to think about how to apply privilege separation to the zookws and zoobar infrastructure so that bugs in the infrastructure don't allow an adversary, for example, to transfer zoobars to the adversary account.

The web server for this project uses the /jail directory to setup chroot jails for different parts of the web server. The **make** command compiles the web server, and **make setup** installs it with all the necessary permissions in the /jail directory.

As part of this assignment, you will need to change how the files and directories are installed, such as changing their owner or permissions. To do this, you should not change the permissions directly. Instead, you should edit the **chroot-setup.sh** script in the lab directory, and re-run **sudo make setup** . If you change the permissions in a different script, your server might not work.

Two aspects make privilege separation challenging in the real world and in this assignment. First, privilege separation requires that you take apart the application and split it up in separate pieces. Although we have tried to structure the application well so that it is easy to split, there are places where you must redesign certain parts to make privilege separation possible. Second, you must ensure that each piece runs with minimal privileges, which requires setting permissions precisely and configuring the pieces correctly. Hopefully, by the end of this project, you'll have a better understanding of why many applications have security vulnerabilities related to failure to properly separate privileges: proper privilege separation is hard!

One problem that you might run into is that it's tricky to debug a complex application that's composed of many pieces. To help you, we have provided a simple debug library in debug.py , which is imported by every Python script we give you. The debug library provides a single function, log(msg) , which prints the message msg to stderr (which should go to the terminal where you ran zookld), along with a stack trace of where the log function was called from.

If something doesn't seem to be working, try to figure out what went wrong before proceeding further.

As introduced in part A, the zookws web server is modeled after OKWS from the attached paper. zookws consists of a launcher daemon zookld that launches services configured in the file zook.conf , a dispatcher zookd that routes requests to corresponding services, as well as several services. For simplicity zookws does not implement helper or logger daemon.

The file zook.conf is the configuration file that specifies how each service should run. For example, the zookd entry:

```
[zookd]
    cmd = zookd
    uid = 0
    gid = 0
    dir = /jail
```

specifies that the command to run zookd is zookd , that it runs with user and group ID 0 (which is the superuser root), in the jail directory /jail .

The zook.conf file configures only one HTTP service, zookfs_svc , that serves static files and executes dynamic scripts. The zookfs_svc does so by invoking the executable zookfs , which should be jailed in the directory /jail by chroot . You can look into /jail ; it contains executables (except for zookld), supporting libraries, and the zoobar web site. See zook.conf and zookfs.c for details.

The launcher daemon zookld , which reads zook.conf and sets up all services is running under root and can bind to a privileged port like 80. Note that in the default configuration, zookd and vulnerable services are inappropriately running under root and that zookld doesn't jail processes yet; an attacker can exploit buffer overflows and cause damages to the server, e.g., unlink a specific file as you have done in Part A.

To fix the problem, you should run these services under unprivileged users rather than root. You will modify zookld.c and zook.conf to set up user IDs, group IDs, and chroot for each service. This will proceed in a few steps: first you will modify zookld.c to support chroot , second you will modify zookld.c to support user and group IDs other than root, and finally you will modify zook.conf to use this support.

Exercise - 2 Modify the function launch_svc in zookld.c so that it jails the process being created. launch_svc creates a new process for each entry in zook.conf , and then configures that process as specified in zook.conf . Your job is to insert the call to chroot in the specified place. You want to run man chroot to read the manual page about chroot . If you do this correctly, services won't be able to read files outside of the directory specified. For example, zookd shouldn't be able to read the real /etc/passwd anymore.

Run sudo make check to verify that your modified configuration passes our basic tests in check_lab2.py , but keep in mind that our tests are not exhaustive. You probably want to read over the cases before you start implementing.

Exercise - 3 Modify the function launch_svc in zookld.c so that it sets the user and group IDs and the supplementary group list specified in zook.conf . For example, you want to set in zook.conf the uid in zookd 's entry to, say, 61011, and have zookld.c change the user ID to 61011. You should do the same for group IDs. You will need to use the system calls setresuid , setresgid , and setgroups .

Think carefully about when your code can set the user ID. For example, can it go before setegid or chroot ?

This will also require you to modify chroot- setup.sh to ensure that the files on disk, such as the database, can be read only by the processes that should be able to read them. You can either use the built-in chmod and chown commands or our provided set_perms function, which you can invoke like so:

set_perms 1234:5678 755 /path/to/file

which will set the owner of the file to 1234, the group to 5678, and the permissions to 755 (i.e., user read/write/execute, group read/execute, other read/execute).

Run sudo make check to verify that your modified configuration passes our basic tests.

Now that none of the services are running as root, we will try to further privilege-separate the zookfs_svc service that handles both static files and dynamic scripts. Although it runs under an unprivileged user, some Python scripts could easily have security holes; a vulnerable Python script could be tricked into deleting important static files that the server is serving. Conversely, if the static service is compromised, it might read the databases used by the Python scripts, such as person.db and transfer.db . A better organization is to split zookfs_svc into two services, one for static files and the other for Python scripts, running under different users.

Exercise - 4 Create two new HTTP services (replacing zookfs_svc), along the lines of the existing zookfs_svc service, such that one will execute dynamic content, and one will serve static files. Modify the configuration file zook.conf to split the zookfs_svc service into two services running under different users: the static_svc service that only serves static files, and the dynamic_svc service that only executes the intended Python scripts (i.e., /zoobar/index.cgi).

Set file and directory permissions (using chroot-setup.sh) to ensure that the static service cannot read the database files from the dynamic service, that the dynamic service cannot modify static files, and that the dynamic service cannot be tricked into executing other scripts, such as the various .py programs under zoobar/ .

This separation requires zookd to determine which service should handle a particular request. You may use zookws 's URL filtering to do this, without modifying the application or the URLs that it uses. The URL filters are specified in zook.conf , and support regular expressions. For example, url = .* matches all requests, while url = /zoobar/(abc|def)\.html matches requests to /zoobar/abc.html and /zoobar/def.html .

Do not rely on URL filters for security; it is exceedingly difficult to do so correctly. For example, even if you configure the filter to pass only URLs matching .cgi to the dynamic service, an adversary can still invoke a hypothetical buggy /zoobar/foo.py script by issuing a request for /zoobar/foo.py/xx.cgi .

There are many executable files on the filesystem, and we cannot mark all of them non-executable for the zookfs service. For example, the zookfs service needs to execute /jail/usr/bin/python to run the zoobar website. We have added a feature to zookfs to only run trusted executables marked by a particular combination of owner user and group. To use this function, add an args = UID GID line to the service's configuration. For example, the following zook.conf entry:

```
[safe_svc]
    cmd = zookfs
    uid = 0
    gid = 0
    dir = /jail
    args = 123 456
```

specifies that safe_svc will only execute files owned by user ID 123 and group ID 456.

You should only be modifying configurations and permissions for this exercise in order to enforce privilege separation.

Run sudo make check to verify that your modified configuration passes our tests.

Interlude: RPC library

For the next part, you will privilege -separate the zoobar application itself in several processes. We would like to make sure we can deal with any future such bugs that come up. That is, if one piece of the zoobar application has an exploitable bug, then an attacker cannot use that bug to exploit other parts of the zoobar application. A challenge in spitting the zoobar application in several processes running with their own privileges is that the different processes must interact and have a way to communicate. You will first study a Remote Procedure Call (RPC) library that allows processes to communicate over a Unix socket. Then, you will use that library to separate the zoobar in several processes that communicate using RPC.

The RPC library itself shouldn't have any exploitable bugs, but historically parsing of messages has been a problem. We provide you with a functional RPC library, but think carefully about ways that an attacker could leverage the (typically complex) parsing code latent in most RPC libraries.

To illustrate how our RPC library might be used, we have implemented a simple "echo" service for you, in zoobar/echo-server.py . This service is invoked by zookld ; look for the echo_svc section of zook.conf to see how it is started.

echo- server.py is implemented by defining an RPC class EchoRpcServer that inherits from RpcServer , which in turn comes from zoobar/rpclib.py . The EchoRpcServer RPC class defines the methods that the server supports, and rpclib invokes those methods when a client sends a request. The server defines a simple method that echos the request from a client.

echo-server.py starts the server by calling run_sockpath_fork(sockpath) . This function listens on a UNIX-domain socket. The socket name comes from the argument, which in this case is /echosvc/sock (specified in zook.conf). When a client connects to this socket, the function forks the current process. One copy of the process receives messages and responds on the just-opened connection, while the other process listens for other clients that might open the socket

We have also included a simple client of this echo service as part of the Zoobar web application. In particular, if you go to the URL /zoobar/index.cgi/echo?s=hello , the request is routed to zoobar/echo.py . That code uses the RPC client (implemented by rpclib) to connect to the echo service at /echosvc/sock and invoke the echo operation. Once it receives the response from the echo service, it returns a web page containing the echoed response.

The RPC client-side code in rpclib is implemented by the call method of the RpcClient class. This methods formats the arguments into a string, writes the string on the connection to the server, and waits for a response (a string). On receiving the response, call parses the string, and returns the results to the caller.

Privilege-separating the login service in Zoobar

We will now use the RPC library to improve the security of the user passwords stored in the Zoobar web application. Right now, an adversary that exploits a vulnerability in any part of the Zoobar application can obtain all user passwords from the person database.

The first step towards protecting passwords will be to create a service that deals with user passwords and cookies, so that only that service can access them directly, and the rest of the Zoobar application cannot. In particular, we want to separate the code that deals with user authentication (i.e., passwords and tokens) from the rest of the application code. The current zoobar application stores everything about the user (their profile, their zoobar balance, and authentication info) in the Person table (see zoodb.py). We want to move the authentication info out of the Person table into a separate Cred table (Cred stands for Credentials), and move the code that accesses this authentication information (i.e., auth.py) into a separate service.

The reason for splitting the tables is that the tables are stored in the file system in zoobar/db/ , and are accessible to all Python code in Zoobar. This means that an attacker might be able to access and modify any of these tables, and we might never find out about the attack. However, once the authentication data is split out into its own database, we can set Unix file and directory permissions such that only the authentication service—and not the rest of Zoobar—can access that information.

Specifically, your job will be as follows:

- Decide what interface your authentication service should provide (i.e., what functions it will run for clients). Look at the code in login.py and auth.py , and decide what needs to run in the authentication service, and what can run in the client (i.e., be part of the rest of the zoobar code). Keep in mind that your goal is to protect both passwords and tokens. We have provided initial RPC stubs for the client in the file zoobar/auth_client.py .
- Create a new auth_svc service for user authentication, along the lines of echo-server.py . We have provided an initial file for you, zoobar/auth-server.py , which you should modify for this purpose. The implementation of this service should use the existing functions in auth.py .
- Modify zook.conf to start the auth-server appropriately (under a different UID).
- Split the user credentials (i.e., passwords and tokens) from the Person database into a separate Cred database, stored in /zoobar/db/cred . Don't keep any passwords or tokens in the old Person database.
- Modify chroot-setup.sh to set permissions on the cred database appropriately, and to create the socket for the auth service.
- login code in login.py to invoke your auth service instead of calling auth.py directly

Exercise - 5 Implement privilege separation for user authentication, as described above.

Don't forget to create a regular Person database entry for newly registered users.

Run sudo make check to verify that your privilege-separated authentication service passes our tests.

Now, we will further improve the security of passwords, by using hashing and salting. The current authentication code stores an exact copy of the user's password in the database. Thus, if an adversary somehow gains access to the cred.db file, all of the user passwords will be immediately compromised. Worse yet, if users have the same password on multiple sites, the adversary will be able to compromise users' accounts there too!

Hashing protects against this attack, by storing a hash of the user's password (i.e., the result of applying a hash function to the password), instead of the password itself. If the hash function is difficult to invert (i.e., is a cryptographically secure hashes), an adversary will not be able to directly obtain the user's password. However, a server can still check if a user supplied the correct password during login: it will just hash the user's password, and check if the resulting hash value is the same as was previously stored.

One weakness with hashing is that an adversary can build up a giant table (called a "rainbow table"), containing the hashes of all possible passwords. Then, if an adversary obtains someone's hashed password, the adversary can just look it up in its giant table, and obtain the original password.

To defeat the rainbow table attack, most systems use salting. With salting, instead of storing a hash of the password, the server stores a hash of the password concatenated with a randomly-generated string (called a salt). To check if the password is correct, the server concatenates the user -supplied password with the salt, and checks if the result matches the stored hash. Note that, to make this work, the server must store the salt value used to originally compute the salted hash! However, because of the salt, the adversary would now have to generate a separate rainbow table for every possible salt value. This greatly increases the amount of work the adversary has to perform in order to guess user passwords based on the hashes.

A final consideration is the choice of hash function. Most hash functions, such as MD5 and SHA1, are designed to be fast. This means that an adversary can try lots of passwords in a short period of time, which is not what we want! Instead, you should use a special hash-like function that is explicitly designed to be slow. A good example of such a hash function is PBKDF2, which stands for Password-Based Key Derivation Function (version 2).

Exercise - 6 Implement password hashing and salting in your authentication service. In particular, you will need to extend your Cred table to include a salt column; modify the registration code to choose a random salt, and to store a hash of the password together with the salt, instead of the password itself; and modify the login code to hash the supplied password together with the stored salt, and compare it with the stored hash. You can store the hashed password in the existing password column you have in the Cred table

To implement PBKDF2 hashing, you can use the Python PBKDF2 module. Roughly, you should import pbkdf2 , and then hash a password using pbkdf2.PBKDF2 (password, salt).hexread(32) . We have provided a copy of pbkdf2.py in the zoobar directory. Do not use the random.random function to generate a salt as the documentation of the random module states that it is not cryptographically secure. A secure alternative is the function os.urandom .

Run sudo make check to verify that your hashing and salting code passes our tests. Keep in mind that our tests are not exhaustive.

A surprising side-effect of using a very computationally expensive hash function like PBKDF2 is that an adversary can now use this to launch denial -of-service (DoS) attacks on the server's CPU. For example, the popular Django web framework recently posted a security advisory about this, pointing out that if an adversary tries to log in to some account by supplying a very large password (1MB in size), the server would spend an entire minute trying to compute PBKDF2 on that password. Django's solution is to limit supplied passwords to at most 4KB in size. For this project, we do not require you to handle such DoS attacks.

Submission Guidelines

1. Part A
   - Due date **18th March 2019, 11:55pm**
   - Run make prepare-submit in directory /home/httpd/lab, and submit lab1-handin.tar.gz on moodle.
2. Part B
   - Due date **25th March 2019, 11:55pm**
   - Run make prepare-submit in directory /home/httpd/lab2, and submit lab2-handin.tar.gz on moodle.