6.858 Spring 2017 Lab 2: Privilege separation and server-side sandboxing

Handed out:	Wednesday, February 22, 2017
Part 1 due:	Friday, March 3, 2017 (5:00pm)
Parts 2 and 3 due:	Friday, March 10, 2017 (5:00pm)
All parts due:	Friday, March 17, 2017 (5:00pm)

Introduction

This lab will introduce you to privilege separation and server-side sandboxing, 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 lab is a clone of the OKWS web server, discussed in lecture. In this lab, 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.

You will also extend the Zoobar web application to 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 that we provide to securely sandbox executable profiles.

To fetch the new source code, use Git to commit your Lab 1 solutions, and merge them into our lab2 branch:

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ git status
...
httpd@vm-6858:~/lab$ git add bugs.txt exploit-*.py [and any other new files...]
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab1'
[lab1 c54dd4d] my solution to lab1
```

```
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
...
httpd@vm-6858:~/lab$ git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch lab2 from origin.
Switched to a new branch 'lab2'
httpd@vm-6858:~/lab$ git merge master
Merge made by recursive.
...
httpd@vm-6858:~/lab$
```

In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code that is changed in the second lab assignment). In that case, the `git merge` command will tell you which files are *conflicted*, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with `git commit -a`.

You'll then need to patch flask in order to get it to work with the lab:

```
httpd@vm-6858:~/lab$ sudo make fix-flask
[sudo] password for httpd: 6858
./fix-flask.sh
patching file /usr/lib/python2.7/dist-packages/werkzeug/routing.py
Done
```

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

```
httpd@vm-6858:~/lab$ 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-6858:~/lab$ sudo make setup
[sudo] password for httpd: 6858
./chroot-setup.sh
+ grep -qv uid=0
+ id
...
+ python /jail/zoobar/zoodb.py init-person
+ python /jail/zoobar/zoodb.py init-transfer
httpd@vm-6858:~/lab$
```

Prelude: 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-6858:~/lab$ sudo make setup
[sudo] password for httpd: 6858
./chroot-setup.sh
+ grep -qv uid=0
+ id
...
+ python /jail/zoobar/zoodb.py init-person
+ python /jail/zoobar/zoodb.py init-transfer
httpd@vm-6858:~/lab$ 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 browser, as follows:

```
httpd@vm-6858:~/lab$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0c:29:57:90:a1
          inet addr:172.16.91.143  Bcast:172.16.91.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe57:90a1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:149 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:15235 (15.2 KB)  TX bytes:12801 (12.8 KB)
          Interrupt:19 Base address:0x2000
```

In this particular example, you would want to open your browser and go to `http://172.16.91.143:8080/zoobar/index.cgi/`, or, if you are using KVM, to `http://localhost: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` in the `zoobar` directory.

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 lab uses the `/jail` directory to setup `chroot` jails for different parts of the web server, much as in the OKWS paper. 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 lab, 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 lab. 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 lab, 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, or contact the course staff, before proceeding further.

Part 1: Privilege-separate the web server setup using Unix principals and permissions

As introduced in Lab 1, the `zookws` web server is modeled after OKWS from lecture 5. Similar to OKWS, `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 as OKWS does.

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 Lab 1.

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 2 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`. You will need to use the system calls `setresuid`, `setresgid`, and `setgroups`. Change the `zook.conf` uid and gid for zookd and zookfs so that they run as something other than root.

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

You will have 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).

Hint: look in `/tmp/zookld.out` for information about permission failures.

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, the static file serving code might be tricked into serving up 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 as different users.

Exercise 4. Modify `zook.conf` to replace `zookfs_svc` with two separate services, `dynamic_svc` and `static_svc`. Both should use `cmd = zookfs`. `dynamic_svc` should execute just `/zoobar/index.cgi` (which runs all the Python scripts), but should not serve any static files. `static_svc` should serve static files but not execute anything.

Run the dynamic and static services with different user and group IDs. Set file and directory permissions (using `chroot-setup.sh`) to ensure that the static service cannot read the database files, that the dynamic service cannot modify static files.

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`.

We have added a feature to `zookfs` to only run executables or scripts marked by a particular combination of owning 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 need this `args =` mechanism for the dynamic server to ensure that it only executes `index.cgi`, and not any of the other executables in the file system. You shouldn't rely on your `url =` pattern alone to limit what the dynamic server executes, because it's too hard to write the regular expressions properly. 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`.

You also need the `args =` mechanism for the static service, to prevent it from executing anything at all.

For this exercise, you should only modify configurations and permissions; don't modify any C or Python code.

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

Submit your answers to the first part of this lab assignment by running `make submit-a`. Alternatively, run `make prepare-submit-a` and upload the resulting `lab2a-handin.tar.gz` file to the submission web site.

Interlude: RPC library

In this part, you will privilege-separate the `zoobar` application itself in several processes. We would like to limit the damage from any future bugs that come up. That is, if one piece of the `zoobar` application has an exploitable bug, we'd like to prevent an attacker from using that bug to break into other parts of the `zoobar` application.

A challenge in splitting the `zoobar` application into several processes is that the processes must 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 `zoobar` into several processes that communicate using RPC.

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.

Part 2: 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.
- Modify the 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 hash), 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. Don't remove the password column

from the `Cred` table (the check for exercise 5 requires that it be present); you can store the hashed password in the existing `password` column.

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 lab, we do not require you to handle such DoS attacks.

Challenge 1! (optional) For extra credit, implement the honeywords proposal from Ari Juels and Ron Rivest in your authentication service. Consider implementing the honeychecker as a separate service running with its own user ID. If you decide to complete this challenge, please include a file named honeywords.txt in the top level of your lab submission that gives a brief overview of your approach and solution.

Part 3: Privilege-separating the bank in Zoobar

Finally, we want to protect the zoobar balance of each user from adversaries that might exploit some bug in the Zoobar application. Currently, if an adversary exploits a bug in the main Zoobar application, they can steal anyone else's zoobars, and this would not even show up in the `Transfer` database if we wanted to audit things later.

To improve the security of zoobar balances, our plan is similar to what you did above in the authentication service: split the `zoobar` balance information into a separate `Bank` database, and set up a `bank_svc` service, whose job it is to perform operations on the new `Bank` database and the existing `Transfer` database. As long as only the `bank_svc` service can modify the `Bank` and `Transfer` databases, bugs in the rest of the Zoobar application should not give an adversary the ability to modify zoobar balances, and will ensure that all transfers are correctly logged for future audits.

Exercise 7. Privilege-separate the bank logic into a separate `bank_svc` service, along the lines of the authentication service. Your service should implement the `transfer` and `balance` functions, which are currently implemented by `bank.py` and called from several places in the rest of the application code.

You will need to split the `zoobar` balance information into a separate `Bank` database (in `zoodb.py`); implement the bank server by modifying `bank-server.py`; add the bank service to `zook.conf`; modify `chroot-setup.sh` to create the new `Bank` database and the socket for the bank service, and to set permissions on both the new `Bank` and the existing `Transfer` databases accordingly; create client RPC stubs for invoking the bank service; and modify the rest of the application code to invoke the RPC stubs instead of calling `bank.py`'s functions directly.

Don't forget to handle the case of account creation, when the new user needs to get an initial 10 zoobars. This may require you to change the interface of the bank service.

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

Finally, we need to fix one more problem with the bank service. In particular, an adversary that can access the transfer service (i.e., can send it RPC requests) can perform transfers from *anyone's* account to their own. For example, it can steal 1 zoobar from any victim simply by issuing a `transfer(victim, adversary, 1)` RPC request. The problem is that the bank service has no idea who is invoking the `transfer` operation. Some RPC libraries provide authentication, but our RPC library is quite simple, so we have to add it explicitly.

To authenticate the caller of the `transfer` operation, we will require the caller to supply an extra `token` argument, which should be a valid token for the sender. The bank service should reject transfers if the token is invalid.

Exercise 8. Add authentication to the `transfer` RPC in the bank service. The current user's token is accessible as `g.user.token`. How should the bank validate the supplied token?

Although `make check` does not include an explicit test for this exercise, you should be able to check whether this feature is working or not by manually connecting to your transfer service and verifying that it is not possible to perform a transfer without supplying a valid token.

Submit your answers to parts 2 and 3 of this lab assignment by running `make submit-b`. Alternatively, run `make prepare-submit-b` and upload the resulting `lab2b-handin.tar.gz` file to the submission web site.

Part 4: Server-side sandboxing for executable profiles

You should familiarize yourself with the following new components of the lab source:

- First, the `profiles/` directory contains several executable profiles, which you will use as examples throughout this lab:
  - `profiles/hello-user.py` is a simple profile that prints back the name of the visitor when the profile code is executed, along with the current time.
  - `profiles/visit-tracker.py` keeps track of the last time that each visitor looked at the profile, and prints out the last visit time (if any).
  - `profiles/last-visits.py` records the last three visitors to the profile, and prints them out.
  - `profiles/xfer-tracker.py` prints out the last zoobar transfer between the profile owner and the visitor.
  - `profiles/granter.py` gives the visitor one zoobar. To make sure visitors can't quickly steal all zoobars from a user, this profile grants a zoobar only if the profile owner has some zoobars left, the visitor has less than 20 zoobars, and it has been at least a minute since the last time that visitor got a zoobar from this profile.

- Second, `zoobar/sandboxlib.py` is a Python module that implements sandboxing for untrusted Python profile code; see the `Sandbox` class, and the `run()` method which executes a specified function in the sandbox. The `run` method works by forking off a separate process and calling `setresuid` in the child process before executing the untrusted code, so that the untrusted code does not have any privileges. The parent process reads the output from the child process (i.e., the untrusted code) and returns this output to the caller of `run()`. If the child doesn't exit after a short timeout (5 seconds by default), the parent process kills the child.

  `Sandbox.run()` also uses `chroot` to restrict the untrusted code to a specific directory, passed as an argument to the `Sandbox` constructor. This allows the untrusted profile code to perform some limited file system access, but the creator of `Sandbox` gets to decide what directory is accessible to the profile code.

  `Sandbox` uses just one user ID for running untrusted profiles. This means that it's important that at most one profile be executing in the sandbox at a time. Otherwise, one sandboxed process could tamper with another sandboxed process, since they both have the same user ID! To enforce this guarantee, `Sandbox` uses a lockfile; whenever it tries to run a sandbox, it first locks the lockfile, and releases it only after the sandboxed process has exited. If two processes try to run some sandboxed code at the same time, only one will get the lockfile at a time. It's important that all users of `Sandbox` specify the same lockfile name if they use the same UID.

  How does `Sandbox` know that some sandboxed code has fully exited and it's safe to reuse the user ID to run a different user's profile? After all, the untrusted code could have forked off another process, and is waiting for some other profile to start running with the same user ID. To prevent this, `Sandbox` uses Unix's resource limits: it uses `setrlimit` to limit the number of processes with a given user ID, so that the sandboxed code simply cannot fork. This means that, after the parent process kills the child process (or notices that it has exited), it can safely conclude there are no remaining processes with that user ID.

- The final piece of code is `zoobar/profile-server.py`: an RPC server that accepts requests to run some user's profile code, and returns the output from executing that code.

This server uses `sandboxlib.py` to create a `Sandbox` and execute the profile code in it (via the `run_profile` function). `profile-server.py` also sets up an RPC server that allows the profile code to get access to things outside of the sandbox, such as the zoobar balances of different users. The `ProfileAPIServer` implements this interface; `profile-server.py` forks off a separate process to run the `ProfileAPIServer`, and also passes an RPC client connected to this server to the sandboxed profile code.

Because `profile-server.py` uses `sandboxlib.py`, which it turn needs to call `setresuid` to sandbox some process, the main `profile-server.py` process needs to run as root. As an aside, this is a somewhat ironic limitation of Unix mechanisms: if you want to improve your security by running untrusted code with a different user ID, you are forced to run some part of your code as root, which is a dangerous thing to do from a security perspective.

Python profiles with privilege separation

To get started, you will need to add `profile-server.py` to your `zook.conf` and modify `chroot-setup.sh` to create a directory for its socket, `/jail/profilesvc`. Remember that `profile-server.py` needs to run as root, so put 0 for the uid in its `zook.conf` entry.

Exercise 9. Add `profile-server.py` to your web server. Change the `uid` value in `ProfileServer.rpc_run()` from 0 to some other value compatible with your design from lab 2.

Make sure that your Zoobar site can support all of the five profiles. Depending on how you implemented privilege separation in lab 2, you may need to adjust how `ProfileAPIServer` implements `rpc_get_xfers` or `rpc_xfer`.

Run `sudo make check` to verify that your modified configuration passes our tests. The test case (see `check_lab2_part4.py`) creates some user accounts, stores one of the Python profiles in the profile of one user, has another user view that profile, and checks that the other user sees the right output.

If you run into problems from the `make check` tests, you can always check `/tmp/html.out` for the output html of the profiles. Similarly, you can also check the output of the server in `/tmp/zookld.out`. If there is an error in the server, they will usually display there.

The next problem we need to solve is that some of the user profiles store data in files; for example, see `last-visits.py` and `visit-tracker.py`. However, all of the user profiles currently run with access to the same files, because `ProfileServer.rpc_run()` sets `userdir` to `/tmp` and passes that as the directory to `Sandbox` (which it turn `chroot`s the profile code to that directory). As a result, one user's profile can corrupt the files stored by another user's profile.

Exercise 10. Modify `rpc_run` in `profile-server.py` so that each user's profile has access to its own files, and cannot tamper with the files of other user profiles.

Remember to consider the possibility of usernames with special characters. Also be sure to protect all of these files from other services on the same machine (such as the `zookfs` that serves static files).

Run `make check` to see whether your implementation passes our test cases.

Finally, recall that all of `profile-server.py` currently runs as root because it needs to create a sandbox. This is dangerous, and we would like to reduce the amount of code in `profile-server.py` that runs as root. In particular, the `ProfileAPIServer` that runs as part of `profile-server.py` does not strictly need to run as root (it does not invoke the sandbox), and in fact, it might be the most vulnerable part of the code to attacks, because it accepts RPC commands from the untrusted profile code!

Exercise 11. Change `ProfileAPIServer` in `profile-server.py` to avoid running as root. Recall that `profile-server.py` forks off a separate child process to run `ProfileAPIServer`, so you can switch to a different user ID (and group ID, if necessary) in `ProfileAPIServer.__init__`.

You will need to make sure that `rpc_xfer` can still perform transfers from the profile owner's account. It may be helpful to obtain the correct token before giving up root privileges.

As before, use `make check` to ensure your code passes our tests.

You are now done with the basic sandbox.

Challenge 2! (optional) Think of some interesting features that you could implement using Python server-side profiles, possibly in combination with extending the sandboxing infrastructure (e.g., providing an API for sending messages between users, or for sharing files between users). For example, can you build profile code that analyzes the social graph of who visited whose profile, or an equivalent to a Facebook wall, all using untrusted profile code?

Write a profile that demonstrates this functionality in `profiles/my-profile.py`. Describe what your profile is implementing in a comment at the top of the profile source code. Make any changes to your `ProfileAPIServer` necessary to support your feature.

Challenge 3! (optional) Now that profiles contain Python code, and can give away the user's zoobars, it's important that the user's profile code is not modified by an attacker, and only the correct profile code is executed by `profile-server.py`.

Create an RPC server that is in charge of modifying user profiles, and which requires a valid user token in order to modify a user's profile. Change the rest of the Zoobar application code to modify user profiles via this RPC server. Set

permissions on the profile database so that the rest of the Zoobar application cannot modify profiles directly. Change `profile-server.py` to read profile code directly from the profile database, instead of accepting it as input to the `run` RPC call.

`make check` only does a cursory inspection of the person db, so it may be that your solution is correct but the test fails, or that the test succeeds but your solution is wrong. Therefore, if you've completed the challenge and want us to grade it, add an empty file named `challenge3.txt` to the lab directory so we know to take a look at your solution.

You are done! Submit your answers to the lab assignment by running `make submit`. Alternatively, run `make prepare-submit` and upload the resulting `lab2-handin.tar.gz` file to the submission web site.

Acknowledgments

Thanks to Stanford's CS155 course staff for the initial zoobar web application code, which we extended in this lab assignment.