

System Security Lab 2

Alex W 1003474

Sheikh Salim 1003367

Exercise 1

```
scripts used to run and test server
```

```
sudo make all setup; sudo ./zookld zook.conf  
sudo make check
```

Glossary for lookup

```
zookws: webserver  
       no logger, no helper daemon  
zookld: launcher daemon  
zook.conf: configs for services  
         zookfs_svc: serve static file, execute dynamic script  
         zookfs  
zookd: routes requests to services  
  
zookld => zookfs_svc, echo_svc  
zookfs_svc => zookd => http_serve  
1. => http_serve_executable  
2. => http_serve_file  
3. => http_serve_directory  
    => looks for index.html/php/cgi in the directory, then call  
http_serve => http_serve_file/executable
```

Playground

Zoobar Foundation for Vigilant Research

[Log out alex](#)

Supporting the important advocates of the new world order

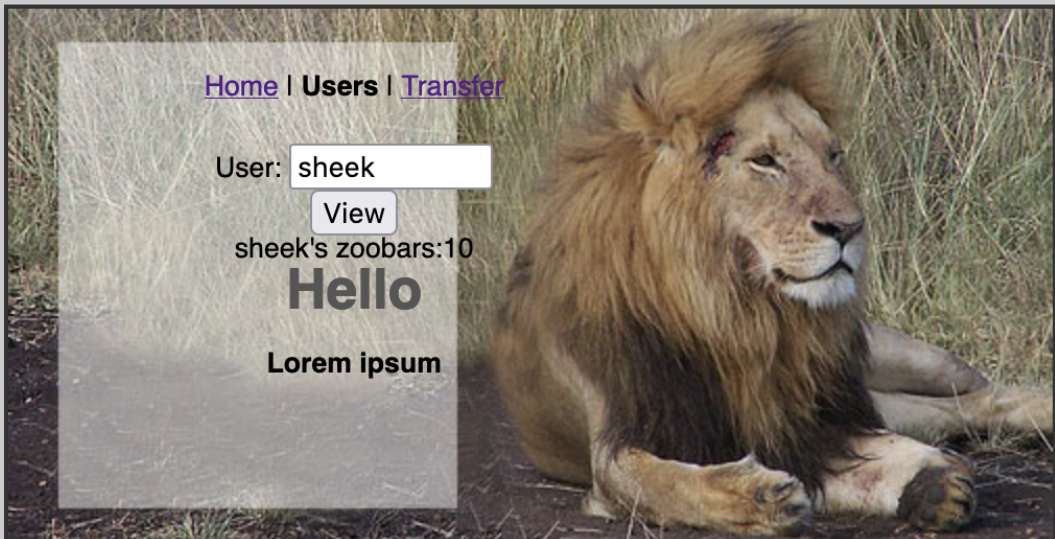
[Home](#) | [Users](#) | [Transfer](#)

User:

sheek's zoobars:10

Hello

Lorem ipsum



Time	Sender	Recipient	Amount
Mon Jun 28 03:00:50 2021	sheek	alex	2
Mon Jun 28 03:01:02 2021	alex	sheek	2

As shown in the screenshot, we created 2 accounts, sheek and alex. Based on the transaction history, sheek has sent alex 2 zoobars, and vice versa. Sheek has also updated his profile description with "Hello, Lorem ipsum", as viewed by alex's profile. alex also sees that sheek has 10 zoobars.

Understanding Serve Executable (Extra)

CGI

cgi (Common Gateway Interface) is used by the zook server to launch CGI scripts in a different process to process any request. The CGI process will process the request and return a HTML to the server, and the server relays that back to the client's browser. In this case, python scripts written in flask in zoobar folder are all defined as CGI services in `index.cgi`

Within the `index.cgi`, it loads up all method definitions in the directory. Only methods with a return html method return something to us

Flow is -> if user enters a method via `/zoobar/index.cgi/users` they will get to execute the users method in python.

- http will direct request to service 1
- service 1 will direct to index.cgi
- index.cgi will then call the python method

NON-CGI

If the admin does not want the method to run via cgi, need to specify in conf. This specification allows you to run methods directly like `/zoobar/echo-server.py?s=hello`

Exercise 2

In this exercise, we simply have to **chroot** into a directory that has been specified, which is **/jail**. The following code shows the method implemented:

```
if ((dir = NCONF_get_string(conf, name, "dir")))  
{  
    /* chroot into dir */  
  
    warnx("path %s", argv[0]);  
    warnx("dir %s", dir);  
    int result = chroot(dir);  
    if (result != 0)  
        warnx("Failed to chroot to jail");  
    chdir("/");  
}
```

Exercise 3

In this section, we need to set the correct UIDs and GIDs to the appropriate processes, as defined in the config. To do so, we need to maintain the following order

1. We first have to change the GIDs. This is so as we need to make sure that we are in root when changing the GIDs. Hence, **setgid** and **setgroups** have to be done prior
2. With the above done, only then can we set the UID. If we had done UIDs first, then we would no longer be in root, hence would have been unable to conduct the first step. This is so as we were utilising **setresuid**, which would have set all three **Real UID**, **Effective UID** and **Saved UID**. This has been done to prevent privilege re-escalation by the processes due to incomplete setting of IDs.

The following code shows how the order is adhered to

```
// First - SET the GID  
if (NCONF_get_number_e(conf, name, "gid", &gid))  
{  
    /* change real, effective, and saved gid to gid */  
    warnx("setgid %ld", gid);  
    setresgid(gid, gid, gid);  
}  
  
// Second - SET the additional GIDs  
  
if ((groups = NCONF_get_string(conf, name, "extra_gids")))  
{  
    ngids = 0;  
    CONF_parse_list(groups, ',', 1, &group_parse_cb, NULL);  
    /* set the grouplist to gids */  
    for (i = 0; i < ngids; i++)  
    {  
        warnx("extra gid %d", gids[i]);  
    }  
    setgroups(ngids, gids);  
}
```

```
}

// Finally we can set UID and expect to not be root by the end
if (NCONF_get_number_e(conf, name, "uid", &uid))
{
    /* change real, effective, and saved uid to uid */
    warnx("setuid %ld", uid);
    setresuid(uid, uid, uid);
}
```

Exercise 4

To separate the services, the template in `zook.conf` is used.

We updated the config as following:

```
[zook]
    port          = 8080
    http_svcs     = dynamic_svc, static_svc
    extra_svcs    = echo_svc

[zookd]
    cmd = zookd
    uid = 61011
    gid = 61011
    dir = /jail

[static_svc]
    cmd = zookfs
    url = .*(\.(html|css|js|jpg|ico)).*
    uid = 61009
    gid = 61009
    dir = /jail
    args = 61008 61008

[dynamic_svc]
    cmd = zookfs
    url = .*(\.cgi).*
    uid = 61012
    gid = 61012
    dir = /jail
    args = 61007 61007
```

```
chroot-setup.sh
```

```
set_perms 61007:61007 755 /jail/zoobar/index.cgi
```

Changed in made:

1. Separating zookfs_svc to static_svc, dynamic_svc
2. Ensure they have different uid, so we assigned static_svc to uid to 61011, dynamic_svc to 61009
3. Register the http_svc to zook, with static_svc first, then dynamic_svc
4. Set up url filters using regex
 1. for static_svc, we need to filter for all static files, eg html, css, js, jpg, ico
 2. for dynamic_svc, we need to filter for all dynamic files, hence, only .cgi.
5. Using regex101.com, we confirm that our filter works

```
:/.*(\.(html|css|js|jpg|ico)).*
```

TEST STRING

```
/zobar/index.cgi/login↵
```

```
/zobar/media/zobar.css↵
```

```
/zobar/media/lion_sleeping.jpg
```

REGULAR EXPRESSION

```
:/.*(\.(cgi)).*
```

TEST STRING

```
/zobar/index.cgi/login↵
```

```
/zobar/media/zobar.css↵
```

```
/zobar/media/lion_sleeping.jpg
```

6. To ensure that .cgi runs with a separate permission from main svc process, in dynamic_svc, we set its uid and gid to 61008, and change the corresponding permission of index.cgi to be executable for 61008

only, to adhere to the security design that only 61008 is able to execute index.cgi, and not any user issued cgi.

Exercise 5

Flow of authentication mechanism (previously)

auth.py The stock code does the following:

1. Whenever the user registers or login, a token is generated by **auth.py**. The token is generated with a simple md5 hash of the user's password and a **random** float. This will happen, like most functions, in a single DB session and pushed into the DB. Hence we can expect the token to change everytime this method is called
2. In **login**, the method of authentication is to simply string compare the password with the plaintext password stored in the **person** DB. If match -> generate a token.
3. In **register**, the user is made first, followed by the generation of the token.
4. In **check_token**, the username and token is checked with the respective content of the DB. If matches, return true

login.py

1. The cookie is stored as **username#token** under the cookie name of **PyZoobarLogin**. **checkCookie()** function will check the existence of this cookie, and **loginCookie** will create the cookie payload.
2. When the user logs in, a **User** object is essentially instantiated, which tallies with the current user session. This is done by the method **setPerson**. Logout will clear this instantiation by setting **self.person = None**

The new **auth** Database

In this section, we will first separate out sensitive information from the **Person** database into the **Cred** database. **Cred** will be used purely for authentication henceforth.

```
class Person(PersonBase):
    __tablename__ = "person"
    username = Column(String(128), primary_key=True)
    zoobars = Column(Integer, nullable=False, default=10)
    profile = Column(String(5000), nullable=False, default="")

class Cred(CredBase):
    __tablename__ = "cred"
    username = Column(String(128), primary_key=True)
    password = Column(String(128))
    token = Column(String(128))
```

Because of these changes, all calls for a token within **auth.py** will now be swapped from **Person.token** to **Cred.token**.

Changes to `auth.py`

Given the separation of `Cred` and `Person`, a refactor was also made accordingly. The `register` method, which had previously only interacted with the `Person` db will now have to interact with both databases. In this case, storing of the password hash will make a write operation to the `Cred` database, while storing of the newly created user profile will make a write operation to the `Person` database.

Do refer to `auth.py` for full information.

Setting up GRPC auth client/server

In this new setup, `auth.py` will now provide privileged functions, accessible only via the grpc auth server. Below shows the simple methods of implementing, essentially calling from the `auth.py` module:

```
def rpc_login(self, username, password):
    return auth.login(username, password)

def rpc_register(self, username, password):
    return auth.register(username, password)

def rpc_check_token(self, username, token):
    return auth.check_token(username, password)
```

Lastly, we will set up the GRPC caller (or client). In this case, we simply have to dial in to the socket with the corresponding GRPC method names for each of the methods described as such (in `auth_client.py`):

```
def login(username, password):
    with rpclib.client_connect('/authsvc/sock') as c:
        return c.call('login', username=username, password=password)

def register(username, password):
    with rpclib.client_connect('/authsvc/sock') as c:
        return c.call('register', username=username, password=password)

def check_token(username, token):
    with rpclib.client_connect('/authsvc/sock') as c:
        return c.call('check_token', username=username, token=token)
```

Following the setup of the GRPC methods, we then proceed to swap out all `auth` module calls in `login.py` to the `auth_client.py` GRPC client methods.

Permissioning auth services

First, we setup the GRPC server via `zook.conf` as follows:

```
[auth_svc]
cmd = /zoobar/auth-server.py
```

```
args = /authsvc/sock # Note that we give it the following socket
dir = /jail
uid = 61013
gid = 61012
```

We will also set up the necessary permissions in `chroot-setup.sh` as follows:

```
# For the authsvc server socket
create_socket_dir /jail/authsvc 61013:61012 755
....
python /jail/zoobar/zodb.py init-cred

....
# For part 5 -- Auth service
set_perms 61013:61012 700 /jail/zoobar/db/cred
set_perms 61013:61012 600 /jail/zoobar/db/cred/cred.db
set_perms 61013:61012 700 /jail/zoobar/auth-server.py

set_perms 61012:61012 770 /jail/zoobar/db/person
set_perms 61012:61012 660 /jail/zoobar/db/person/person.db
```

Rationale behind Permissions

As previously mentioned, we note that `auth.py` utilises 2 different databases - the `Cred` and `Person` database for the `register` endpoint. Hence, minimally,

- `auth` service has to have read and write access to both `cred.db` and `person.db`. In this spirit, it was essential to provide a common group, `61012` between the dynamic service and the auth service, such that `auth` service will have secure access to interact with the databases. We also instituted `r+w(6)` permission, as there is no need for the services to have executable rights to the database
- `auth-server.py` was also provided with a strict `700` policy, where only the rightful uid should be able to run as the server. We saw that there is no need for other services to be able to run the server, since it would have nullified our efforts at privilege separation
- The `/jail/authsvc` socket however has been given a `755` permission to allow interaction with the grpc server.

Exercise 6

In this exercise, we will utilise a more secure method of authentication via password hashing and salting.

Changes to table schema

```
class Cred(CredBase):
    __tablename__ = "cred"
    username = Column(String(128), primary_key=True)
    password = Column(String(128)) # EX6: Swapped to hold hash, under same
col
```



```
token = Column(String(128))
salt = Column(String(128)) # Added for EX6
```

Encoding

We note that PBKDF2 only accepts either a string or unicode type for salt. Given that `os.urandom()` returns a `byte array`, we hence used `base64` to encode it into a string (`binary -> ascii`) that can then be stored in the database and simultaneously used for the `PBKDF2` function.

The below function shows how salt is being initialised into the PBKDF function as described above.

```
def _setup(self, passphrase, salt, iterations, prf):
    # Sanity checks:
    # passphrase and salt must be str or unicode (in the latter
    # case, we convert to UTF-8)
    ...
    if isinstance(salt, str):
        salt = salt.encode("UTF-8")
    elif not isinstance(salt, bytes):
        raise TypeError("salt must be str or unicode")
```

Changing `auth.py` for salting features

In this section, we would have to modify the following functions:

1. `register` -> register now has to take in the user's password and attempt to salt + hash the password with a randomly generated salt via PBKDF2. The salt, passwordHash into the database. Below are the modifications made:

```
def register(username, password):
    salt = os.urandom(8).encode('base-64')
    password_hash = pbkdf2.PBKDF2(password, salt).hexread(32)
    newcred = Cred()
    newcred.password = password_hash
    newcred.username = username
    newcred.salt = salt
    ...
    # proceed to commit accordingly
```

2. `login` -> login has to now take in the user's password, and attempt to salt + hash the password given with the salt in the `cred` database for that user. If this result tallies with the passwordHash entry in the database, user credentials are legitimate. This is done using PBKFD2 as below:

```
def login(username, password):
    db = cred_setup()
```

```

cred = db.query(Cred).get(username)

if not cred:
    return None
# Modified for Task 6
if cred.password == pbkdf2.PBKDF2(password, cred.salt).hexread(32):
    return newtoken(db, cred)
else:
    return None

```

Exercise 7

This exercise aims to separate the bank functions by privilege separating it into a new process, with rpc capabilities. As a reminder, bank functions are called primarily from:

1. `user.py`
2. `transfer.py`
3. `login.py`

The new `bank.db`

```

class Bank(BankBase):
    __tablename__ = "bank"
    username = Column(String(128), primary_key=True)
    zoobars = Column(Integer, nullable=False, default=10)

```

We will also be removing the `zoobars` key from the `Person` database, hence making it only accessible by this new `Bank` database

With this changes in place, we will also replace all calls to `Person` from `bank.py` to match this new implementation. The following code shows the changes being made, where `Bank` database is now being used instead for the `transfer()` method:

```

def transfer(sender, recipient, zoobars):
    # Exercise 7 - Swapped to bankDB
    bankdb = bank_setup()
    senderp = bankdb.query(Bank).get(sender)
    recipientp = bankdb.query(Bank).get(recipient)

    sender_balance = senderp.zoobars - zoobars
    recipient_balance = recipientp.zoobars + zoobars

    if sender_balance < 0 or recipient_balance < 0:
        raise ValueError()
    ...

def balance(username):
    db = bank_setup()

```

```
# person balance queried from bank instead
person = db.query(Bank).get(username)
return person.zoobars
```

Do refer to [bank.py](#) for full information.

Setting up GRPC auth client/server

Similar to the setup in [auth](#) service, we created client and server functions for interactions with [bank](#) service. Due to the repetitive nature of this part, please refer to [bank_client.py](#) and [bank-server](#) for more information.

Hurdle 1: Parsing datatypes

One of the problems with incorporating RPC in this part was an error we encountered when trying to make the [get_log](#) function into an rpc method. This is so as SQLAlchemy returns a Query object that is intrinsically incompatible with the rpc library utilised.

```
raise TypeError(repr(o) + " is not JSON serializable")
TypeError: <sqlalchemy.orm.query.Query object at 0x7fbd60263a10> is not JSON serializable
```

```
TypeError: <sqlalchemy.orm.query.Query object at 0x7fccd47d5490> is not
JSON serializable
```

Thus, we had to implement a serialise function, such that it can be parsed as a string, with accordance to the handout description. Credits to [this](#) and [this](#) stackoverflow answers:

```
def serialise_msg(sql_obj):
    return {c.name: getattr(sql_obj, c.name) for c in
sql_obj.__mapper__.columns}

...

def rpc_get_log(self, username):
    res = bank.get_log(username)
    return [serialise_msg(z) for z in res]
```

Handling of account creation

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.

One of the challenges in this section is the initialisation of zoobars. Previously, zoobars were initialised within the [auth.py](#) package, given that the [Person](#) database is accessible by the service.

Given now that we want to segment out and **privilege separate** the bank database, we hence need to create a function within the bank interface to handle the initialisation of 10 zoobars for each newly created account. The following is a simple function implemented:

```
def initialise_zoobars(username):
    db = bank_setup()
    person = db.query(Bank).get(username)
    # Confirm person exists
    if person:
        return None
    newbankentry = Bank()
    newbankentry.username = username
    # no need to initialise zoobars, SQL handle by default
    db.add(newbankentry)
    db.commit()
```

Following the creation of the service, we then proceed to call this function, via rpc, from the main process executing in `login.py` under the `addRegistration` function as follows:

```
def addRegistration(self, username, password):

    token = auth_client.register(username, password)
    if token is not None:
        #Exercise 7 - Lastly proceed to make a call to initialise
zoobars
        bank_client.initialise_zoobars(username)
        return self.loginCookie(username, token)
    else:
        return None
```

Managing Bank Permissions

```
# zook.conf
[bank_svc]
    cmd = /zoobar/bank-server.py
    args = /banksvc/sock
    dir = /jail
    uid = 61014
    gid = 61014
```

```
# chroot-setup.sh
create_socket_dir /jail/banksvc 61014:61012 755
...
```

```
set_perms 61014:61014 700 /jail/zoobar/db/bank
set_perms 61014:61014 600 /jail/zoobar/db/bank/bank.db
set_perms 61014:61014 700 /jail/zoobar/db/transfer
set_perms 61014:61014 600 /jail/zoobar/db/transfer/transfer.db
```

Rationale behind Permissions

From the above, we see that the bank service operates as a silo on its own, with no shared GIDs (having both uid and gid of **61014**, not shared with anyone). This is so as we saw that the bank service did not require any additional permissions to other files owned by other services, thus conforming easily to the principle of least privilege.

In the **bank.py** file, we note that bank traditionally only has access to the **bank.db** and **transfer.db** only. This makes it fairly easy to manage permissions as below:

- Only bank service (**61014**) should have any access to the **bank.db** and **transfer.db**. As this is not a shared database, we can restrict it to have permissions of **600**, only read and writable by the bank service. We do have to provide permissions of **700** for the directory of the databases, similar to exercise 5.
- **bank-server.py** was also provided with a strict **700** policy, where only the rightful bank service uid should be able to run the server.
- The **/jail/banksvc** socket however has been given a **755** permission to allow interaction with the grpc server.

Exercise 8

For this step, we will simply swap out all the code to now use **token**. The main swap will be at **transfer.py**, which now calls the transfer rpc call with **g.user.token**

Testing the feature

We then add the authentication procedure as such, essentially having the bank making a rpc call to the auth service to verify the validity of the token. If valid, we call transfer as per normal:

```
def rpc_transfer(self, sender, recipient, zoobars, token):
    # Exercise 8 - Authenticate the request via token
    if not auth_client.check_token(sender, token):
        log("Transfer authentication failed")
        raise ValueError('Token is invalid')

    return bank.transfer(sender, recipient, zoobars)
```

In order to test the authentication check, we created a scenario as such:

```
try:
    if 'recipient' in request.form:
        zoobars = eval(request.form['zoobars'])
        bank_client.transfer(g.user.person.username,
                             request.form['recipient'], zoobars, 'fake-token')
        warning = "Sent %d zoobars" % zoobars
except (KeyError, ValueError, AttributeError) as e:
    traceback.print_exc()
    warning = "Transfer to %s failed" % request.form['recipient']
```

- Instead of sending the token, we now send a string "fake-token" that should trigger an exception

Testing the feature on browser

With valid token

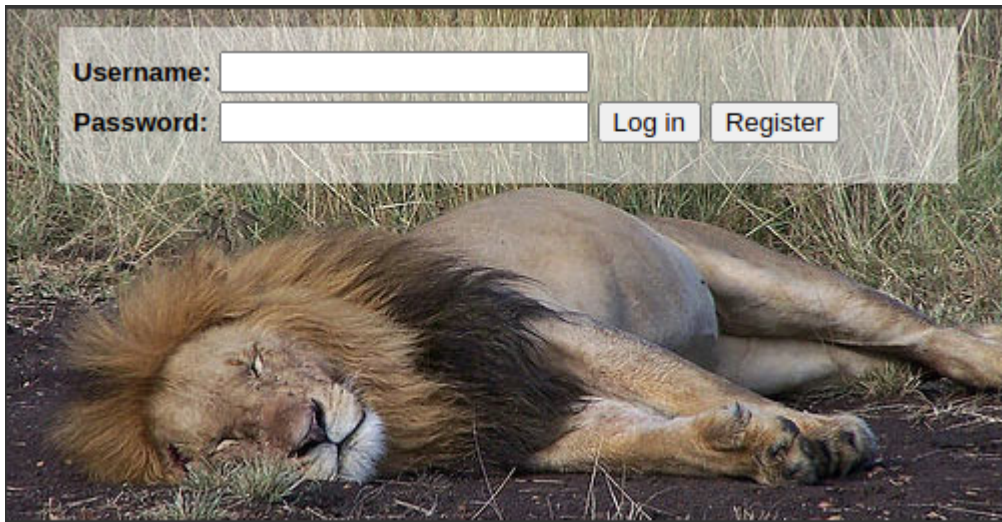
Name	Value
PyZoobarLogin	sheek#f37e08eafbdf8da04d3...



With invalid token

Name	Value
PyZoobarLogin	sheek#f37e08eafc df 8da04d3...

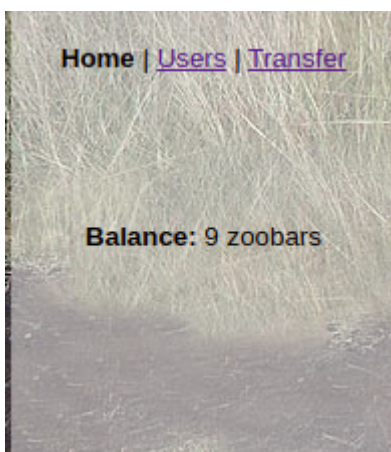
We note that with a modified token, the request failed and redirects us back to the login page. The transfer is also noted to not have occurred, as balance remains at 9 Zoobars.



Name	×	Headers	Preview	Response	Initiator	Timing	Cookies
transfer							
login?nexturl=http://localhost:8080/zo...							

Failed to load response data

2 / 5 requests | 1.5 kB / 106 kB transferred



Exercise 9

To support profiles, we added

1. Profile service to zook.conf:

```
[profile_svc]
  cmd = /zoobar/profile-server.py
  args = /profilesvc/sock
```

```
uid = 0
gid = 61012
dir = /jail
```

- Note the uid is set to 0 as it has to run as root to spawn and set uid for its child processes

2. Create necessary sock with the right permission

```
create_socket_dir /jail/profilesvc 61006:61006 755
```

3. Set profile-server.py with the right permission

```
set_perms 61006:61006 700 /jail/zoobar/profile-server.py
```

- this allows sandboxlib to run profile-server.py as a sandboxed process, with uid **61006** updated in `ProfileServer.rpc_run()`, with the correct permission to access the socket and the python source code.
- 700 is used as only uid **61006** needs access to the python script, following the principle of least privileges.

Exercise 10

```
user_uuid = uuid.uuid3(uuid.NAMESPACE_OID, user.__str__().__str__())
user_path = os.path.join(userdir, user_uuid)
if not os.path.exists(user_path):
    os.mkdir(user_path)
    os.chmod(user_path, stat.S_IRWXU ^ stat.S_IRWXG)
```

The code above is used to generate uuid for each user, based on their username string. This handles any special characters there may be in their username. An example folder would be **813a9bfb-1689-3a5e-923b-1a8fc97e92f1**.

After creating the directory for each user, permission is set. `stat.S_IRWXU`, `stat.S_IRWXG` stand for user read, write, execute and group read, write, execute respectively. This prevents one user from accessing and tempering the file of another user.

Exercise 11

To ensure ProfileAPIServer does not execute with root privileges, we set the uid to **61014**, and gid to **61012**, corresponding to bank services to ensure that the profile server is able to make transactions, while not having any unneeded privileges. gid has to be set before uid as it is not possible to change gid when uid is not **0**.


```
os.setgid(61012)
os.setuid(61014)
```

sudo make check

```
sudo make check
./check_lab2.py
+ setting up environment in fresh /jail..
+ running make.. output in /tmp/make.out
+ running zookld in the background.. output in /tmp/zookld.out
PASS App functionality
PASS Exercise 2
PASS Exercise 3
PASS Exercise 4
PASS Exercise 5
PASS Exercise 6
PASS Exercise 7
+ restoring /jail; test /jail saved to /jail.check..
./check_lab2_part4.py
+ setting up environment in fresh /jail..
+ running make.. output in /tmp/make.out
+ running zookld in the background.. output in /tmp/zookld.out
+ profile output logged in /tmp/html.out
PASS App functionality
PASS Profile hello-user.py
PASS Profile visit-tracker.py
PASS Profile last-visits.py
PASS Profile xfer-tracker.py
PASS Profile granter.py
PASS Exercise 10: /testfile check
PASS Exercise 11: ProfileAPIServer uid
+ restoring /jail; test /jail saved to /jail.check..
```