PlaygroundBankQuickstart

Jump to bottom

sethnielson edited this page 16 days ago · 1 revision

This quickstart is intended to get you up and running with playground *quickly*. You won't understand everything, but you will have a working system in a very short amount of time. As you follow the quick start, make mental notes of the things you want to understand better and see if the additional documentation helps you figure it out.

Playground Quickstart

Playground is an *overlay* network. An overlay network is a network that operates on top of another network with its own addressing and semantic schemes. For example, email is an overlay network. It has its own addressing (e.g., `sethjn@cs.jhu.edu`) and its own semantic connections (I'm connected to everyone in my address book). The fact that it *usually* runs over the IP network is irrelevant. When you're thinking about sending someone an email, do you think about the IP addresses involved? No! Of course not! And the addressing means that your message can get to your contact regardless what system they're using. In fact, email could be delivered without IP addresses at all.

Similarly, we are going to create a Playground network that has its own addressing and its own semantic connections. In this quick start, the entire network will run on your own personal computer. But later, we will build a network that connects the entire class. The playground network runs on top of TCP/IP, but that *doesn't matter*! Ignore that. Try to focus on the overlay network we're setting up here as much as possible. Occasionally, you do have to think about the TCP/IP pieces (just as email does), but it's minimal.

The first thing to do is get Playground installed.

Installing Playground

The current major version of Playground is Playground 3.0 (Playground3). It can be cloned out of github from https://github.com/CrimsonVista/Playground3. You should clone it to your local computer, even though you will typically not need to modify it.

Once you have it cloned, you can use pip (a python installation manager) to install it for your python to use. Please note, *I HIGHLY recommend* using python virtual environments as shown below (the current directory is shown as the prompt... your individual prompt will look different and your individual directories can be different):

```
[~] > mkdir netsec_env
[~] > cd netsec_env
[~/netsec_env] > python3 -m venv .
[~/netsec_env] > source bin/activate
[~/netsec_env] (netsec_env) > git clone https://github.com/CrimsonVista/Playground3
[~/netsec_env] (netsec_env) > pip install -e ./Playground3
```

Note, some versions of pip have an error about `bdist_wheel`. If you get this error, you may need to install `pip install wheel`.

Playground is now installed. There are two major components:

1. The underlying virtual hardware layer
2. The API to access the network from a Python program

We'll start by configuring some hardware

Playground Hardware

All playground networking controls are accessible through the `pnetworking` command. Once you've installed playground, you should have access to this utility. If not, make sure that you're in the right virtual environment (you will need to call `source bin/activate` from the directory you configured... read up on virtual environments if you need more info).

`pnetworking` has a number of subcommands. We won't explore them all today. It also makes use of a config file. For simplicity, we'll start by configuring a config file for your user name. Later, we'll be able to have more than one configuration, which is useful for connecting to different networks.

To initialize your playground network, execute the following command

```
[~/netsec_env] (netsec_env) > pnetworking initialize local
```

You should get a message that this succeeded, but you can always check by running the status command:

```
[~/netsec_env] (netsec_env) > pnetworking status

Configuration: /home/sethjn/.playground

Playground Networking Configuration:
-------------------------------------
```

The configuration is currently blank because we have created no hardware.

To get communications going in playground, we need two virtual pieces of hardware: a switch and a nic (network interface card). In networking, a switch connects multiple devices together and a nic is how your device interacts with the switch.

We're going to have one switch and one nic. You might ask yourself, how will we have networking if we only have one nic? Don't we need at least two nic's? Sure. But just like you can network programs together on your computer, you can network playground programs together on your computer as well. It's all local for now.

To create a switch, execute the following command:

```
pnetworking add my_switch switch
```

The `add` command takes at least two arguments: the name of the device and the type. So this command created a device called `my_switch` that is of type `switch` . You can give it other arguments to specify exactly what IP address and port it should be using, but this is all we need for our initial tests.

if you do a `pnetworking status` now, you should see something like this:

```
Configuration: /home/sethjn/WIN_DEV/stage/20191/playground/.playground

Playground Networking Configuration:
-------------------------------------

switch my_switch:
        Status: Disabled
        Auto-Enable: True
        PNMS Managed: True
        TCP Location: 127.0.0.1:63902
```

Most of this doesn't matter. The `Auto-Enable` and the `PNMS Managed` have to do with some auto-configuration stuff. The only thing worth noting is perhaps:

1. It is currently disabled
2. It is using a TCP/IP location

Let's talk about the TCP/IP first. As I said earlier, Playground creates an overlay network on top of TCP/IP. The virtual hardware pieces connect to each other over TCP/IP. But we will be writing playground programs that use the virtual hardware with playground addresses. We will *think* in terms of playground even though underneath, the actual network traffic goes over TCP/IP. Again, think of email. Email has to be sent over TCP/IP, but you don't think about it that way. You just say, "I want to send an email from me (sethjn@cs.jhu.edu) to a student (somestudent@jhu.edu)". You have a from and a to address and you let however it actually gets there be hidden and unseen.

Similarly, we will have playground addresses and playground ports. THESE ARE NOT TCP addresses and ports. I can only say this so many times, but every year students confuse the two. Please understand this. I'll say it one more time:

Playground Addresses and Ports are not TCP Addresses and ports

So what is a playground address and port? Let's start with the address. Playground addresses are any four numbers "dotted" together.

```
0.0.0.0
100.100.100.100
9999999.9999999.9999999.9999999
1.2.3.4
192.168.1.1
```

All of the above examples are legal playground addresses. We will typically not use addresses that look like IP addresses to avoid confusion. In fact, for our purposes, we will have our first number always be `20191` which represents the year plus the term. In other words, most of the "official" addresses you will see are:

```
20191.x.y.z
```

of course, this is just convention. There is nothing stopping you (or a classmate) from using something else. But, for example, we will allocate addresses for each team using this scheme. The class servers (such as the bank) will use this prefix as well.

But what *is* an address? A playground address is used by a nic to tell the switch what traffic it is interested in. The virtual nic sends messages with to addresses and from addresses. The switch takes the incoming data, looks at the destination address, and routes it to any device (or to any devices) that are claiming that address.

When you create a VNIC (virtual nic) device, you assign it a playground address.

```
pnetworking add my_nic vnic 20191.10.20.30
```

This adds a virtual nic to the hardware configuration that will claim `20191.10.20.30` as its address when it connects to the switch.

In addition to creating a vnic, we have to make two other configurations. First, we have to tell it what switch to connect to, and we have to tell it what playground addresses it routes for. This second piece is in case we have *two* (or more) nics at the same time. They can claim different outbound traffic. But when we just have one, we set it as the "default" route.

```
pnetworking config my_nic connect my_switch
pnetworking config my_nic route add default
```

Run `pnetworking status` again, and you should see both devices configured but disabled. Let's turn them on.

```
pnetworking on
pnetworking status

Configuration: /home/sethjn/.playground

Playground Networking Configuration:
-------------------------------------

switch my_switch:
        Status: Enabled (PID: 1186)
        Auto-Enable: True
        PNMS Managed: True
        TCP Location: 127.0.0.1:63902

vnic my_nic:
        Status: Enabled (PID: 1193)
        Auto-Enable: True
        Playground Address: 20191.10.20.30
        Connected To: my_switch (127.0.0.1:63902) (Live)
        Routes:
                __default__
```

This status tells you that we have a switch running and a vnic connected to it claiming the address `20191.10.20.30` . That's good! We now have a working playground networking!

As hinted at earlier, there are API's for writing programs that communicate over the Playground network instead of TCP/IP. We won't cover that here. Instead, we have a bank application for you to play with.

Bank Quickstart

The Playground Bank is a client/server application with a fair bit of functionality. But with default parameters, it is easy to install, configure, and use.

Before Cloning, install the required dependencies OUTSIDE the virtual environment.

```
[~/netsec_env] > sudo apt-get install build-essential libssl-dev libffi-dev python-dev

[~/netsec_env] > sudo apt install python3-pip

[~/netsec_env] > pip3 install cryptography
```

Reactivate your virtual environment.

First, clone the project from GitHub.

```
[~/netsec_env] (netsec_env) > git clone https://github.com/CrimsonVista/BitPoints-Bank-
Playground3
```

Unfortunately, the bank doesn't have much of an "installer" yet. So, for now, you'll have to run it directly out of its source directory.

```
[~/netsec_env] (netsec_env) > cd BitPoints-Bank-Playground3/src
```

The first thing we want to do is setup a server. The initialization process is now automated, but it requires some input from you along the way. You will need to create a password for the bank and the mint but you can put in weak passwords on these. They don't have anything to do with network security, it's for encrypting the data on-disk. For my testing, I just used "bank" as the bank password and "mint" as the mint password.

You will also need to specify an administrator. I've used the name `admin1` . As you might imagine, you also have to specify a password for the administrator.

Lastly, the bank installation relies on playground being configured and active. If you don't have playground setup, this won't work either.

```
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
initialize server generate 20191_bank admin1 --verbose
```

I won't show the complete output, but it steps through generating a certificate and key for the bank, passwords, bitpoints, and the administrator. If you come across any errors, please let me know.

Once the bank server is generated, it requires one configuration parameter: the port it runs on. You can pick anything. In the example below, I picked 700

```
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
config server:port 700
```

You can now start the server. When the server starts up, it needs to decrypt all of the bank data stored on disk and will ask for a bank password. This is the bank password you used in the initialization phase (again, I just used "bank" for this low-security test).

```
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
server
Enter bank password:
Bank Server Started at (20191.10.20.30, 700)
To access start a bank client protocol to (20191.10.20.30, 700)
```

You will notice it says the bank is using `20191.10.20.30`. That's the address we configured on our nic, of course. The bank automatically located the right virtual hardware.

Now let's start up a client. First open a different window. Navigate to the bank directory, activate the virtual environment, and make sure everything is working.

When it comes time to connect to the class bank, you won't need to generate a server. I will be running the class server. Your local server is for your local experimentation. The bank client needs to have a certificate for the bank so that it knows when it's really speaking to the bank (and not an imposter). The client also has an initialization process, but for systems where the server is already running, *you don't need to do a separate client initialization*.

We do need to configure our client to connect to the bank. Instead of command-line parameters, this is stored in a local config file:

```
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
config client:bank_addr 20191.10.20.30
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
config client:bank_port 700
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
config client:username admin1
```

Now we're ready to run the bank:

```
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
client
```

If everything worked correctly, the client will spit out a bunch of messages about the network connection until it gets to the bank shell. The client shell is pretty powerful. It supports tab complete and up-arrow to repeat previous instructions.

Once your connected to the bank, it's time to create a few accounts, add a few users, and give them some money. (You may have to press "Enter" after you see the message "Logged into the Bank").

```
Bank Client> account list
```

You will see that the only account present is `__admin__` . This is a special account you shouldn't generally touch. What we want to do is create some users and some accounts and give the users permissions on those accounts

```
Bank Client> help
help [cmd]
batch [batch_file]
quit/0
admin/*
access current/0
access current [user/account]
access current [user] [account]
access set [username] [access]
access set [username] [access] [account]
account list/0
account list [user]
account current/*
account switch [account name]
account balance/0
account balance all/0
account deposit [bp file]
account withdraw [amount]
account transfer [dst] [amount] [memo]
account create [account name]
user list/0
user list [account]
user create [username]
user passwd/0
user passwd [user]
export/0
export [account]
```

We'll start by creating two users `user1` and `user2` :

```
Bank Client> admin
Bank Client [Admin]> user create user1
Bank Client [Admin]> user create user2
```

Although not shown above, the program will ask you for an initial password for `user1` and `user2` . Next, let's create some accounts

```
Bank Client [Admin]> account create account1
Bank Client [Admin]> account create account2
```

Now we need to give permissions to the users to these accounts. We'll give `user1` access to `account1` and `user2` access to `account2` .

```
Bank Client [Admin]> access set user1 * account1
Bank Client [Admin]> access set user2 * account2
```

These two commands say to give `user1` all permissions ('*') to `account1` (and likewise for `user2`). There are a couple of different permissions and you can see them by running the command `access current`

```
Bank Client [Admin]> access current user1
      Current Access:
        account1: btdwa
```

- 'b' is for balance. If you only have `b` , balance is the only command you can do with an account
- 't' is for transfer. Likewise, with only this permission you can only transfer
- 'd' is for deposit
- 'w' is for withdrawl
- 'a' is for account admin (add other users, etc)

Note that account admin is not the same as global admin. Your administrator user (`admin1`) has administrative rights to the *bank*. But `user1` has been given administrative rights to the account `account1` . This allows `user1` to add users, give them permissions, etc.

Now. Money. Bitpoints. The bank, in its default configuration, was pre-loaded with 100,000 self-generated bitpoints. These bitpoints will not be accepted by the real bank that the class will eventually use, because they aren't signed by the right mint certificate. But it gives you something initial to play with. All of the pre-loaded bitpoints are in the bank's hidden "VAULT" account. As an administrator, give yourself transfer permissions to this account, and transfer some funds to user1:

```
Bank Client [Admin]> account switch VAULT
Bank Client [Admin]> access set admin1 t VAULT
Bank Client [Admin]> account transfer account1 100 "This is an example memo."
```

When you're done, quit and restart the client logging in as user2. You can either call the configure command to change the client's configuration, or override with the command line `-x` parameter:

```
[~/netsec_env/BitPoints-Bank-Playground3/src] (netsec_env) > python OnlineBank.py
client -x username user1
```

When you connect to the bank this time, you'll find you have far fewer privileges. You can switch to `account1` and get a balance, but you cannot even see `account2`. You can transfer money to it though! Transferring money *to* an account does not require permissions on that account. After all, who's going to complain if you deposit?

Play around and see what works (and what breaks). Most of the bank commands should be working. However, deposits and withdrawals (which actually transfer bitpoints over the network) is not guaranteed at this stage.

Clone this wiki locally

```
https://github.com/CrimsonVista/20194NetworkSecurity.wiki.git
```