Graham Dungan
February 10, 2025
UIN: 332001764

CSCE 413: Software Security
PoC 8: Port Knocking

Create a Network Application

For this assignment, I created a simple Python application that accepts UDP requests through several sockets. A demo script presenting all major requirements can be found as `demo.sh`. To run the demo script;

1. Enter the `PoC8` directory.
   `cd PoC8`
2. Run the script.
   `./demo.sh`

Application Behavior

The python server can be ran via `python3 ./knockknock.py`.

This program offers a class called `KnockingServer`. This class accepts a sequence of ports (in the order that they should be knocked) and a timeout for the intervals between knocking. If a knock is received more than that many seconds later, the sequence is reset before evaluating the new knock.

The `run()` function starts running the server after it has been instantiated. The server will first create individual threads hosting sockets that listen for incoming UDP requests for each port in the sequence. More directly, the `start_listeners()` function is called that runs a thread for each port, calling the `listener()` function. Once all threads have been created, they will monitor all ports and call a handler function `handle_knock()` for their respective port if a UDP request is made.

The `handle_knock()` first checks the time of the incoming request compared to the last request, and ensures that the knocks occurred within the timeout range. If the newest knock occurs at a time beyond the timeout period, it will reset the sequence and continue to check the knock. Next, the function then compares the incoming knock with the expected knock, as tracked by an index `expected_knock_index`. If the knocks are the same, it traverses the sequence, otherwise, it resets the sequence. The sequence is complete when the `expected_knock_index` matches the length of the knock sequence array.

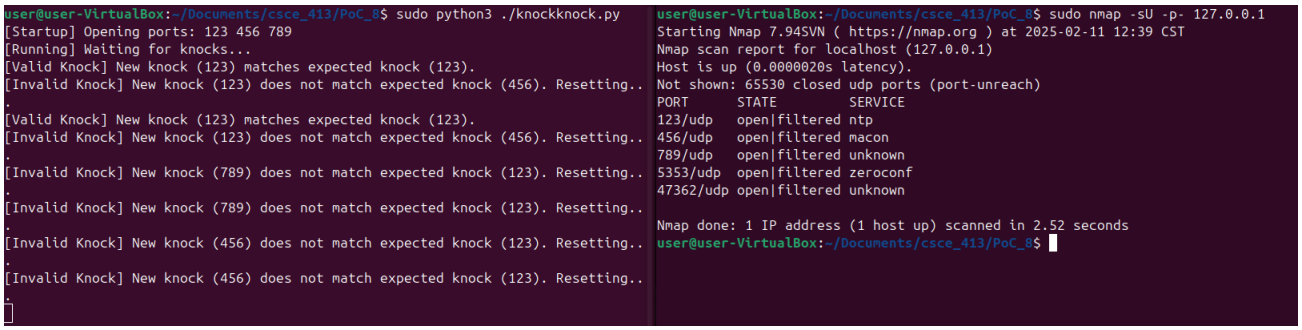
Once the sequence is complete, the application will close all ports and open port 8080 for its main service. It should be noted that a real-world application of this will not need to close all other ports, but I am doing this just to demonstrate a change in the available sockets.

Scan with NMAP

Once the program is running a scan can be performed with nmap. Since the program is opening UDP ports, some modifications will be made to the nmap scan,

```
nmap -sU -p- 127.0.0.1
```

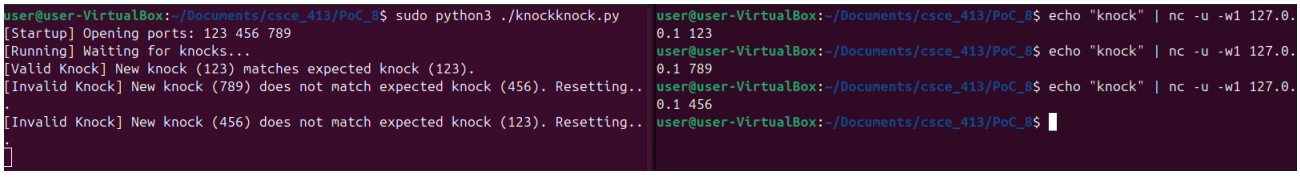
Where the `-sU` flag specifies to scan for all UDP ports, `-p-` scans for all ports from range 1-65535, and `127.0.0.1` specifies the localhost. Running this scan reveals,



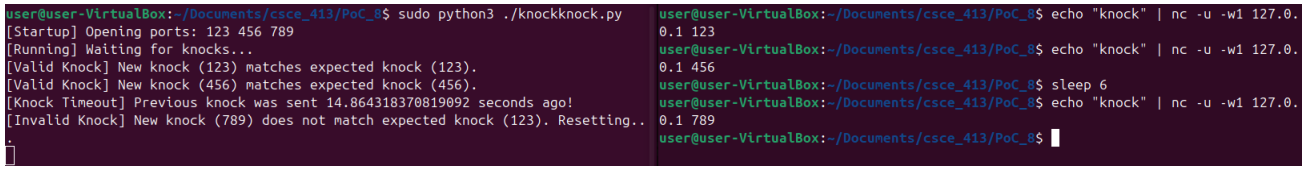
As seen here, ports 123, 456, and 789 are open. Other dummy ports could also be added to this script for decreasing the change of brute force knocking. Note that port 8080, the main service, has not been opened as the correct sequence of knocks has not been implemented yet.

Implement Port Knocking Logic

The port knocking logic can be found on lines 45-73 of `knockknock.py`. As explained earlier, the service only traverses the sequence of knocks if the received knock matches the expected knock. if a timeout or incorrect knock occurs, the sequence is reset.
What follows is a demonstration of an incorrect order of knocks being performed,



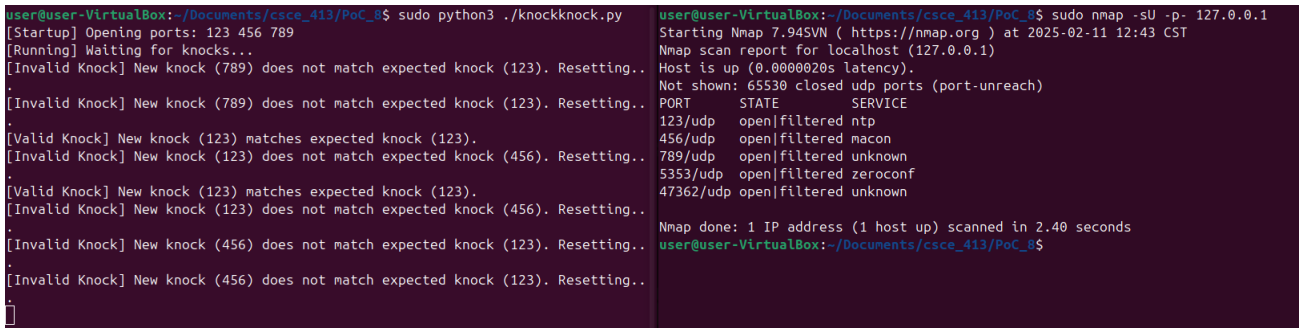
What follows is a series of correct knocks interrupted by a timeout,



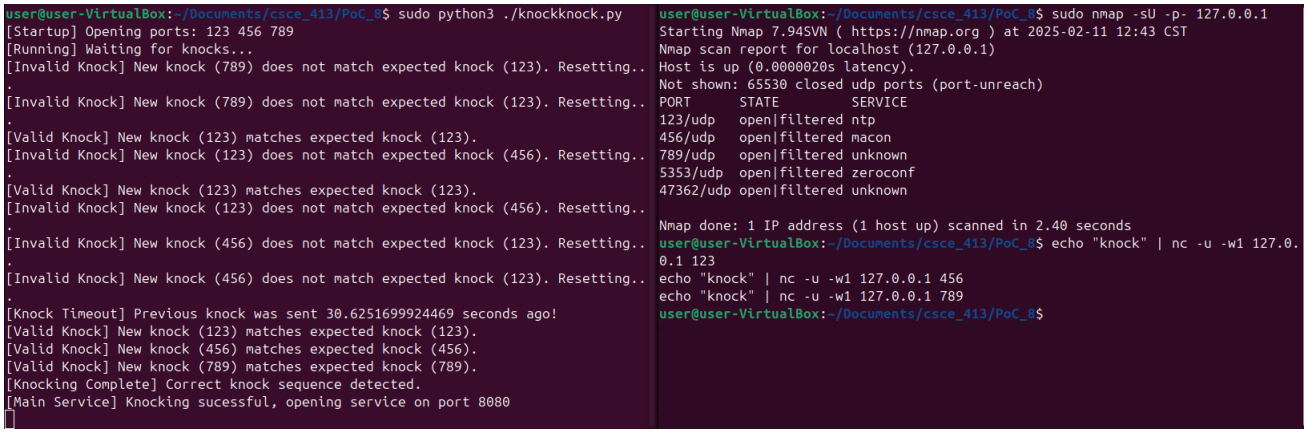
As seen in these examples, inputting an incorrect sequence of knocks or waiting too long will result in the sequence being reset.

Before and After Knock Sequence

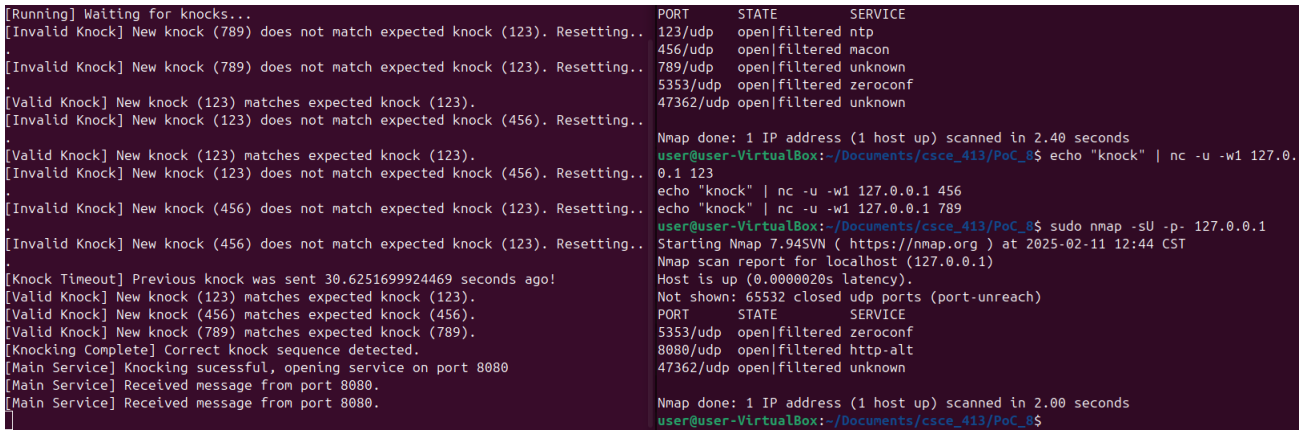
First we will run the nmap scan as discussed earlier to reveal that port 8080 is not open.
We will then make the series three correct knocks.



We will now run nmap again to view all opened ports.



It is seen that, after the correct sequence of knocks, the server has opened port 8080 for communication.



Demo Script

What follows is screenshots of the demonstration script.