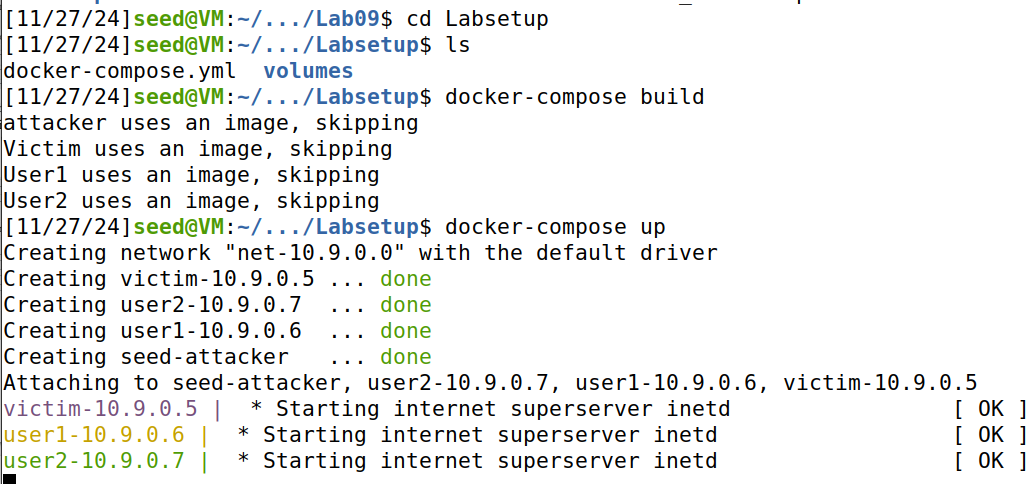
CS445 Lab Report: **TCP Attack and Vulnerabilities**

Austin Decker

06 December 2024

# Environment Setup

**Building and Running the Containers:**

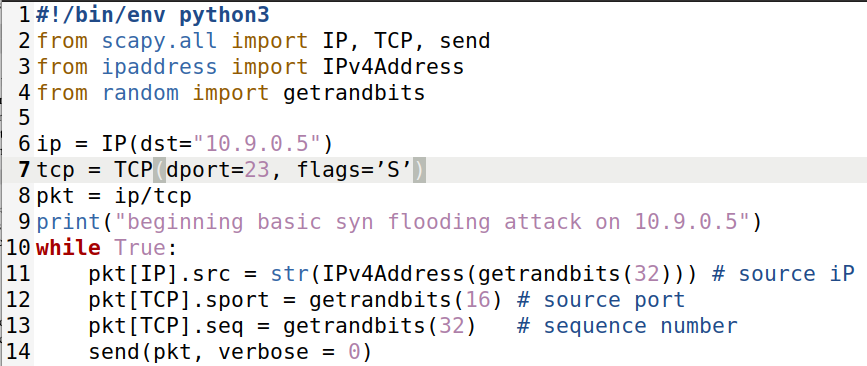
****

# Tasks:

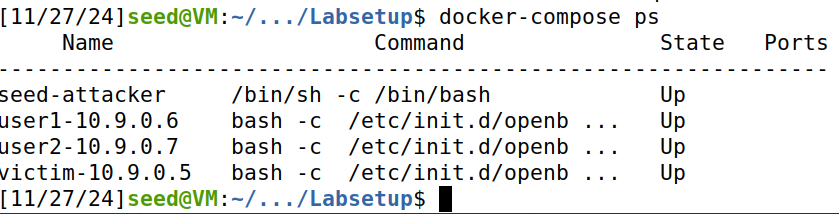
## Task 1: SYN Flooding Attack

### Task 1.1: Launching the Attack Using Python

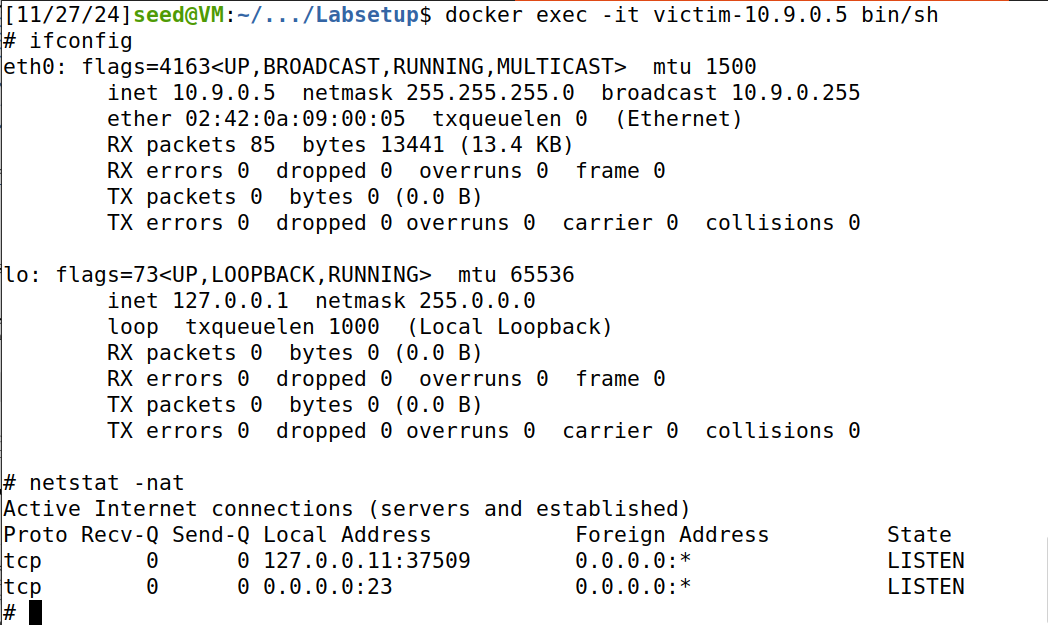
**Code:**

****

The Destination IP was determined by the IP of the Victim container:

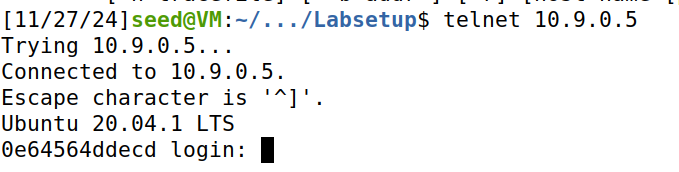


**Running Default Attack (Victim Queue Before)**

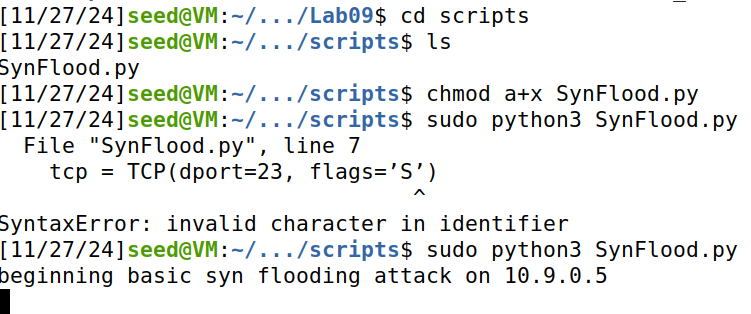
****

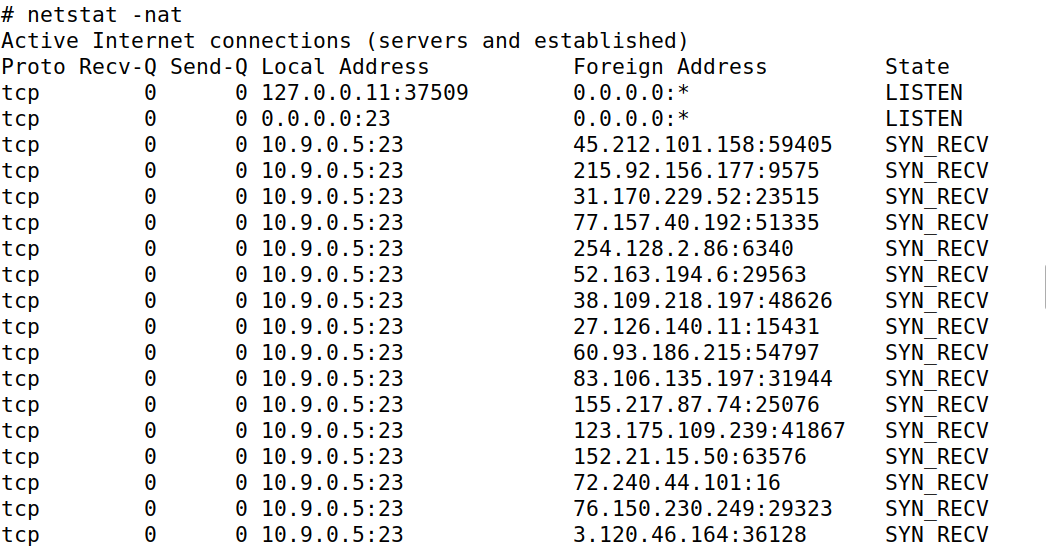
The above screenshot shows proof of being inside the victim container. It also shows the current state of the connections queue before the base syn flooding attack.

**Telnetting into victim (before attack)**

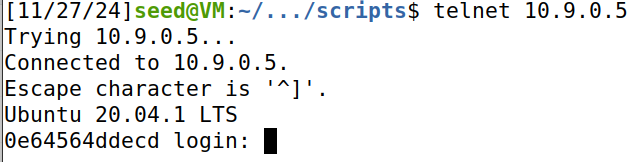
****

**Running Default Attack 3 mins (Victim Queue after)**

****

****

**Making a Telnet Connection To Victim**

****

The Python script failed the syn flooding attack against the victim (10.9.0.5). There are a variety of factors that could be causing the attack to fail:

* **TCP Cache Issue**

The screenshot shows that the victim remembers our host (10.9.0.1) when it telnetted before the attack

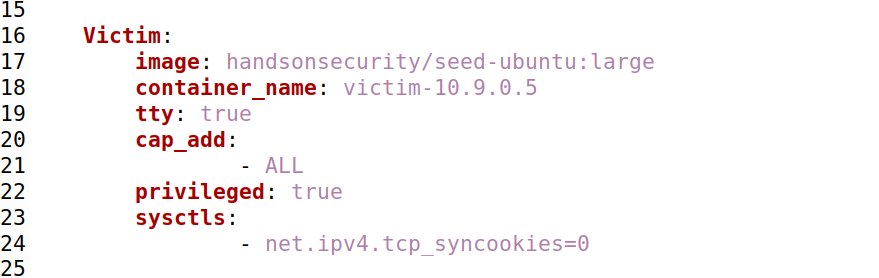
* **TCP retransmission Issue**

The default re-transmission for the victim is 5, afterwards the half-open connection is removed from the victims connection queue. The script may not be fast enough to fill up the newly opened slots which allows for the legit telnet connection to go through.

* **Victim Queue size**

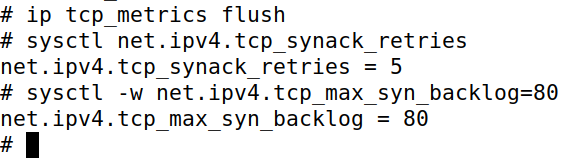
Our Victim container has a backlog queue of 256

The SYN cookie Countermeasure default on Ubuntu has been turned off in the lap setup. It has been turned off for me inside the docker-compose.yml provided in the lab.



**Redoing the attack with modified settings**

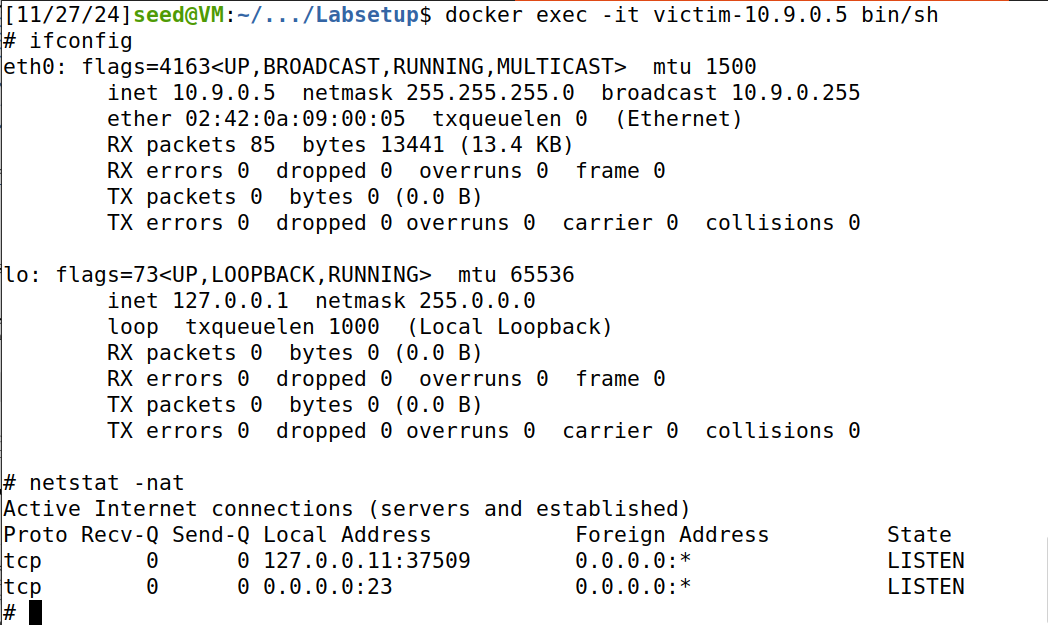
**Lowering The Victims Queue capacity and flushing ip tcp\_metrics**

****

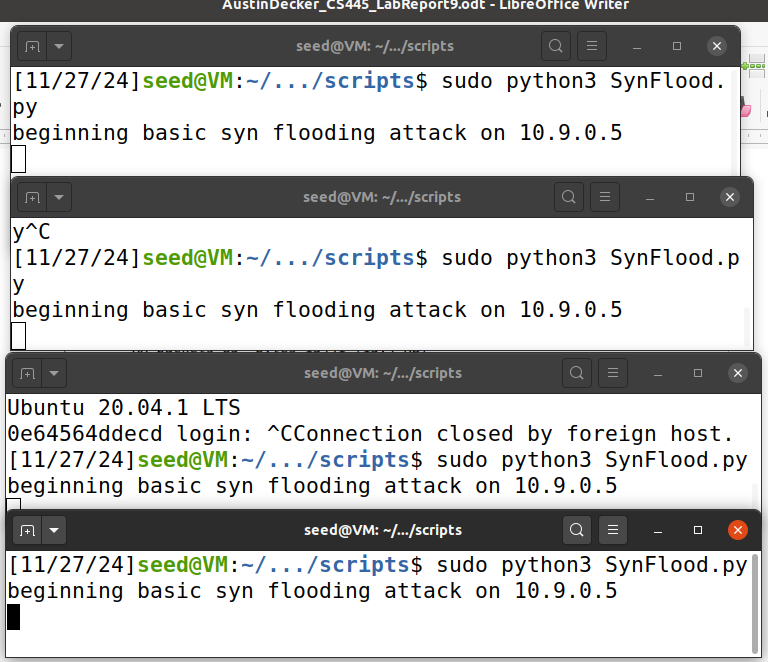
Note that I will also **not** be tel-netting into the victim before the attack to prevent the TCP Caching issue.

Note that I will also be running the synflooding attack in 4 different terminals to counter the re-transmission issue.

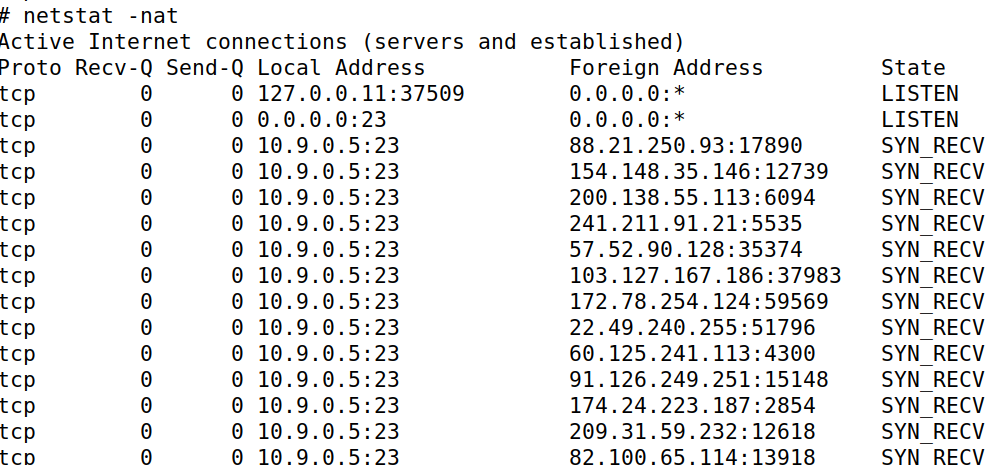
**Victims Queue (Before Attack)**

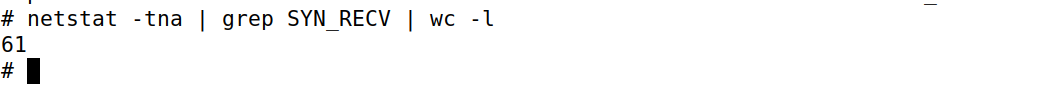
****

**Running Multiple Attacks Simultaneously**

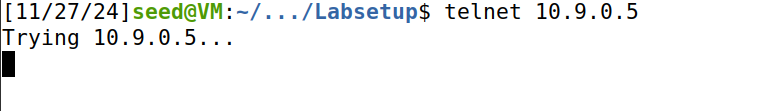
****

**Victims Queue and buffer count (after attack)**

****

****

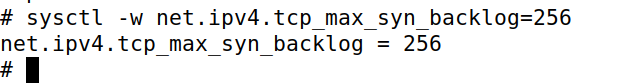
**Telnetting into Victim**

****

With the modifications done on the victim container as well as running multiple instances of the script, I was able to successfully perform a syn flooding attack against the victim. 4 instances of the script running simultaneously seemed to be enough to counter the re-transmission issue and have a high success rate.

## Task 1.2: Launch the Attack Using C

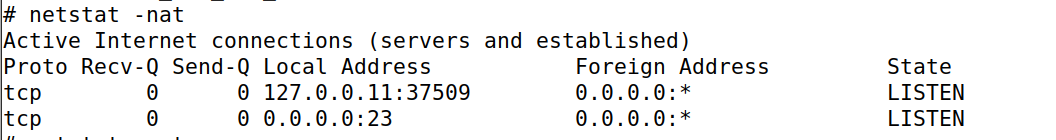
**Restoring Victim Queue Size**

****

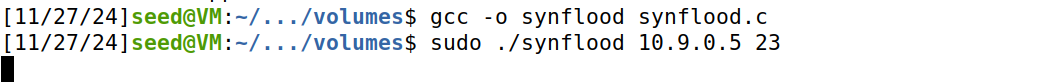
**Code:**

The provided C code is too long to capture as a screenshot

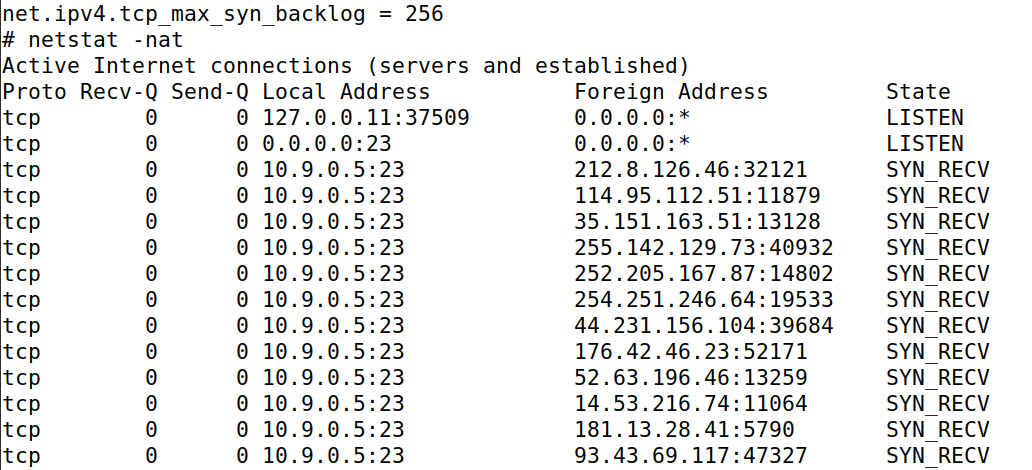
**Victim connection Buffer (before attack)**

****

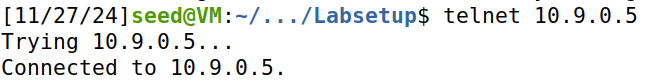
**Compiling and Running the syn flood c code implementation**

****

**Victim connection Buffer (After attack)**

****

**Telnetting to victim**

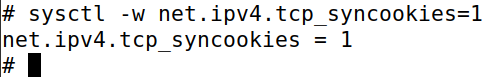
****

It should be noted that while telnetting to the victim container, It took a significantly long time until the telnet connection was received. While the C synflood attack didn’t fail completely, a legit packet still got through. This could be because of the previously mentioned issues in task 1.1. However even though it partially failed, the C implementation is still noticeably faster and more potent compared to the python implementation. In order for the python implementation to succeed, It had to be run multiple times and the victim connection queue had to be shrunk because it is much slower. The C implementation did not require any additional modifications to the victim container for it to work and did not require multiple instances running in the background to make up for the lack of speed.

Running the c script again, I was able to successfully perform the syn flooding attack against the victim container. Whatever happened during my first attempt seemed to have not happened this time.

## 

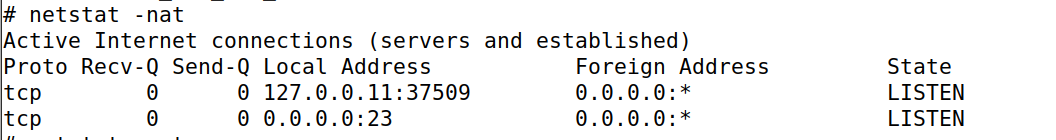
## Task 1.3: Enabling SYN Cookie Countermeasure



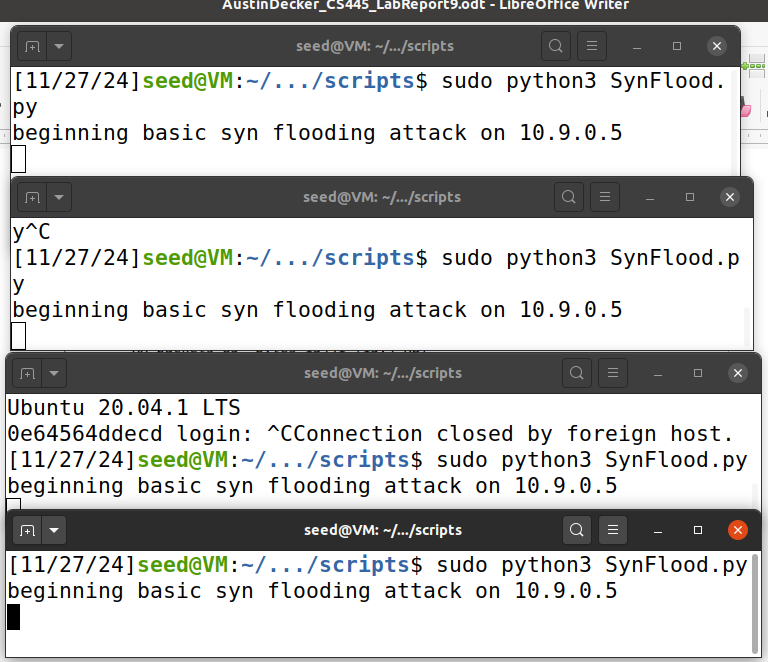
**Clearing Cache**

****

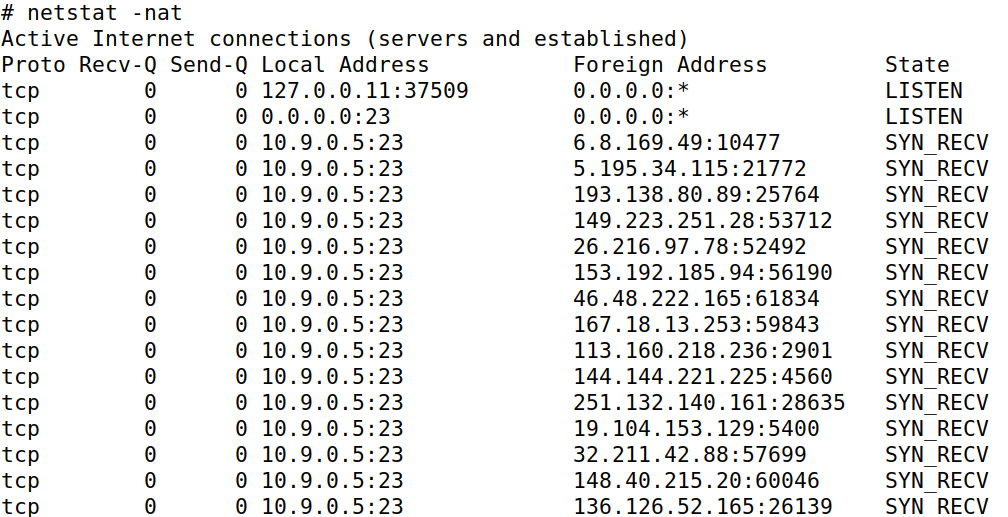
**Victim Connection Buffer (Before Attacks)**

****

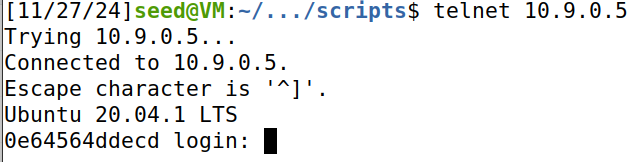
**Running synflood.py (4 instances)**



**Victim connection Buffer**

****

**Running Telnet**

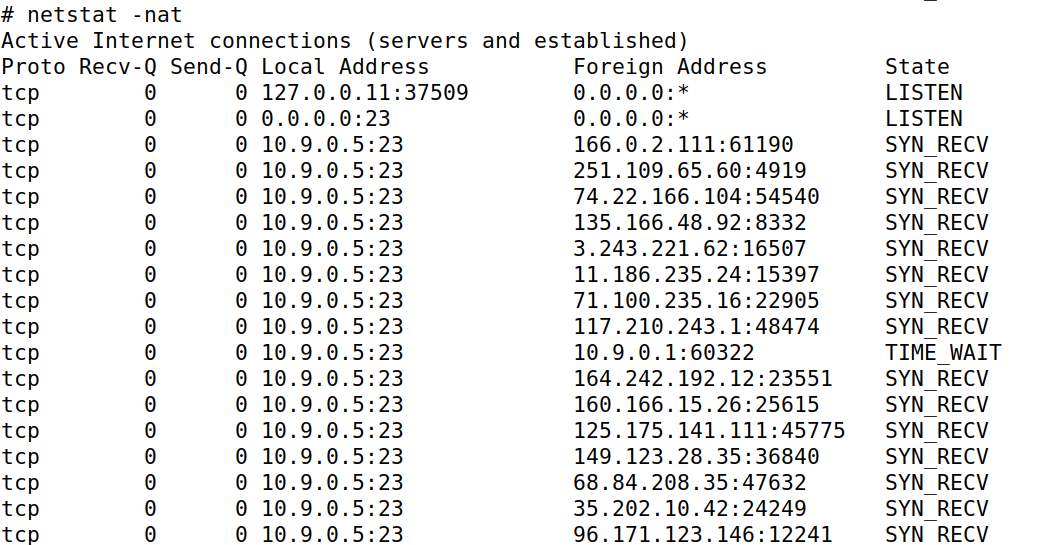
****

As the screenshot shows, the python script failed to block the legitimate connection to the victim container.

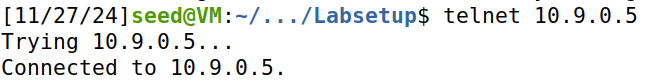
**Running c implementation**

****

**Victim Connection Buffer:**

****

**Running Telnet**

****

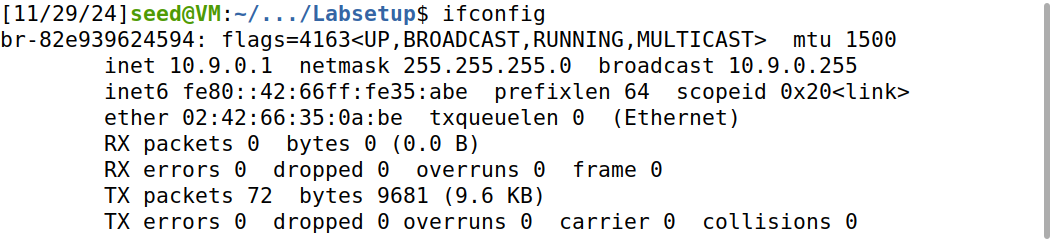
The C syn flood script also failed to prevent the host from telnetting to the victim container.

## Task 2: TCP RST Attacks on telnet Connections

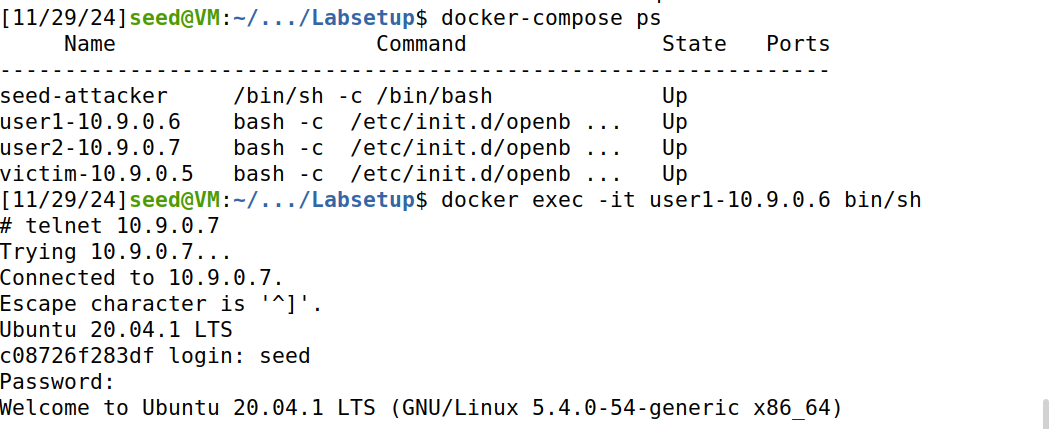
**Code:**

****

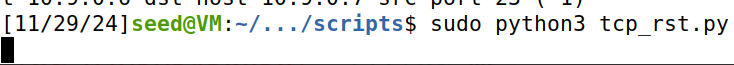
**The Interface was determined by the command below:**

****

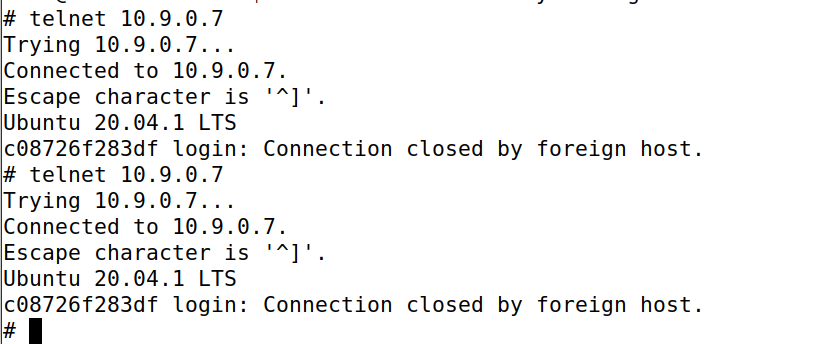
**Starting Telnet connection between User1 and User2**

****

**Running the automated tcp\_rst.py script (as 10.9.0.1)**

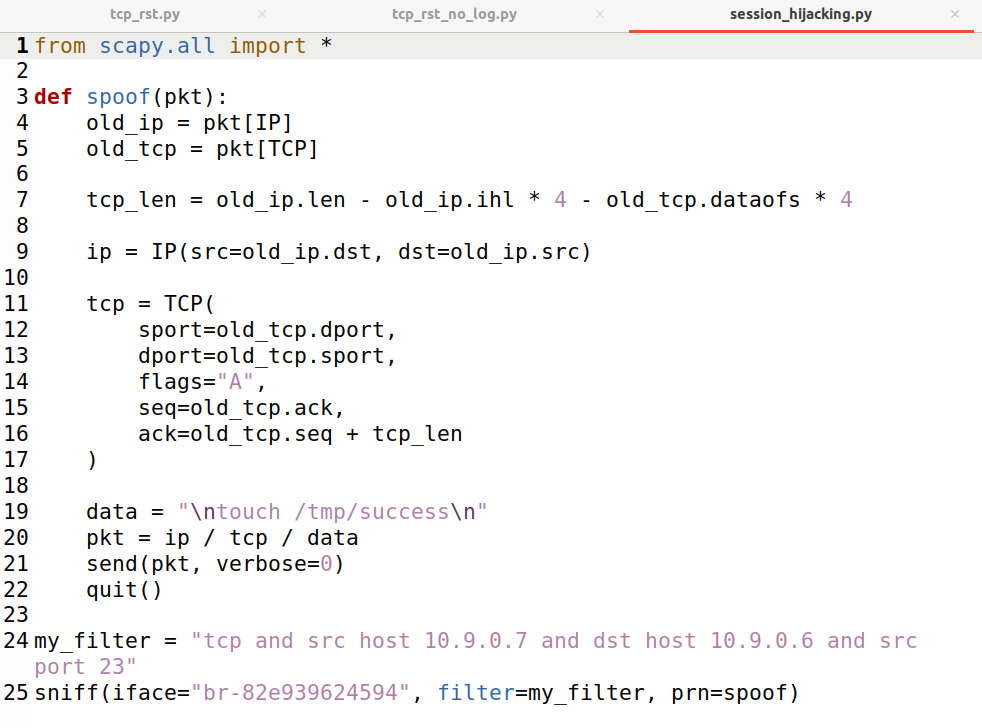
****

**Telnet Connection Closed:**

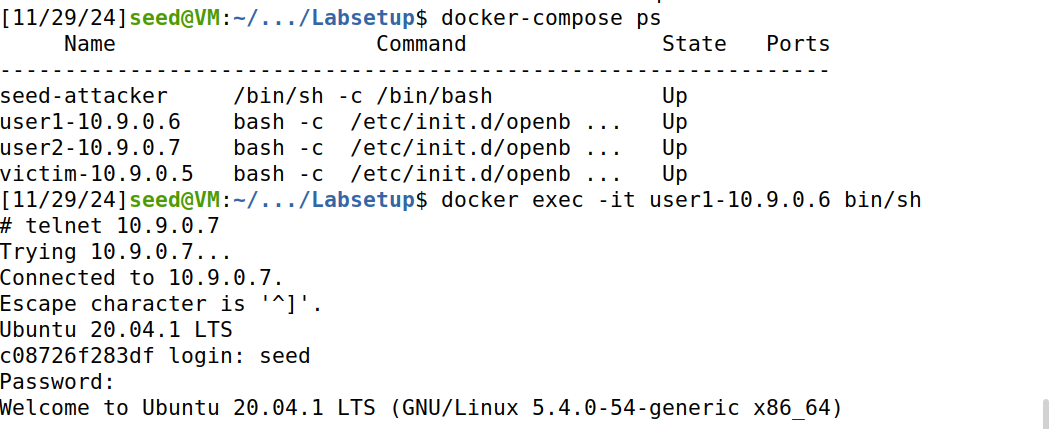
****

## Task 3: TCP Session Hijacking

**Code**

****

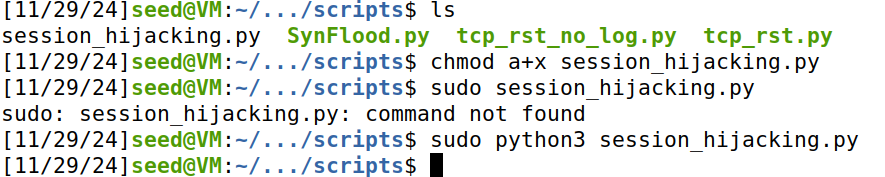
**User1 telnetting into User2:**



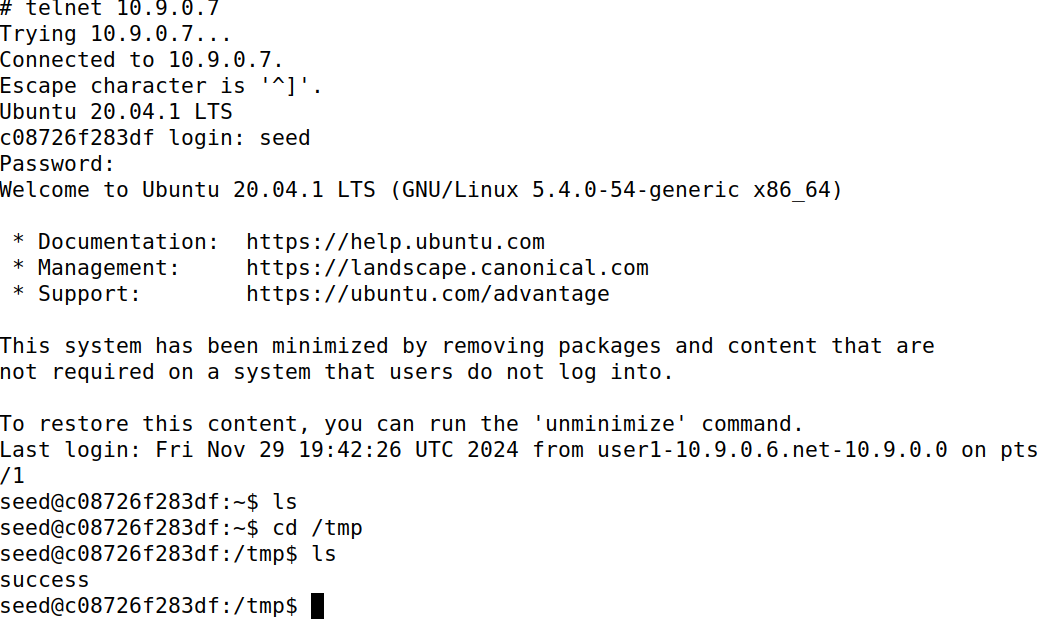
**tmp directory before script (10.9.0.7)**

****

**Running session\_hijacking.py (10.9.0.1)**

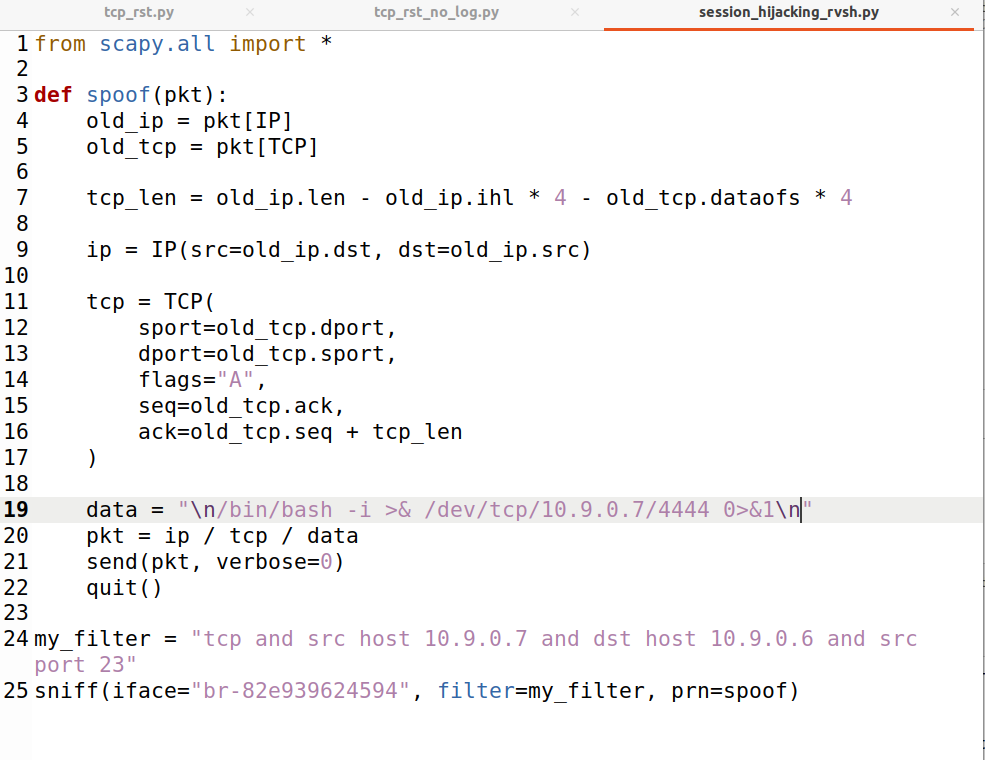
****

**Script Results:**

****

## Task 4: Creating Reverse Shell Using TCP Session Hijacking

**Code:**

****

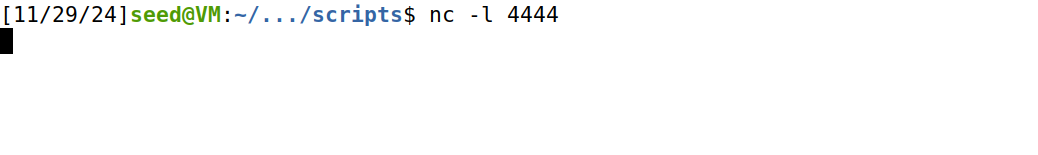
**User 1 Telnetting into User 2:**

****

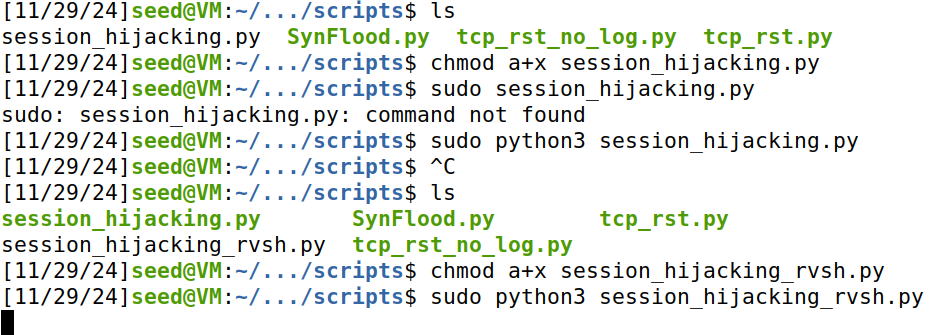
**Before Reverse Shell:**

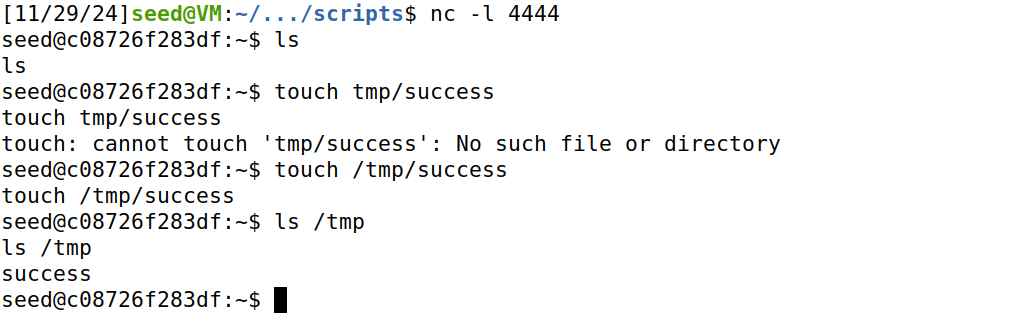
****

**Host 10.9.0.1 Runs tcp server on port 4444**

****

**Running Modified script (10.9.0.1) and getting root shell**

****

****

The above root shell occurs when user1(10.9.0.6) tries to type something while in a telnet connection with user2(10.9.0.7). The attacker (10.9.0.1) sniffs the telnet communication between user1 and user2 and uses the modified hijacking script to get access to user2.