Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Network Security Group, Department of Computer Science, ETH Zurich

Information Security Lab: Module 05

Defense Labsheet

TAs: Christelle Gloor, Felix Stöger

DDoS Attacks & Defenses

Fall 2022

Emails: christelle.gloor@inf.ethz.ch, felix.stoeger@inf.ethz.ch

Deadline: 05.12.22, 15:00 CET

1 Problem Overview

In this exercise, you are tasked with providing a DDoS protection service for a fictional customer. This will require you to implement a firewall that can defend against increasingly sophisticated attacks by cleverly utilizing the network layer of the SCION architecture to block malicious traffic, while being careful not to accidentally block packets from legitimate clients of your customer.

1.1 Technical Setup

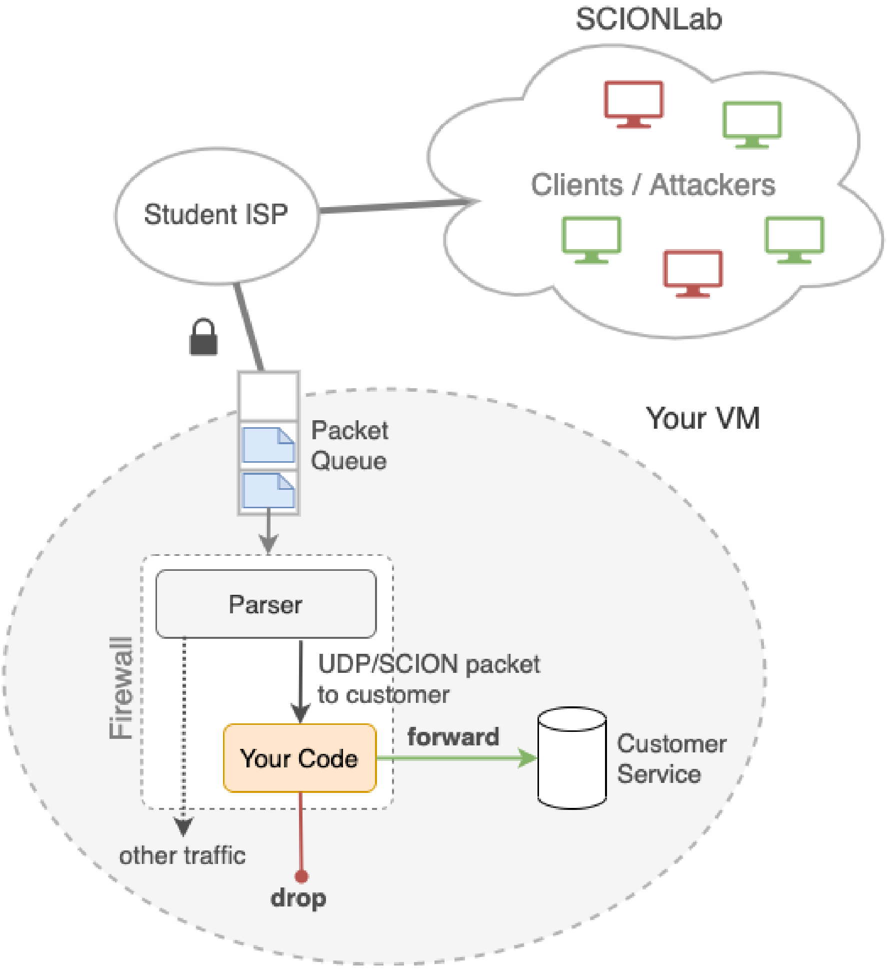

Figure 1: The relevant network components for this task. The customer server is exposed to SCION clients in our infrastructure network, and your task is to filter incoming requests.

You will continue working with the same VM as last week. The relevant components for this task are shown in Fig. 1.

We have set up a code skeleton for a firewall that intercepts all incoming traffic to the customer.

Behind the scenes: This works using a `netfilter` queue that receives traffic via an `iptables` rule. The packets will end up in this queue, and if your firewall is running, it will inspect these packets and make a decision on whether or not to drop them. While your firewall is not running, traffic is delivered to the customer unfiltered, i.e., the queue is bypassed.

Your task: Implement the decision logic in the firewall that determines for each packet, whether to forward it to the customer or drop it. Your code may keep state and use any kind of data structures to make these decisions.

1.2 Attacks

The task is split into three increasingly sophisticated attacks, to which we will refer as *Level 1*, *Level 2*, and *Level 3*. Each level will earn you a number of points toward your final score (as defined in §4.1).

Part of the challenge is that you are not allowed to submit separate solutions for each of these subtasks. Instead, the goal is to build **a single solution** that defends against all three different classes of attacks.

In this sheet, we will not provide any description of the attacks. It is part of the task—just like in real-world DDoS mitigation—to inspect the attack traffic and come up with effective measures against it.

1.3 Customer Service

The customer you are protecting exposes a public service that runs the following simple protocol:

$$\begin{aligned} A \leftarrow B: &\quad n_B \\ A \rightarrow B: &\quad n_A, \text{data}, \{\text{data}\}_{\text{sk}_A} \end{aligned}$$

where n_B, n_A are randomly generated nonces, and the notation $\{x\}_{\text{sk}_A}$ refers to a signature over x generated with the private key of A. In essence, upon receiving a request from a client B, the server A computes some data and sends it back to the client along with a signature.

These computations on the server take some time, which is simulated in this lab by an artificial processing delay of around 400 ms. Therefore, it is important that your firewall keeps the forwarding of malicious requests to the server minimal, ensuring that it has enough resources left to serve legitimate customers.

Since you are playing the role of an external DDoS defense service with potentially many customers, you do not have the ability to change the target protocol. Instead, your task is to put network-layer defences in place that protect the server.

2 Implementation

You can find the code skeleton for the firewall at `~/src/defense`.

- `main.go` contains the entry point of the program and a skeleton for the `filter` function, which is called for each packet.
- `firewall.go` is our implementation of the firewall provided to you. You will not need to touch this file to solve the task.
- `print.go` contains some auxiliary code for pretty-printing packets. Feel free to modify it or use it as a reference for accessing packet fields, e.g., the SCION path.

Simply run `make` to compile your firewall, and then execute it using the command `firewall`[1]. If `make` returns with "`Nothing to be done for 'default'`", run `make clean` and then try again.

You are only allowed to use the following Go libraries in your submitted code for this week:

- Standard Go libraries (i.e., that have no domain prefix like `github.com`)
- The libraries already imported in the code skeleton files (i.e., the SCION `slayers` or the `gopacket` libraries)

The automatic grader will check this rule when you submit your code and alert you if your code is violating it. **If your final submission contains unauthorized imports, you will get zero points for your solution**.

You are free to use any tools and libraries while working on your solution (e.g., Wireshark or tshark for analyzing packet traces).

3 Testing

To launch test attacks, which are identical to those run on the grading pipeline, you can use the pre-installed `isl` command-line tool:

```
isl run def1
```

This starts the Level 1 traffic generator and provides immediate feedback on your performance. You will usually want to run your firewall in parallel, so we recommend using `tmux` again to keep track of both outputs side-by-side. While the attack is running, the *Defense Task* dashboard on your local Grafana server (at `http://localhost:8002` on your host machine) provides live feedback.

[1] The binary is linked to `/usr/local/bin` automatically, and you should also execute it from there.The reason for this is that setting capabilities is not supported on most mounted volumes, such as your `src` folder.

You are allowed to run as many local test attacks as you want.[2] Note that **each run is randomized**, so you cannot rely on observing the attacker's locations from past attack traces. Instead, your firewall must be able to make its decisions based on live traffic.

4 Grading

We use the same GitLab setup for testing and submission as for the first part of this module. Because the runs are non-deterministic, we will execute your code multiple times, remove the worst result and take the median of the remaining runs as your final score.

4.1 Grading Scheme

The points for this part of the lab are distributed as follows:

- **Level 1:** maxScore = **15**
- **Level 2:** maxScore = **15**
- **Level 3:** maxScore = **20**

For each level, your score will be determined by the **success rate** r of requests from legitimate clients. A request counts as *successful* if the client receives a response from the server within a couple of seconds. It counts as *failed* if no response is received for the request, for which there are three potential reasons: (a) the customer server drops the server because its processing queue is full; (b) the server takes too long to process the request so it times out; or (c) the request is dropped by your firewall.

The success rate for a run is computes as follows:

$$r := \frac{n_{\text{success}}}{n_{\text{success}} + n_{\text{failed}}}$$

where n refers to the number of requests issued by clients over the period of the run. To account for a certain degree of randomness for empty solutions (i.e., some client requests may get through by chance), we adjust the scale slightly. Success rates under a threshold μ are not awarded with any points. The following formula computes the final score:

$$\text{score}(r) := \frac{\max(r - \mu, 0)}{1 - \mu} \cdot \text{maxScore} \quad \text{for } \mu := 0.2$$

[2] The traffic orchestrator will enforce that the capacity of the traffic generation infrastructure is not exceeded and may sometimes deny test runs. Run `isl status` to check the current load on the system.

Example: for Level 1, a success rate of $r = 0.1$ will give 0 points. A mediocre solution with $r = 0.7$ will give 9 points, and a near-perfect performance of $r = \frac{54}{55}$ will be awarded with a full score of 15 points (rounded to the nearest integer).

4.2 Final Submission

Your most recent commit on the master branch before the deadline will count as your submission. We will re-run your solution after the lab has concluded, to ensures that your solution performs consistently. For that, we run your solution six times, and take the median of the best five runs (worst run is discarded).

If you want to keep experimenting but still have your current submission to fall back on, we recommend using a separate branch. Commits on other branches will not be considered for scoring by our system.