Network Security 2019

Project 2: Firewall Rule Generator

2019-11-07 T12:49

1 General Information

- This is the second of two graded course projects. This project will determine 7.5 % of your final grade. Not handing in this project will result in a 1 for this project, but passing the course is still possible.
- This project must be completed individually. You are of course allowed to discuss the `iptables` specification and firewalls in general with your colleagues; however, discussions about the implementation and sharing of code is not permitted.
- Plagiarism checks will be performed. In addition to manual checks we use an online plagiarism checker hosted by a foreign research institute. Therefore, please ensure there is no sensitive personal information in your code.
- The due date of this project is **2019-12-20 23:59** (UTC+1). This is a hard deadline that will be automatically enforced.
- In case you experience any issues during the project, please ask for help using an issue in the corresponding Gitlab repository.[1] Only send an email to our mailinglist[2] for personal or sensitive questions (e.g., military service).

2 Project Goal

In this project, you need to create a program that receives a network description and a set of allowed communications and produces the correct iptables rules for each router in the network, such that no unallowed communication is possible. Documentation about iptables can be found at `https://netfilter.org/documentation/index.html` or via the commands "`man iptables`" and "`man iptables-extensions`" in Linux.

2.1 Input Definition

Each test case is a JSON file, describing a network and its allowed communications. Note that only those communications may take place in the network and *all others should be prevented*.

[1] `https://gitlab.inf.ethz.ch/PRV-PERRIG/netsec-course/netsec19-issues`
[2] `networksecurity-project@lists.inf.ethz.ch`

2.1.1 "network"

You are provided with an undirected bipartite graph of routers and subnets. A link can only exist between routers and subnets. The network is also a tree. The *"network"* consists of

- an array of *"routers"*, each having
  - a unique *"id"*;
- an array of *"subnets"*, each having
  - a unique *"id"*,
  - its network *"address"*, and
  - its network *"prefix"* length; and
- an array of *"links"* between a router and a subnet, each having
  - the *"routerId"*,
  - the *"subnetId"*,
  - the *"ip"* of the interface on the router (unique per router), and
  - the *"interfaceId"* of the interface (unique per router).

2.1.2 "communications" array

- For each communication object, there is
  - *"sourceSubnetId"*: the subnet id of the source of the communication;
  - *"targetSubnetId"*: the target subnet id
  - *"protocol"*: either `tcp`, `udp`, or `icmp`;
  - *"sourcePortStart"*: the minimal port (inclusive) that the source is using;
  - *"sourcePortEnd"*: the maximal port (inclusive) that the source is using;
  - *"targetPortStart"*: the minimal port (inclusive) of the target;
  - *"targetPortEnd"*: the maximal port (inclusive) of the target; and
  - *"direction"*: `bidirectional` if the communication should be allowed in both directions, or `unidirectional` if it should only be allowed in one.

2.2 Output Format iptables

You need to store the filter rules of each router in the iptables restore file format. This format is specified as follows:

```
*{table}
:{chain} {policy} [{packets_count}:{bytes_count}]
{rule}
```

```
COMMIT
```

An example that does not perform network address translation (NAT) and drops all traffic except TCP and UDP is shown below.

```
*nat
:OUTPUT ACCEPT [0:0]
:PREROUTING ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
COMMIT

* filter
:INPUT DROP [0:0]
:OUTPUT DROP [0:0]
:FORWARD DROP [0:0]
-A FORWARD -p tcp -j ACCEPT
-A FORWARD -p udp -j ACCEPT
COMMIT
```

Because our verification infrastructure has certain limitations, you need to pay attention to the following points:

- Always include the `nat` table, even if it is empty.
- Always include all chains.
- Do not use `--source-port`, only use `--sport`.

2.3 Assumptions & Clarifications

- The given network is a tree. Thus, there is a unique path between any two nodes in the network.
- Hosts are not explicitly included in the given model.
- Hosts will only receive packets that have their destination. There is no broadcast IP traffic. Hosts cannot fake their source IP address.
- Assume that there is one switch per subnet, connecting all hosts in a star-topology.
- Hosts do not forward packets.
- The requirement is for all tasks to prevent *any* communication that has not been explicitly allowed in the input, including `RELATED` communication.
- All hosts in a subnet are allowed to communicate freely between each other.

- There is no NAT in the network.
- There are no overlapping subnets in the network.
- The `bidirectional` flag allows the target to send replies but *not* establish a new connection.
- The `unidirectional` flag is only used for UDP (as it does not make much sense for TCP or ICMP).

3 Organization

3.1 Git Repository

We have initialized a Gitlab repository[3] for you with the following contents:

- `input/`: In this directory you find all the test-case files.
- `output/`: If not overwritten with the `-o` flag, this is where the output of your application should be written to.
- `compile`: This file will be executed by the automated-testing environment before any tests are run. You should modify this file. If your project needs to be compiled, this file should contain the commands needed to compile the project. If no compilation is needed, this file can do nothing (or install dependencies). Note that you may only use dependencies that are explicitly whitelisted, see Section 3.4.
- `run`: This file will be executed by the testing environment when the tests are being run. You should modify this file. It will receive the command-line arguments listed in Section 3.2. Your compile script may overwrite this file.
- `evaluator.py`: Script you can use to verify your solutions, see Section 3.3.
- `.gitlab-ci.yml`: Used for the automated testing—do not modify!

3.2 Command-Line Arguments

Your application should support the following keyword command-line arguments (passed to the `run` file):

`-i INPUT_DIR` (optional, default = `inputs/`)
: Path to the directory with input files. Your application should scan this directory for test cases. Test cases in it will be named `:id.json`, where `:id` is the test-case ID. Your application should process all test cases found.

[3] `https://gitlab.inf.ethz.ch/COURSE-NETSEC-IPTABLES-AS-2019/<YOUR_ETH_USERNAME>-iptables-project-netsec-fall-19`

`-o OUTPUT_DIR` (optional, default = `outputs/`)
: Path to directory where you should write your output files. If this directory does not yet exist, you should create it. The content of this directory must follow the pattern `output/:testcaseId/:routerId`. We have initialized `outupts/` with a correct solution for the fist test case to illustrate this.

3.3 Verification Tool

You may use the `evaluator.py` script to verify your solutions without having to use the GitLab CI. This script uploads your solutions to our verifier service and takes the following arguments:

Positional Arguments

`Testcase id` (optional, multiple)
: The IDs of the test cases to verify. If none are given, all test cases found in the solution directory will be verified.

Keyword Arguments

`-s SOLUTION_DIR` (optional, default = `outputs/`)
: The directory where the script will look for your solutions.

If you see `«Something went wrong»`, this means that your submission could not be properly parsed by the evaluator framework.

Note that we only provide this tool for your convenience. The actual grading of the project will be done using the GitLab CI!

3.4 Guidelines and Restrictions

Only programming languages, modules, libraries, software packages, etc. that are explicitly listed on the whitelist[4] may be used. We will happily consider requests to extend this whitelist. In principle we will allow anything as long as

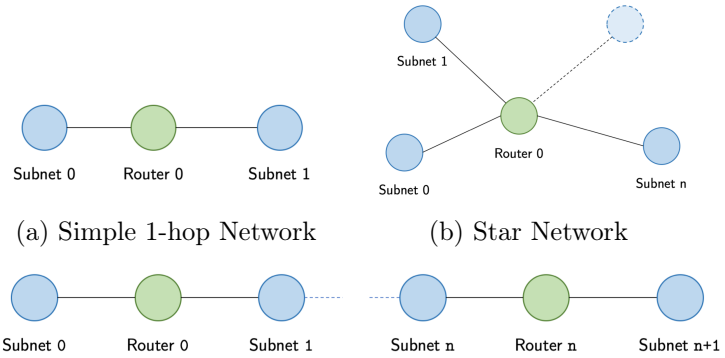
1. we can run it in our automated-testing environment and
2. it does not implement core functionality of this project.

You can request extensions of the whitelist by creating an issue on `https://gitlab.inf.ethz.ch/PRV-PERRIG/netsec-course/netsec19-issues`. Note that hard-coding solutions is considered cheating and will be detected.

[4] `https://svn.inf.ethz.ch/svn/perrig/netsec/teaching/netsec-2019/students/iptables-project/whitelist.txt`

4 Test Cases and Grading

There are 21 test cases with 167 routers in total; the first few test cases contain simple networks like 1-hop, star, and line networks (see Figure 1), the rest are random tree networks.



(a) Simple 1-hop Network

(b) Star Network

(c) Line Network

Figure 1: Examples of simple networks.

Your score for the project will be calculated as

$$\text{final score} = 1 + 5 \cdot \frac{\text{number of correct test cases}}{\text{total number of test cases}}.$$

Both the ratio of successful test cases and the corresponding grade will be returned by the `evaluator.py` script.

We will not make adjustments to this score. The only exception to this is when we notice that you did not follow the project rules or when we detect plagiarism. The auto grader is highly transparent and you have plenty of time to ensure that your code passes the test cases. If you experience problems with the auto grader, you must address these *before* the deadline.

Note that, in the beginning of the project, some test cases might be added. If this happens, we will clearly communicate this, and we guarantee not to add any test cases in the last two weeks of the project.

Note also that our test server does not have unlimited resources. It is likely that testing times will increase when many students start submitting solutions at the same time. We will not accept this as an excuse for failed tests and we will not extend the deadline because of this—there is plenty of time to submit.