Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Network Security

Exercise Sheet 06: Network Firewalls Part 2 & Network Monitoring & Intrusion Detection

Prof. Dr. Mathias Fischer, August See, Finn Sell

Summer semester 2024

Goals and Objectives Solve this task sheet in groups as you organized yourselves beforehand. Your solution should be handed in as a pdf document that describes your results and how you got there. Submit your group solution to this task sheet in Moodle. You should reach at least 50%.

1 Network Firewalls: Nftables (33.3%)

This task requires access to a Linux machine with nftables (see the VM we provided previously). To verify your rules, use or write a small program that sends TCP and UDP packets (cf. Listing 1). Use Wireshark or similar to verify your results.

For each task, explain your commands/rule, include a wireshark screenshot with packets showing the implemented behavior, and describe what is visible in the screenshot.

1. What is netfilter, and what are nftables?
2. What is the role of a table, chain, hook in nftables? What is the relationship between hook and priority in nftables?
3. Forward a TCP and UDP port using nftables. externalAddress:1234 <-> 127.0.0.1:4321.
4. Echo packets using nftables.
   - Echo incoming TCP/IP packets with destination port 9000 on all interfaces (default that connects to the internet, often eth0, and loopback, often lo).
   - Change the destination port of the packets to 9001.
   - Change the source IP of the packets to 1.2.3.4.
5. Drop outgoing UDP packets via the default interface that contain the bytes 0xca 0xfe at position 100 (position in bytes, counted from the start of the UDP header). Which netfilter hook and priority do you need to use for that and why?

6. Drop incoming TCP packets via the default interface that contains the bytes 0xbe 0xef at position 10000 (position in bytes, counted from the start of the TCP header). Which netfilter hook and priority do you need to use and why?

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
sock.sendto(bytes(message, "utf-8"), (ip, udp_port))
```

Listing 1: Send UDP packet using python

2 Network Monitoring using Libpcap and Tcpdump (33.3%)

The first step in network monitoring requires getting the network packets off the wire and parsing their content. Depending on the layer (the Internet model has five layers), you can do meta-data analysis or deep-packet inspection. In this first part of the exercise, you will do the necessary steps to capture and interpret network packets.

During the exercise, if not stated otherwise, use the pcap file `sample1m.pcap` (see moodle). This network trace is a shortened trace from the MAWILab[1] dataset that was reduced to the first one million packets.

2.1 Libpcap

One common way to retrieve and parse network packets within an application is to use libpcap. Since the library is written in C, we recommend solving the following challenges with C++. However, bindings are available for various other programming languages, e.g., pypcap[2] for Python. Thus, we also accept solutions written in Python.

Please do not use any third-party libraries (other than libpcap) in your implementation. If you are using C++, you can compile your solution as follows:

```
g++ -o pcap_task pcap_task.cc -lpcap
```

If you are using Python, you can compile it with

```
python pcap_task.py
```

Each of the subtasks builds upon the previous subtask. Hence, during this assignment, you can extend your solution and hand in a single version of your code for all libpcap tasks. However, when executing your solution, it should accept the path of the pcap file to read in as the first argument on the command line. Please print the solution to every subtask on *stdout*. Your documentation should include a short description of how you solved the libpcap tasks, along with your results.

[1] http://www.fukuda-lab.org/mawilab/v1.1/2018/11/06/20181106.html
[2] http://pypcap.readthedocs.io

2.1.1 Simple Statistics

You will now retrieve network packets within your application and identify the protocols in use for closer analysis. During this task, you can use predefined protocol headers that already come with the system, including files or scapy. Also, keep in mind that headers might be of variable size.

1. Count the number of packets in the network trace. For that, create the fundamental structure of your application to retrieve packets through libpcap and implement a counter to verify the general working.

2. Considering IPv4 only, count the number of packets, the number of IPs, and the amount of transmitted payload. Does the trace contain more network protocols? Identify all present network (layer 3) protocols and count the packets per protocol. Specifically for IPv4, parse the header to analyze the IP addresses and payload data.
   Caution: It might be reasonable to identify and handle IP fragmentation.

3. Considering IPv4 only, how many bytes of payloads are transmitted over TCP and UDP? Does the trace contain more transport protocols? Are there packets with inconsistency regarding their payload length? Identify all present transport (layer4) protocols over IPv4 and parse TCP/UDP packets to analyze the payload bytes. Are there any indications that someone is deliberately sending unusual packets?

4. Count the number of flows and identify the most chatty conversations on the network and transport layer. We define a flow as a tuple of source and destination, where the source initiates the communication. On the network layer, identify the IPv4 flows. On the transport layer, sources and destinations are not only identified by the IPv4 addresses but also by the corresponding TCP/UDP protocol and the port number. On both layers, count the flows and identify the flow with the highest amount of transmitted bytes on each level.

   The following communication with packets on layer 4 can be described as a single flow: **(1.2.3.4:1337, 5.6.7.8:80)**

   ```
   1.2.3.4:1337 -> 5.6.7.8:80   # The first occurrence of both addresses in
   the same packet; defines the flow direction
   5.6.7.8:80 -> 1.2.3.4:1337   # Counts to the same flow (response)
   ...
   1.2.3.4:1337 -> 5.6.7.8:80   # Counts to the same flow (request)
   ...
   5.6.7.8:80 -> 1.2.3.4:1337   # Counts to the same flow (response)
   ```

5. Identify HTTP requests that submit login credentials via basic HTTP Auth. For that, use the network trace `illauth.pcap` (see Moodle), which includes two login

attempts over HTTP. Then, implement a basic HTTP parser to identify authentication with HTTP Auth and print the used login credentials (base64 encoded string is sufficient).

2.1.2 Attack Detection

Now that you can parse and interpret network packets byte by byte, you will extract attack indications. This requires tracking connections and analyzing flows.

- **Scenario 1: Identifying IP addresses that did not finish the three-way TCP handshake**
  Sending the first SYN packet to initiate the TCP Handshake but not to finish with the ACK after receiving the SYN/ACK is a common way to scan for open ports or to launch an SYN-Flood attack. Print the top 5 IP addresses (ordered by flows) that show this behavior.
- **Scenario 2: Identifying IP addresses that receive an ICMP port unreachable packet in response to initiating a UDP communication.**
  ICMP port unreachable packets in response to a UDP packet indicate that the respective port is closed. This can indicate a network scan, in which an attacker scans ports of a target system. Print the top 5 IP addresses (ordered by flows) that follow this attack characteristic.

2.2 Tcpdump

One of the useful network analysis tools is tcpdump, combining power and simplicity into a single command-line interface.

1. Read the given pcap file with tcmdump and write the first 10000 packets to another capture file named sample.pcap. Then, verify the number of packets in the new capture.
2. List the top five talkers for a period of time using simple command line field extraction to get the IP address and sort and count the occurrences.
3. Use filters to list packets with TCP RST flags.
4. Only capture on HTTP data packets on port 80. Avoid capturing the TCP session setup (SYN / FIN / ACK) packets.
5. Capture and print first ten plaintext passwords.
6. Is it possible to identify port scans using tcpdump? If yes, show how such cases can be detected.

3 Intrusion Detection via Zeek (33.3%)

In the second first part of the tasksheet, you will learn about analyzing data traffic, evaluating Zeek log files, and detecting attacks via Zeek scripting. To start with, you can prepare the Zeek environment by executing the following commands on your Linux VM:

```
$ cd ~/Desktop
$ wget http://195.37.209.19/data/teaching/netsec/ids_setup.sh
$ chmod +x ./ids_setup.sh && ./ids_setup.sh
```

In production, Zeek is deployed in the network to capture live traffic. In the exercises here, however, you are replaying previously captured traffic. Fortunately, Zeek can easily read traffic captures (`pcap` files) instead of capturing traffic live. When running Zeek on the commandline[3], it writes log files into the current working directory. Thus, run Zeek in a folder where you want the log files to be stored. For the exercises in this tasksheet, you can run Zeek as follows:

```
zeek -r <pcap_file> local [<custom_script_files>]
```

The option `-r <pcap_file>` reads a pcap file from a given path. After that follows a list of Zeek scripts to be loaded. The name `local` is a special keyword and refers to the basic and default scripts. In addition, append the path to the custom scripts you want to execute.

3.1 Working with Logfiles

Zeek, by default, already produces several log files while analyzing the traffic. Please check the Zeek website to find out about the log files[4] that are created. Let Zeek analyze the pcap file `zeek_monitoring.pcap` that is provided in moodle together with this tasksheet. To answer the following questions, process the resulting log files with any tool or programming language you choose. However, we encourage you to use well-known Linux command line utilities such as `cat`, `awk`, `sort`, and `head` to practice your Linux skills.

With Zeek log files, you can easily get an overview of how the network traffic looks like. In particular, these might be some basic and frequently questions to be asked:

1. Which logfile reports summary statistics of TCP/UDP connections?
2. How many connections exist?
3. What are the top five connections with respect to their duration?
4. Find the source and destination IP address of all UDP and TCP connections that lasted more than 4 sec.

[3] `https://docs.zeek.org/en/lts/quickstart/index.html#zeek-as-a-command-line-utility`
[4] `https://docs.zeek.org/en/lts/script-reference/log-files.html`

5. Show the five destination ports that received the most network traffic, organized in descending order.

6. What are the top five connections with respect to the amount of transmitted bytes (in response)?

3.2 Detecting Malicious Communication by Zeek Scripting

Zeek comes with its own script language that allows to extend its functionality. In the following tasks, you have to write your scripts and load them in Zeek. Find a language tutorial[5] and the language reference[6] online.

For the following tasks, use the pcap file `zeek_detection.pcap` that is provided in Moodle together with this tasksheet. **Important**: This trace potentially contains real malware. **DO NOT** execute any files that you find in the pcap, i.e., transmitted files that you extract from the captured network trace.

Mirai is a botnet that received much attention in the media. The infected machines, i.e., bots, connect to a command and control (C&C) server to retrieve updates and instructions for attacks. By now, the protocol between the bots and their C&C server is known, including the structure of the exchanged messages. In this task, you make Zeek to detect a bot that is calling out to its C&C server. You do so by inspecting the network traffic for the malicious byte pattern. Code Listing 2 contains a Zeek snippet that you can use to implement the detection. Copy this snippet into a file, e.g., named `mirai.zeek`, and append it as a command line argument when running Zeek.

Hint: The `connection`[7] type contains the `conn_id`[8] type (named `id`), which itself contains the connection's identifying 4-tuple of IP addresses and ports. In contrast to other programming languages, access a member of an object using the $ operator.

```
event packet_contents(c: connection, contents: string)
    {
    # Mirai is over TCP only
    if (tcp != get_conn_transport_proto(c$id)) { return; }

    # TODO: Check port (destination port is telnet)

    # TODO: Check byte pattern
    if (contents[:] == "\xde\xad\xbe\xef") {
        print fmt("Detected the sample pattern for destination %s", "?");
        }
    }
```

Listing 2: Zeek snippet for the detection of Mirai

1. Based on the given code snippet, briefly explain how Zeek can generally check for a specific byte pattern.

[5] `https://docs.zeek.org/en/lts/examples/scripting/index.html`
[6] `https://docs.zeek.org/en/lts/script-reference/index.html`
[7] `https://docs.zeek.org/en/lts/scripts/base/init-bare.zeek.html#type-connection`
[8] `https://docs.zeek.org/en/lts/scripts/base/init-bare.zeek.html#type-conn_id`

2. Make the snippet *return* if the destination port number is not 23.
   Hint: As a port in Zeek encodes both a number and a protocol, use the function `port_to_count` [9] to retrieve the port number from a port object.

3. Replace the byte pattern to match to malicious byte sequence `\x00\x00\x00\x01`.

4. Upon finding a Mirai C&C connection, print the IP address of the C&C server.

[9] `https://docs.zeek.org/en/lts/scripts/base/bif/zeek.bif.zeek.html#id-port_to_count`