Testing iptables firewall rules with scapy

Comply with Cybersecurity Requirements like UNECE R-155
February 3, 2024
Simone Weiß and Michael Estner
Agenda

01 Who are we
02 Why test your firewall rules
03 Netfilter and iptables
04 Scapy
05 Firewall tests with scapy
06 Summary
Who are we

Simone Weiß

Embedded Systems Developer
Elektrobit – Driving the future of software

• Degree in computer science
• Worked mostly on embedded Linux distributions
• Cybersecurity (ISO 21434/UNECE)
• Private: Cats and nature

Michael Estner

Senior Software engineer
Elektrobit – Driving the future of software

• Degree in electrical engineering
• Embedded Linux & Python
• Private: MMA, hiking and cooking
We are Elektrobit

Your software solution provider

35 YEARS AUTOMOTIVE EXPERIENCE

10 YEARS OPEN-SOURCE EXPERIENCE

EMBEDDED LINUX SYSTEM WITH YOCTO

EB CORBOS LINUX BUILT ON UBUNTU

CYBERSECURITY MANAGEMENT SYSTEM (CSMS) COMPLIANT
Why test your firewall rules?

Cybersecurity requirements
Cybersecurity requirements

UN R155 and ISO 21434 – Cybersecurity maintenance

**UN R155**

Regulation on cybersecurity
- Mandatory in all UNECE member countries (64)
- Defines requirements for the cybersecurity management system (CSMS) in vehicles
- Ensures that cybersecurity practices and measures are adequately applied across the development process and life cycle of vehicles
- Applies to all software in vehicles

**ISO/SAE 21434**

Road vehicles – Cybersecurity engineering
- Requirements and recommendations to develop a cybersecurity product
- Baseline for CSMS
Introduction

Packet filtering

- Packet filter inspects traffic in the network stack

- Use cases
  - Firewall
  - Traffic statistics
  - Logging
  - ...

- Linux packet filter
  - Kernelspace
    - netfilter
  - Userspace
    - iptables, ip6tables, ebtables, arptables
    - nftables

Diagram:
- Application
- Packet filter
- Driver
- Ingressing
- Egressing
Netfilter

- Community-driven FOSS project
- Provides packet filtering and network address translation for the Linux kernel

Key components:
- Hooks: Intercept packets at different stages
- Tables: Has a specific packet-handling task
- Chains: Sequences of rules within a table
- Targets: Action when packet matches a rule
Ip(6)tables

- Userspace program to interact with the netfilter
- Organized in tables that contains several chains
- A chain is a list of rules which can match a set of packets
- A rule specifies what to do with a packet that matches
  - If so, the next rule is the one specified by the target: User-defined, or ACCEPT, DROP, QUEUE, or RETURN
  - If not, next rule is the next in the chain
Ip(6)tables

*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:DOCKER - [0:0]
:DOCKER-ISOLATION-STAGE-1 - [0:0]
:DOCKER-ISOLATION-STAGE-2 - [0:0]
:DOCKER-USER - [0:0]
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
COMMIT
*nat

**Chain FORWARD**
- Rule 1: All traffic is sent to the DOCKER-USER chain

**Chain DOCKER-USER**
- Rule 1: Target is return for all traffic

**Chain FORWARD**
- Rule 2: All traffic is sent to the DOCKER-ISOLATION-STAGE-1 chain

**Chain DOCKER-ISOLATION-STAGE-1**
- Rule 1: All traffic from docker0 interface to anywhere that is not itself is sent to the DOCKER-ISOLATION-STAGE-2 chain.
- Rule 2: Return all traffic

**Chain DOCKER-ISOLATION-STAGE-2**
- Rule 1: Drop all traffic to docker0
- Rule 2: Return else
### Ip(6)tables

*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:DOCKER - [0:0]
:DOCKER-ISOLATION-STAGE-1 - [0:0]
:DOCKER-ISOLATION-STAGE-2 - [0:0]
:DOCKER-USER - [0:0]
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN

COMMIT

*nat

---

- **Chain FORWARD**
  - Rule 1: All traffic is sent to the DOCKER-USER chain

- **Chain DOCKER-USER**
  - Rule 1: Target is return for all traffic

- **Chain FORWARD**
  - Rule 2: All traffic is sent to the DOCKER-ISOLATION-STAGE-1 chain

- **Chain DOCKER-ISOLATION-STAGE-1**
  - Rule 1: All traffic from docker0 interface to anywhere that is not itself is sent to the DOCKER-ISOLATION-STAGE-2 chain.
  - Rule 2: Return all traffic

- **Chain DOCKER-ISOLATION-STAGE-2**
  - Rule 1: Drop all traffic to docker0
  - Rule 2: Return else
### Chain FORWARD
- Rule 1: All traffic is sent to the DOCKER-USER chain

### Chain DOCKER-USER
- Rule 1: Target is return for all traffic

### Chain FORWARD
- Rule 2: All traffic is sent to the DOCKER-ISOLATION-STAGE-1 chain

### Chain DOCKER-ISOLATION-STAGE-1
- Rule 1: All traffic from docker0 interface to anywhere that is not itself is sent to the DOCKER-ISOLATION-STAGE-2 chain.
  - Rule 2: Return all traffic

### Chain DOCKER-ISOLATION-STAGE-2
- Rule 1: Drop all traffic to docker0
  - Rule 2: Return else

```
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:DOCKER - [0:0]
:DOCKER-ISOLATION-STAGE-1 - [0:0]
:DOCKER-ISOLATION-STAGE-2 - [0:0]
:DOCKER-USER - [0:0]
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
COMMIT
*nat
```
**Ip(6)tables**

• **Chain FORWARD**
  - Rule 1: All traffic is sent to the DOCKER-USER chain

• **Chain DOCKER-USER**
  - Rule 1: Target is return for all traffic

• **Chain FORWARD**
  - Rule 2: All traffic is sent to the DOCKER-ISOLATION-STAGE-1 chain

• **Chain DOCKER-ISOLATION-STAGE-1**
  - Rule 1: All traffic from docker0 interface to anywhere that is not itself is sent to the DOCKER-ISOLATION-STAGE-2 chain.
  - Rule 2: Return all traffic

• **Chain DOCKER-ISOLATION-STAGE-2**
  - Rule 1: Drop all traffic to docker0
  - Rule 2: Return else

```
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:DOCKER - [0:0]
:DOCKER-ISOLATION-STAGE-1 - [0:0]
:DOCKER-ISOLATION-STAGE-2 - [0:0]
:DOCKER-USER - [0:0]
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
COMMIT
*nat
```
• Chain **FORWARD**
  – Rule 1: All traffic is sent to the **DOCKER-USER** chain

• Chain **DOCKER-USER**
  – Rule 1: Target is return for all traffic

• Chain **FORWARD**
  – Rule 2: All traffic is sent to the **DOCKER-ISOLATION-STAGE-1** chain

• Chain **DOCKER-ISOLATION-STAGE-1**
  – Rule 1: All traffic from docker0 interface to anywhere that is not itself is sent to the **DOCKER-ISOLATION-STAGE-2** chain.
  – Rule 2: Return all traffic

• Chain **DOCKER-ISOLATION-STAGE-2**
  – Rule 1: Drop all traffic to docker0
  – Rule 2: Return else

```c
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:DOCKER - [0:0]
:DOCKER-ISOLATION-STAGE-1 - [0:0]
:DOCKER-ISOLATION-STAGE-2 - [0:0]
:DOCKER-USER - [0:0]
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
COMMIT
*nat
```
Toollandscape

- socat
- nemesis
- scapy
- netcat...
Why scapy?

- Python based interactive packet manipulating library
- With scapy you can define, send and receive complete custom packets
- You can manipulate across different layers
- Low barrier to create custom network packets
- Easy to integrate in the existing test eco system

**Transport Layer**
- TCP
- UDP

**Network Layer**
- IPV4
- IPV6

**Link Layer**
- Ethernet
Scapy and the netfilter

- Ingressing

Scapy and the netfilter

- Egressing

Scapy – Basic examples

TCP upon IP protocol with basic fields
• IP(src="1.2.3.4", dst="1.2.3.5")/TCP(dport=80, flags="S")

UDP upon IP with random sourceport
• IP(src=RandIP(), dst="1.2.3.4")/UDP(sport=RandShort(), dport=80, chksum=0xFFFF)

ICMP
• IP(dst="1.2.3.4") / ICMP(type=3, code=0)

Sending, Receiving
• Available at different layers, in loops...

Sniffing
• sniff(iface='eth3', filter = lambda s: s[TCP].flags == 18, prn = lambda x: x[IP].dst)
Craft a fitting package:

```python
packet = IP(ttl=8, dst="192.168.7.2")/TCP(dport=1234, flags=0x02)
```

Send it:

```python
send(packet, iface="tap0")
```

Sniff for it:

```python
sniff(iface="eth0", filter="tcp and port 1234", count=1, prn=packet1_check)
```

```python
def packet1_check(x):
    if x.ttl == 8 and x[TCP].flags == "S":
        print("accepted by FW")
    else:
        print("rejected by FW")
```
Bringing it all together...

Craft a fitting package:
```python
packet = IP(src="192.168.7.0", dst="192.168.7.2") / TCP(dport=22)
```

Send it:
```python
send(packet, iface="tap0")
```

Sniff for it:
```python
sniff(iface="eth0", filter="tcp and src 192.168.7.0, count=1, prn=packet2_check")
```

```python
def packet2_check(x):
    if x[TCP].dport == 22:
        print("accepted by FW")
    else:
        print("rejected by FW")
```
Bringing it all together...

Iptables –t nat –A OUTPUT –d 192.168.7.1/32 -o eth0 –p tcp --dport 100 –j REDIRECT --to-ports 60001

Craft a fitting package:

\[
\text{pkt} = \text{TCP(dport=100)}
\]

Send it:

\[
\text{s.setsockopt}(
    \text{socket.SOL_SOCKET, 25, str(}"\text{eth0}"
)\
\text{s.bind}(("192.168.7.2", 0))\
\text{s.sendto} (\text{bytes(pkt), (}"192.168.7.1", 0))
\]

Sniff for it:

\[
\text{sniff(iface="tap0", filter="tcp and port 60001", count="1", prn=packet3_check)}
\]

def packet3_check(x):
    if x[TCP].dport == 60001:
        print("accepted by FW")
    else:
        print("rejected by FW")
Test firewall rules - Demo
<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why you have to test your firewall rules</td>
</tr>
<tr>
<td>Netfilter basics</td>
</tr>
<tr>
<td>Ip(6)tables overview</td>
</tr>
<tr>
<td>Toollandscape for network testing</td>
</tr>
<tr>
<td>Scapy usage</td>
</tr>
<tr>
<td>Test iptables firewall rules</td>
</tr>
</tbody>
</table>
Questions?

Simone Weiß
Embedded Systems Developer, EB-EST-CMS-P-1
Elektrobit – Driving the future of software
simone.weiss@elektrobit.com
elektrobit.com

Michael Estner
Senior Software Engineer, EB-EST-CMS-P-1
Elektrobit – Driving the future of software
michael.estner@elektrobit.com
elektrobit.com