Red Hat Enterprise Linux 10-beta

Configuring firewalls and packet filters

Managing the firewalld service, the nftables framework, and XDP packet filtering features

Last Updated: 2025-03-21

Red Hat Enterprise Linux 10-beta Configuring firewalls and packet filters

Managing the firewalld service, the nftables framework, and XDP packet filtering features

Legal Notice

Abstract

Packet filters, such as firewalls, use rules to control incoming, outgoing, and forwarded network traffic. In Red Hat Enterprise Linux (RHEL), you can use the `firewalld` service and the `nftables` framework to filter network traffic and build performance-critical firewalls. You can also use the Express Data Path (XDP) feature of the kernel to process or drop network packets at the network interface at a very high rate.

Table of Contents

RHEL BETA RELEASE

Red Hat provides Red Hat Enterprise Linux Beta access to all subscribed Red Hat accounts. The purpose of Beta access is to:

- Provide an opportunity to customers to test major features and capabilities prior to the general availability release and provide feedback or report issues.
- Provide Beta product documentation as a preview. Beta product documentation is under development and is subject to substantial change.

Note that Red Hat does not support the usage of RHEL Beta releases in production use cases. For more information, see the Red Hat Knowledgebase solution What does Beta mean in Red Hat Enterprise Linux and can I upgrade a RHEL Beta installation to a General Availability (GA) release?.

CHAPTER 1. USING AND CONFIGURING FIREWALLD

A firewall is a way to protect machines from any unwanted traffic from outside. It enables users to control incoming network traffic on host machines by defining a set of *firewall rules*. These rules are used to sort the incoming traffic and either block it or allow through.

firewalld is a firewall service daemon that provides a dynamic customizable host-based firewall with a D-Bus interface. Being dynamic, it enables creating, changing, and deleting the rules without the necessity to restart the firewall daemon each time the rules are changed.

firewalld uses the concepts of zones and services, that simplify the traffic management. Zones are predefined sets of rules. Network interfaces and sources can be assigned to a zone. The traffic allowed depends on the network your computer is connected to and the security level this network is assigned. Firewall services are predefined rules that cover all necessary settings to allow incoming traffic for a specific service and they apply within a zone.

Services use one or more ports or addresses for network communication. Firewalls filter communication based on ports. To allow network traffic for a service, its ports must be open. **firewalld** blocks all traffic on ports that are not explicitly set as open. Some zones, such as trusted, allow all traffic by default.

Note that **firewalld** with **nftables** backend does not support passing custom **nftables** rules to **firewalld**, using the **--direct** option.

1.1. CONFIGURING FLOWTABLE IN NFTABLES BY USING FIREWALLD

The **firewalld** utility now supports configuration of the software fastpath **flowtable** feature provided by the **nftables** utility. **flowtable** uses the connection tracking feature in the **Netfilter** framework and **nftables** to manage traffic rules. This collectively improves packet forwarding performance of established connections. By default, packet forwarding inside a zone is allowed. **firewalld** by default provides two network zones: **public** and **internal** for public and private networks respectively.

Procedure

1. Assign a network interface to the **internal** zone:

```
# firewall-cmd --permanent --zone=internal --add-interface=enp7s0
```

2. Assign a second network interface to the **internal** zone:

```
# firewall-cmd --permanent --zone=internal --add-interface=enp7s1
```

3. Put in the **/etc/firewalld/firewalld.conf** file the following line, to enable flowtable support on network interfaces:

```
NftablesFlowtable=enp7s0 enp7s1
```

4. Reload **firewalld** for rule changes to take effect:

```
# firewall-cmd --reload
```

Verification

- Check the configuration to ensure that the zones and rules are correctly applied:

```
# nft list chain inet firewalld filter_FORWARD

table inet firewalld {
    flowtable fastpath {
            hook ingress priority filter + 10
            devices = { enp7s0, enp7s1 }
    }
...
   chain filter_FORWARD {
    type filter hook forward priority filter + 10; policy accept;
    ct state { established, related } meta l4proto { tcp, udp } flow add @fastpath  <--- new rule
    ct state { established, related } accept
 }
...
```

Additional resources

- The **firewall-cmd(1)** man page

1.2. CONFIGURING ZONE PRIORITIES FOR TRAFFIC CLASSIFICATION BY USING FIREWALLD

With zone priorities, you can control the packet classification order by specifying priorities for **ingress** and **egress** traffic. The benefit is that you can specify the traffic classification order in a zone. So, zone A may be considered before zone B regardless of the source address or interfaces. A zone of a lower priority value has higher precedence over a zone with a higher priority value. This classification has a pair of **ingress** priority value and **egress** priority value.

1.2.1. Setting same priority value for both traffic types in a zone

By using the **--set-priority** option, you can set a common value for both **ingress** and **egress** traffic classification without explicit specification.

Prerequisites

1. Create a new zone:

```
# firewall-cmd --permanent --new-zone=example-zone
```

2. Set a common zone priority value for the **example-zone** zone with **--set-priority**:

```
# firewall-cmd --permanent --zone example-zone --set-priority -10
```

   By setting a lower value ensures the higher precedence. This ensures that all configured operations for both traffic types in this zone will take precedence over operations from other zones.

3. Apply permanent configuration to runtime:

```
# firewall-cmd --reload
```

Verification

- Display the priority value for both traffic types:

```
# firewall-cmd --permanent --info-zone example-zone

example-zone
  target: default
  ingress-priority: -10
  egress-priority: -10
  ...
  icmp-block-inversion: no
  ...
  services: dhcpv6-client mdns samba-client ssh
  ...
  forward: yes
  masquerade: no
  ...
```

This setting ensures that the traffic will be considered for classification into the **example-zone** before other zones.

1.2.2. Setting different priority value for each traffic type in a zone

By setting distinct values for **ingress** and **egress** traffic, you can set priorities for the traffic classification in a zone.

Procedure

1. Create a new zone:

```
# firewall-cmd --permanent --new-zone=example-zone
```

2. Set a zone priority value for **ingress** traffic in the **example-zone** zone with **--set-ingress-priority**:

```
# firewall-cmd --permanent --zone example-zone --set-ingress-priority -10
```

3. Set a zone priority value for **egress** traffic in the **example-zone** zone with **--set-egress-priority**:

```
# firewall-cmd --permanent --zone example-zone --set-egress-priority 100
```

4. Apply permanent configuration to runtime:

```
# firewall-cmd --reload
```

Verification

- Display the priority value for both traffic types:

```
# firewall-cmd --permanent --info-zone example-zone

example-zone (active)
  target: default
```

```
ingress-priority: -10
egress-priority: 100
icmp-block-inversion: no
interfaces: eth0
...
services: dhcpv6-client mdns samba-client ssh
...
forward: yes
masquerade: no
...
```

These values indicate that the **ingress** traffic has priority over the **egress** traffic in the **example-zone** zone before other zones.

CHAPTER 2. UNDERSTANDING THE EBPF NETWORKING FEATURES IN RHEL 10

The extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine that allows code execution in the kernel space. This code runs in a restricted sandbox environment with access only to a limited set of functions.

In networking, you can use eBPF to complement or replace kernel packet processing. Depending on the hook you use, eBPF programs have, for example:

- Read and write access to packet data and metadata
- Can look up sockets and routes
- Can set socket options
- Can redirect packets

2.1. OVERVIEW OF NETWORKING EBPF FEATURES IN RHEL 10

You can attach extended Berkeley Packet Filter (eBPF) networking programs to the following hooks in RHEL:

- eXpress Data Path (XDP): Provides early access to received packets before the kernel networking stack processes them.
- **tc** eBPF classifier with a direct-action flag: Provides powerful packet processing on ingress and egress. Programs can be attached as an eBPF classifier with a direct-action flag in the **qdisc** hierarchy, or using the link-based **tcx** API.
- Control Groups version 2 (cgroup v2): Enables filtering and overriding socket-based operations performed by programs in a control group.
- Socket filtering: Enables filtering of packets received from sockets. This feature was also available in the classic Berkeley Packet Filter (cBPF), but has been extended to support eBPF programs.
- Stream parser: Enables splitting up streams to individual messages, filtering, and redirecting them to sockets.
- **SO_REUSEPORT** socket selection: Provides a programmable selection of a receiving socket from a **reuseport** socket group.
- Flow dissector: Enables overriding the way the kernel parses packet headers in certain situations.
- TCP congestion control callbacks: Enables implementing a custom TCP congestion control algorithm.
- Routes with encapsulation: Enables creating custom tunnel encapsulation.

XDP

You can attach programs of the **BPF_PROG_TYPE_XDP** type to a network interface. The kernel then executes the program on received packets before the kernel network stack starts processing them. This allows fast packet forwarding in certain situations, such as fast packet dropping to prevent distributed denial of service (DDoS) attacks and fast packet redirects for load balancing scenarios.

You can also use XDP for different forms of packet monitoring and sampling. The kernel allows XDP programs to modify packets and to pass them for further processing to the kernel network stack.

The following XDP modes are available:

- Native (driver) XDP: The kernel executes the program from the earliest possible point during packet reception. At this moment, the kernel did not parse the packet and, therefore, no metadata provided by the kernel is available. This mode requires that the network interface driver supports XDP but not all drivers support this native mode.
- Generic XDP: The kernel network stack executes the XDP program early in the processing. At that time, kernel data structures have been allocated, and the packet has been pre-processed. If a packet should be dropped or redirected, it requires a significant overhead compared to the native mode. However, the generic mode does not require network interface driver support and works with all network interfaces.
- Offloaded XDP: The kernel executes the XDP program on the network interface instead of on the host CPU. Note that this requires specific hardware, and only certain eBPF features are available in this mode.

On RHEL, load all XDP programs using the **libxdp** library. This library enables system-controlled usage of XDP.

NOTE

Currently, there are some system configuration limitations for XDP programs. For example, you must disable certain hardware offload features on the receiving interface. Additionally, not all features are available with all drivers that support the native mode.

In RHEL 10, Red Hat supports the XDP features only if you use the **libxdp** library to load the program into the kernel.

AF_XDP

Using an XDP program that filters and redirects packets to a given **AF_XDP** socket, you can use one or more sockets from the **AF_XDP** protocol family to quickly copy packets from the kernel to the user space.

Traffic Control

The Traffic Control (**tc**) subsystem offers the following types of eBPF programs:

- **BPF_PROG_TYPE_SCHED_CLS**
- **BPF_PROG_TYPE_SCHED_ACT**

These types enable you to write custom **tc** classifiers and **tc** actions in eBPF. Together with the parts of the **tc** ecosystem, this provides the ability for powerful packet processing and is the core part of several container networking orchestration solutions.

In most cases, only the classifier is used, as with the direct-action flag, the eBPF classifier can execute actions directly from the same eBPF program. The **clsact** Queueing Discipline (**qdisc**) has been designed to enable this on the ingress side.

Note that using a flow dissector eBPF program can influence operation of some other **qdiscs** and **tc** classifiers, such as **flower**.

The link-based **tcx** API is provided along with the **qdisc** API. It enables your applications to maintain ownership over a BPF program to prevent accidental removal of the BPF program. Also, the **tcx** API has multiprogram support that allows multiple applications to attach BPF programs in the **tc** layer in parallel.

Socket filter

Several utilities use or have used the classic Berkeley Packet Filter (cBPF) for filtering packets received on a socket. For example, the **tcpdump** utility enables the user to specify expressions, which **tcpdump** then translates into cBPF code.

As an alternative to cBPF, the kernel allows eBPF programs of the **BPF_PROG_TYPE_SOCKET_FILTER** type for the same purpose.

Control Groups

In RHEL, you can use multiple types of eBPF programs that you can attach to a cgroup. The kernel executes these programs when a program in the given cgroup performs an operation. Note that you can use only cgroups version 2.

The following networking-related cgroup eBPF programs are available in RHEL:

- **BPF_PROG_TYPE_SOCK_OPS**: The kernel calls this program on various TCP events. The program can adjust the behavior of the kernel TCP stack, including custom TCP header options, and so on.
- **BPF_PROG_TYPE_CGROUP_SOCK_ADDR**: The kernel calls this program during **connect**, **bind**, **sendto**, **recvmsg**, **getpeername**, and **getsockname** operations. This program allows changing IP addresses and ports. This is useful when you implement socket-based network address translation (NAT) in eBPF.
- **BPF_PROG_TYPE_CGROUP_SOCKOPT**: The kernel calls this program during **setsockopt** and **getsockopt** operations and allows changing the options.
- **BPF_PROG_TYPE_CGROUP_SOCK**: The kernel calls this program during socket creation, socket releasing, and binding to addresses. You can use these programs to allow or deny the operation, or only to inspect socket creation for statistics.
- **BPF_PROG_TYPE_CGROUP_SKB**: This program filters individual packets on ingress and egress, and can accept or reject packets.

Stream Parser

A stream parser operates on a group of sockets that are added to a special eBPF map. The eBPF program then processes packets that the kernel receives or sends on those sockets.

The following stream parser eBPF programs are available in RHEL:

- **BPF_PROG_TYPE_SK_SKB**: An eBPF program parses packets received on the socket into individual messages, and instructs the kernel to drop those messages, accept them, or send them to another socket.
- **BPF_PROG_TYPE_SK_MSG**: This program filters egress messages. An eBPF program parses the packets and either approves or rejects them.

SO_REUSEPORT socket selection

Using this socket option, you can bind multiple sockets to the same IP address and port. Without eBPF, the kernel selects the receiving socket based on a connection hash. With the **BPF_PROG_TYPE_SK_REUSEPORT** program, the selection of the receiving socket is fully programmable.

Flow dissector

When the kernel needs to process packet headers without going through the full protocol decode, they are **dissected**. For example, this happens in the **tc** subsystem, in multipath routing, in bonding, or when calculating a packet hash. In this situation the kernel parses the packet headers and fills internal structures with the information from the packet headers. You can replace this internal parsing using the **BPF_PROG_TYPE_FLOW_DISSECTOR** program. Note that you can only dissect TCP and UDP over IPv4 and IPv6 in eBPF in RHEL.

TCP Congestion Control

You can write a custom TCP congestion control algorithm using a group of **BPF_PROG_TYPE_STRUCT_OPS** programs that implement **struct tcp_congestion_oops** callbacks. An algorithm that is implemented this way is available to the system alongside the built-in kernel algorithms.

Routes with encapsulation

You can attach one of the following eBPF program types to routes in the routing table as a tunnel encapsulation attribute:

- **BPF_PROG_TYPE_LWT_IN**
- **BPF_PROG_TYPE_LWT_OUT**
- **BPF_PROG_TYPE_LWT_XMIT**
- **LWT_SEG6LOCAL** (Technology Preview)

The functionality of such an eBPF program is limited to specific tunnel configurations and does not allow creating a generic encapsulation or decapsulation solution.

Socket lookup

To bypass limitations of the **bind** system call, use an eBPF program of the **BPF_PROG_TYPE_SK_LOOKUP** type. Such programs can select a listening socket for new incoming TCP connections or an unconnected socket for UDP packets.

2.2. OVERVIEW OF XDP FEATURES IN RHEL 10 BY NETWORK CARDS

The following is an overview of XDP-enabled network cards and the XDP features you can use with them:

Network card	Driver	Basic	Redirect	Target	HW offload	Zero-copy	Large MTU
Amazon Elastic Network Adapter	**ena**	yes	yes	yes [a]	no	no	no
aQuantia AQtion Ethernet card	**atlantic**	yes	yes	yes	no	no	yes
Broadcom NetXtreme-C/E 10/25/40/50 gigabit Ethernet	**bnxt_en**	yes	yes	yes [a]	no	no	yes
Cavium Thunder Virtual function	**nicvf**	yes	no	no	no	no	no

Network card	Driver	Basic	Redirect	Target	HW offload	Zero-copy	Large MTU
Freescale DPAA2 Ethernet	**fsl-dpaa2-eth**	yes	yes	yes	no	yes [b]	no
Google Virtual NIC (gVNIC) support	**gve**	yes	yes	yes	no	yes	no
Intel® 10GbE PCI Express Virtual Function Ethernet	**ixgbevf**	yes	no	no	no	no	no
Intel® 10GbE PCI Express adapters	**ixgbe**	yes	yes	yes [a]	no	yes	yes [c]
Intel® Ethernet Connection E800 Series	**ice**	yes	yes	yes [a]	no	yes	yes
Intel® Ethernet Controller I225-LM/I225-V family	**igc**	yes	yes	yes [a]	no	yes	yes [c]
Intel® PCI Express Gigabit adapters	**igb**	yes	yes	yes [a]	no	no	yes [c]
Intel® Ethernet Controller XL710 Family	**i40e**	yes	yes	yes [a] [d]	no	yes	no
Marvell OcteonTX2	**rvu_nicpf**	yes	yes	yes [a] [d]	no	no	no
Mellanox 5th generation network adapters (ConnectX series)	**mlx5_core**	yes	yes	yes [d]	no	yes	yes
Mellanox Technologies 1/10/40Gbit Ethernet	**mlx4_en**	yes	yes	no	no	no	no
Microsoft Azure Network Adapter	**mana**	yes	yes	yes	no	no	no
Microsoft Hyper-V virtual network	**hv_netvsc**	yes	yes	yes	no	no	no
Netronome® NFP4000/NFP6000 NIC [e]	**nfp**	yes	yes	no	yes	yes	no
NXP ENETC Gigabit Ethernet	**fsl-enetc**	yes	yes	yes	no	no	yes

Network card	Driver	Basic	Redirect	Target	HW offload	Zero-copy	Large MTU
NXP Fast Ethernet Controller	**fec**	yes	yes	yes [a]	no	no	no
Pensando Ethernet Adapter	**ionic**	yes	yes	yes	no	no	yes
QEMU Virtio network	**virtio_net**	yes	yes	yes [a]	no	no	yes
QLogic QED 25/40/100Gb Ethernet NIC	**qede**	yes	yes	yes	no	no	no
QorIQ DPAA Ethernet	**fsl_dpa**	yes	yes	yes	no	no	no
STMicroelectronics Multi-Gigabit Ethernet	**stmmac**	yes	yes	yes	no	yes	no
Solarflare SFC9000/SFC9100/EF100-family	**sfc**	yes	yes	yes [d]	no	no	no
Universal TUN/TAP device	**tun**	yes	yes	yes	no	no	no
Virtual Ethernet pair device	**veth**	yes	yes	yes	no	no	yes
VMware VMXNET3 ethernet driver	**vmxnet3**	yes	yes	yes [a] [d]	no	no	no
Xen paravirtual network device	**xen-netfront**	yes	yes	yes	no	no	no

[a] Only if an XDP program is loaded on the interface.

[b] Not available on all hardware revisions.

[c] Transmitting side only. Cannot receive large packets through XDP.

[d] Requires several XDP TX queues for every CPU. For example, the number of queues must be larger than or equal to the largest CPU index.

[e] Some of the listed features are not available for the Netronome® NFP3800 NIC.

Legend:

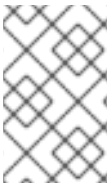
- Basic: Supports basic return codes: **DROP**, **PASS**, **ABORTED**, and **TX**.
- Redirect: Supports the **XDP_REDIRECT** return code.
- Target: Can be a target of a **XDP_REDIRECT** return code.

- HW offload: Supports XDP hardware offload.
- Zero-copy: Supports the zero-copy mode for the **AF_XDP** protocol family.
- Large MTU: Supports packets larger than page size.

CHAPTER 3. NETWORK TRACING USING THE BPF COMPILER COLLECTION

BPF Compiler Collection (BCC) is a library, which facilitates the creation of the extended Berkeley Packet Filter (eBPF) programs. The main utility of eBPF programs is analyzing the operating system performance and network performance without experiencing overhead or security issues.

BCC removes the need for users to know deep technical details of eBPF, and provides many out-of-the-box starting points, such as the **bcc-tools** package with pre-created eBPF programs.



NOTE

The eBPF programs are triggered on events, such as disk I/O, TCP connections, and process creations. It is unlikely that the programs should cause the kernel to crash, loop or become unresponsive because they run in a safe virtual machine in the kernel.

3.1. INSTALLING THE BCC-TOOLS PACKAGE

Install the **bcc-tools** package, which also installs the BPF Compiler Collection (BCC) library as a dependency.

Procedure

- Install **bcc-tools**:

```
# dnf install bcc-tools
```

The BCC tools are installed in the **/usr/share/bcc/tools/** directory.

Verification

- Inspect the installed tools:

```
# ls -l /usr/share/bcc/tools/
...
-rwxr-xr-x. 1 root root  4198 Dec 14 17:53 dcsnoop
-rwxr-xr-x. 1 root root  3931 Dec 14 17:53 dcstat
-rwxr-xr-x. 1 root root 20040 Dec 14 17:53 deadlock_detector
-rw-r--r--. 1 root root  7105 Dec 14 17:53 deadlock_detector.c
drwxr-xr-x. 3 root root  8192 Mar 11 10:28 doc
-rwxr-xr-x. 1 root root  7588 Dec 14 17:53 execsnoop
-rwxr-xr-x. 1 root root  6373 Dec 14 17:53 ext4dist
-rwxr-xr-x. 1 root root 10401 Dec 14 17:53 ext4slower
...
```

The **doc** directory in the listing above contains documentation for each tool.

3.2. DISPLAYING TCP CONNECTIONS ADDED TO THE KERNEL'S ACCEPT QUEUE

After the kernel receives the **ACK** packet in a TCP 3-way handshake, the kernel moves the connection from the **SYN** queue to the **accept** queue after the connection's state changes to **ESTABLISHED**. Therefore, only successful TCP connections are visible in this queue.

The **tcpaccept** utility uses eBPF features to display all connections the kernel adds to the **accept** queue. The utility is lightweight because it traces the **accept()** function of the kernel instead of capturing packets and filtering them. For example, use **tcpaccept** for general troubleshooting to display new connections the server has accepted.

Procedure

1. Enter the following command to start the tracing the kernel **accept** queue:

```
# /usr/share/bcc/tools/tcpaccept
PID   COMM      IP RADDR         RPORT  LADDR    LPORT
843   sshd      4  192.0.2.17    50598  192.0.2.1  22
1107  ns-slapd  4  198.51.100.6  38772  192.0.2.1  389
1107  ns-slapd  4  203.0.113.85  38774  192.0.2.1  389
...
```

   Each time the kernel accepts a connection, **tcpaccept** displays the details of the connections.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpaccept(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpaccept_example.txt** file

3.3. TRACING OUTGOING TCP CONNECTION ATTEMPTS

The **tcpconnect** utility uses eBPF features to trace outgoing TCP connection attempts. The output of the utility also includes connections that failed.

The **tcpconnect** utility is lightweight because it traces, for example, the **connect()** function of the kernel instead of capturing packets and filtering them.

Procedure

1. Enter the following command to start the tracing process that displays all outgoing connections:

```
# /usr/share/bcc/tools/tcpconnect
PID    COMM         IP SADDR      DADDR          DPORT
31346  curl         4  192.0.2.1  198.51.100.16  80
31348  telnet       4  192.0.2.1  203.0.113.231  23
31361  isc-worker00 4  192.0.2.1  192.0.2.254    53
...
```

   Each time the kernel processes an outgoing connection, **tcpconnect** displays the details of the connections.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpconnect(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpconnect_example.txt** file

3.4. MEASURING THE LATENCY OF OUTGOING TCP CONNECTIONS

The TCP connection latency is the time taken to establish a connection. This typically involves the kernel TCP/IP processing and network round trip time, and not the application runtime.

The **tcpconnlat** utility uses eBPF features to measure the time between a sent **SYN** packet and the received response packet.

Procedure

1. Start measuring the latency of outgoing connections:

```
# /usr/share/bcc/tools/tcpconnlat
PID    COMM         IP SADDR      DADDR          DPORT LAT(ms)
32151  isc-worker00 4  192.0.2.1  192.0.2.254    53    0.60
32155  ssh          4  192.0.2.1  203.0.113.190  22    26.34
32319  curl         4  192.0.2.1  198.51.100.59  443   188.96
...
```

   Each time the kernel processes an outgoing connection, **tcpconnlat** displays the details of the connection after the kernel receives the response packet.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpconnlat(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpconnlat_example.txt** file

3.5. DISPLAYING DETAILS ABOUT TCP PACKETS AND SEGMENTS THAT WERE DROPPED BY THE KERNEL

The **tcpdrop** utility enables administrators to display details about TCP packets and segments that were dropped by the kernel. Use this utility to debug high rates of dropped packets that can cause the remote system to send timer-based retransmits. High rates of dropped packets and segments can impact the performance of a server.

Instead of capturing and filtering packets, which is resource-intensive, the **tcpdrop** utility uses eBPF features to retrieve the information directly from the kernel.

Procedure

1. Enter the following command to start displaying details about dropped TCP packets and segments:

```
# /usr/share/bcc/tools/tcpdrop
TIME     PID    IP SADDR:SPORT      > DADDR:DPORT   STATE (FLAGS)
```

```
13:28:39 32253  4  192.0.2.85:51616  > 192.0.2.1:22  CLOSE_WAIT (FIN|ACK)
 b'tcp_drop+0x1'
 b'tcp_data_queue+0x2b9'
 ...

13:28:39 1      4  192.0.2.85:51616  > 192.0.2.1:22   CLOSE (ACK)
 b'tcp_drop+0x1'
 b'tcp_rcv_state_process+0xe2'
 ...
```

Each time the kernel drops TCP packets and segments, **tcpdrop** displays the details of the connection, including the kernel stack trace that led to the dropped package.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpdrop(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpdrop_example.txt** file

3.6. TRACING TCP SESSIONS

The **tcplife** utility uses eBPF to trace TCP sessions that open and close, and prints a line of output to summarize each one. Administrators can use **tcplife** to identify connections and the amount of transferred traffic.

For example, you can display connections to port **22** (SSH) to retrieve the following information:

- The local process ID (PID)
- The local process name
- The local IP address and port number
- The remote IP address and port number
- The amount of received and transmitted traffic in KB.
- The time in milliseconds the connection was active

Procedure

1. Enter the following command to start the tracing of connections to the local port **22**:

```
# /usr/share/bcc/tools/tcplife -L 22
PID   COMM    LADDR      LPORT RADDR       RPORT TX_KB  RX_KB      MS
19392 sshd    192.0.2.1  22    192.0.2.17  43892    53     52 6681.95
19431 sshd    192.0.2.1  22    192.0.2.245 43902    81 249381 7585.09
19487 sshd    192.0.2.1  22    192.0.2.121 43970  6998      7 16740.35
...
```

Each time a connection is closed, **tcplife** displays the details of the connections.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcplife(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcplife_example.txt** file

3.7. TRACING TCP RETRANSMISSIONS

The **tcpretrans** utility displays details about TCP retransmissions, such as the local and remote IP address and port number, as well as the TCP state at the time of the retransmissions.

The utility uses eBPF features and, therefore, has a very low overhead.

Procedure

1. Use the following command to start displaying TCP retransmission details:

```
# /usr/share/bcc/tools/tcpretrans
TIME     PID  IP LADDR:LPORT   T> RADDR:RPORT        STATE
00:23:02 0    4  192.0.2.1:22  R> 198.51.100.0:26788  ESTABLISHED
00:23:02 0    4  192.0.2.1:22  R> 198.51.100.0:26788  ESTABLISHED
00:45:43 0    4  192.0.2.1:22  R> 198.51.100.0:17634  ESTABLISHED
...
```

   Each time the kernel calls the TCP retransmit function, **tcpretrans** displays the details of the connection.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpretrans(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpretrans_example.txt** file

3.8. DISPLAYING TCP STATE CHANGE INFORMATION

During a TCP session, the TCP state changes. The **tcpstates** utility uses eBPF functions to trace these state changes, and prints details including the duration in each state. For example, use **tcpstates** to identify if connections spend too much time in the initialization state.

Procedure

1. Use the following command to start tracing TCP state changes:

```
# /usr/share/bcc/tools/tcpstates
SKADDR           C-PID C-COMM     LADDR     LPORT RADDR       RPORT OLDSTATE    ->
NEWSTATE    MS
ffff9cd377b3af80 0     swapper/1  0.0.0.0   22    0.0.0.0     0     LISTEN      -> SYN_RECV
0.000
ffff9cd377b3af80 0     swapper/1  192.0.2.1 22    192.0.2.45  53152 SYN_RECV    ->
ESTABLISHED 0.067
ffff9cd377b3af80 818   sssd_nss   192.0.2.1 22    192.0.2.45  53152 ESTABLISHED ->
CLOSE_WAIT  65636.773
```

```
ffff9cd377b3af80 1432  sshd       192.0.2.1 22    192.0.2.45  53152 CLOSE_WAIT  ->
LAST_ACK    24.409
ffff9cd377b3af80 1267  pulseaudio 192.0.2.1 22    192.0.2.45  53152 LAST_ACK    ->
CLOSE       0.376
...
```

Each time a connection changes its state, **tcpstates** displays a new line with updated connection details.

If multiple connections change their state at the same time, use the socket address in the first column (**SKADDR**) to determine which entries belong to the same connection.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpstates(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpstates_example.txt** file

3.9. SUMMARIZING AND AGGREGATING TCP TRAFFIC SENT TO SPECIFIC SUBNETS

The **tcpsubnet** utility summarizes and aggregates IPv4 TCP traffic that the local host sends to subnets and displays the output on a fixed interval. The utility uses eBPF features to collect and summarize the data to reduce the overhead.

By default, **tcpsubnet** summarizes traffic for the following subnets:

- **127.0.0.1/32**
- **10.0.0.0/8**
- **172.16.0.0/12**
- **192.0.2.0/24/16**
- **0.0.0.0/0**

Note that the last subnet (**0.0.0.0/0**) is a catch-all option. The **tcpsubnet** utility counts all traffic for subnets different than the first four in this catch-all entry.

Follow the procedure to count the traffic for the **192.0.2.0/24** and **198.51.100.0/24** subnets. Traffic to other subnets will be tracked in the **0.0.0.0/0** catch-all subnet entry.

Procedure

1. Start monitoring the amount of traffic sent to the **192.0.2.0/24**, **198.51.100.0/24**, and other subnets:

```
# /usr/share/bcc/tools/tcpsubnet 192.0.2.0/24,198.51.100.0/24,0.0.0.0/0
Tracing... Output every 1 secs. Hit Ctrl-C to end
[02/21/20 10:04:50]
192.0.2.0/24          856
198.51.100.0/24       7467
```

```
[02/21/20 10:04:51]
192.0.2.0/24          1200
198.51.100.0/24       8763
0.0.0.0/0             673
...
```

This command displays the traffic in bytes for the specified subnets once per second.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcpsubnet(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcpsubnet.txt** file

3.10. DISPLAYING THE NETWORK THROUGHPUT BY IP ADDRESS AND PORT

The **tcptop** utility displays TCP traffic the host sends and receives in kilobytes. The report automatically refreshes and contains only active TCP connections. The utility uses eBPF features and, therefore, has only a very low overhead.

Procedure

1. To monitor the sent and received traffic, enter:

```
# /usr/share/bcc/tools/tcptop
13:46:29 loadavg: 0.10 0.03 0.01 1/215 3875

PID    COMM        LADDR          RADDR             RX_KB   TX_KB
3853   3853        192.0.2.1:22   192.0.2.165:41838 32      102626
1285   sshd        192.0.2.1:22   192.0.2.45:39240  0       0
...
```

The output of the command includes only active TCP connections. If the local or remote system closes a connection, the connection is no longer visible in the output.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcptop(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcptop.txt** file

3.11. TRACING ESTABLISHED TCP CONNECTIONS

The **tcptracer** utility traces the kernel functions that connect, accept, and close TCP connections. The utility uses eBPF features and, therefore, has a very low overhead.

Procedure

1. Use the following command to start the tracing process:

```
# /usr/share/bcc/tools/tcptracer
Tracing TCP established connections. Ctrl-C to end.
T  PID    COMM         IP SADDR         DADDR         SPORT  DPORT
A  1088   ns-slapd     4  192.0.2.153   192.0.2.1     0      65535
A  845    sshd         4  192.0.2.1     192.0.2.67    22     42302
X  4502   sshd         4  192.0.2.1     192.0.2.67    22     42302
...
```

Each time the kernel connects, accepts, or closes a connection, **tcptracer** displays the details of the connections.

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **tcptracer(8)** man page on your system
- **/usr/share/bcc/tools/doc/tcptracer_example.txt** file

3.12. TRACING IPV4 AND IPV6 LISTEN ATTEMPTS

The **solisten** utility traces all IPv4 and IPv6 listen attempts. It traces the attempts including which ultimately fail or the listening program that does not accept the connection. The utility traces functions that the kernel calls when a program wants to listen for TCP connections.

Procedure

1. Enter the following command to start the tracing process that displays all listen TCP attempts:

```
# /usr/share/bcc/tools/solisten
PID    COMM          PROTO       BACKLOG     PORT     ADDR
3643   nc            TCPv4       1           4242     0.0.0.0
3659   nc            TCPv6       1           4242     2001:db8:1::1
4221   redis-server  TCPv6       128         6379     ::
4221   redis-server  TCPv4       128         6379     0.0.0.0
....
```

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **solisten(9)** man page on your system
- **/usr/share/bcc/tools/doc/solisten_example.txt** file

3.13. SUMMARIZING THE SERVICE TIME OF SOFT INTERRUPTS

The **softirqs** utility summarizes the time spent servicing soft interrupts (soft IRQs) and shows this time as either totals or histogram distributions. This utility uses the **irq:softirq_enter** and **irq:softirq_exit** kernel tracepoints, which is a stable tracing mechanism.

Procedure

1. Enter the following command to start the tracing **soft irq** event time:

```
# /usr/share/bcc/tools/softirqs
Tracing soft irq event time... Hit Ctrl-C to end.
^C
SOFTIRQ          TOTAL_usecs
tasklet                  166
block                   9152
net_rx                 12829
rcu                    53140
sched                 182360
timer                 306256
```

2. Press **Ctrl**+**C** to stop the tracing process.

Additional resources

- **softirqs(8)** and **mpstat(1)** man pages on your system
- **/usr/share/bcc/tools/doc/softirqs_example.txt** file

3.14. SUMMARIZING PACKETS SIZE AND COUNT ON A NETWORK INTERFACE

The **netqtop** utility displays statistics about the attributes of received (RX) and transmitted (TX) packets on each network queue of a particular network interface. The statistics include:

- Bytes per second (BPS)
- Packets per second (PPS)
- The average packet size
- Total number of packets

To generate these statistics, **netqtop** traces the kernel functions that perform events of transmitted packets **net_dev_start_xmit** and received packets **netif_receive_skb**.

Procedure

1. Display the number of packets within the range of bytes size of the time interval of **2** seconds:

```
# /usr/share/bcc/tools/netqtop -n enp1s0 -i 2

Fri Jan 31 18:08:55 2023
TX
 QueueID avg_size   [0, 64) [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
 0       0       0       0       0       0       0
 Total   0       0       0       0       0       0

RX
 QueueID avg_size   [0, 64) [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
 0       38.0    1       0       0       0       0
 Total   38.0    1       0       0       0       0
--------------------------------------------------------------------------------
```

```
Fri Jan 31 18:08:57 2023
TX
 QueueID avg_size   [0, 64) [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
 0       0       0       0       0       0       0
 Total   0       0       0       0       0       0

RX
 QueueID avg_size   [0, 64) [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
 0       38.0    1       0       0       0       0
 Total   38.0    1       0       0       0       0
---------------------------------------------------------------------------
```

2. Press **Ctrl**+**C** to stop **netqtop**.

Additional resources

- **netqtop(8)** man page on your system
- **/usr/share/bcc/tools/doc/netqtop_example.txt**

CHAPTER 4. USING XDP-FILTER FOR HIGH-PERFORMANCE TRAFFIC FILTERING TO PREVENT DDOS ATTACKS

Compared to packet filters, such as **nftables**, Express Data Path (XDP) processes and drops network packets right at the network interface. Therefore, XDP determines the next step for the package before it reaches a firewall or other applications. As a result, XDP filters require less resources and can process network packets at a much higher rate than conventional packet filters to defend against distributed denial of service (DDoS) attacks. For example, during testing, Red Hat dropped 26 million network packets per second on a single core, which is significantly higher than the drop rate of **nftables** on the same hardware.

The **xdp-filter** utility allows or drops incoming network packets by using XDP. You can create rules to filter traffic to or from specific:

- IP addresses
- MAC addresses
- Ports

Note that, even if **xdp-filter** has a significantly higher packet-processing rate, it does not have the same capabilities as, for example, **nftables**. Consider **xdp-filter** a conceptual utility to demonstrate packet filtering by using XDP. Additionally, you can use the code of the utility for a better understanding of how to write your own XDP applications.

IMPORTANT

On other architectures than AMD and Intel 64-bit, the **xdp-filter** utility is provided as a Technology Preview only. Technology Preview features are not supported with Red Hat production Service Level Agreements (SLAs), might not be functionally complete, and Red Hat does not recommend using them for production. These previews provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

4.1. DROPPING NETWORK PACKETS THAT MATCH AN XDP-FILTER RULE

You can use **xdp-filter** to drop network packets:

- To a specific destination port
- From a specific IP address
- From a specific MAC address

The **allow** policy of **xdp-filter** defines that all traffic is allowed and the filter drops only network packets that match a particular rule. For example, use this method if you know the source IP addresses of packets you want to drop.

Prerequisites

- The **xdp-tools** package is installed.
- A network driver that supports XDP programs.

Procedure

1. Load **xdp-filter** to process incoming packets on a certain interface, such as **enp1s0**:

```
# xdp-filter load enp1s0
```

   By default, **xdp-filter** uses the **allow** policy, and the utility drops only traffic that matches any rule.

   Optionally, use the **-f <feature>** option to enable only particular features, such as **tcp**, **ipv4**, or **ethernet**. Loading only the required features instead of all of them increases the speed of packet processing. To enable multiple features, separate them with a comma.

   If the command fails with an error, the network driver does not support XDP programs.

2. Add rules to drop packets that match them. For example:
   - To drop incoming packets to port **22**, enter:

```
# xdp-filter port 22
```

     This command adds a rule that matches TCP and UDP traffic. To match only a particular protocol, use the **-p *protocol*** option.

   - To drop incoming packets from **192.0.2.1**, enter:

```
# xdp-filter ip 192.0.2.1 -m src
```

     Note that **xdp-filter** does not support IP ranges.

   - To drop incoming packets from MAC address **00:53:00:AA:07:BE**, enter:

```
# xdp-filter ether 00:53:00:AA:07:BE -m src
```

Verification

- Use the following command to display statistics about dropped and allowed packets:

```
# xdp-filter status
```

Additional resources

- **xdp-filter(8)** man page on your system
- If you are a developer and interested in the code of **xdp-filter**, download and install the corresponding source RPM (SRPM) from the Red Hat Customer Portal.

4.2. DROPPING ALL NETWORK PACKETS EXCEPT THE ONES THAT MATCH AN XDP-FILTER RULE

You can use **xdp-filter** to allow only network packets:

- From and to a specific destination port
- From and to a specific IP address
- From and to specific MAC address

To do so, use the **deny** policy of **xdp-filter** which defines that the filter drops all network packets except the ones that match a particular rule. For example, use this method if you do not know the source IP addresses of packets you want to drop.

WARNING

If you set the default policy to **deny** when you load **xdp-filter** on an interface, the kernel immediately drops all packets from this interface until you create rules that allow certain traffic. To avoid being locked out from the system, enter the commands locally or connect through a different network interface to the host.

Prerequisites

- The **xdp-tools** package is installed.
- You are logged in to the host either locally or by using a network interface for which you do not plan to filter the traffic.
- A network driver that supports XDP programs.

Procedure

1. Load **xdp-filter** to process packets on a certain interface, such as **enp1s0**:

   ```
   # xdp-filter load enp1s0 -p deny
   ```

   Optionally, use the **-f <feature>** option to enable only particular features, such as **tcp**, **ipv4**, or **ethernet**. Loading only the required features instead of all of them increases the speed of packet processing. To enable multiple features, separate them with a comma.

   If the command fails with an error, the network driver does not support XDP programs.

2. Add rules to allow packets that match them. For example:

   - To allow packets to port **22**, enter:

     ```
     # xdp-filter port 22
     ```

     This command adds a rule that matches TCP and UDP traffic. To match only a particular protocol, pass the **-p *protocol*** option to the command.

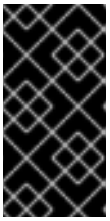
   - To allow packets to **192.0.2.1**, enter:

```
# xdp-filter ip 192.0.2.1
```

Note that **xdp-filter** does not support IP ranges.

- To allow packets to MAC address **00:53:00:AA:07:BE**, enter:

```
# xdp-filter ether 00:53:00:AA:07:BE
```



IMPORTANT

The **xdp-filter** utility does not support stateful packet inspection. This requires that you either do not set a mode using the **-m *mode*** option or you add explicit rules to allow incoming traffic that the machine receives in reply to outgoing traffic.

Verification

- Use the following command to display statistics about dropped and allowed packets:

```
# xdp-filter status
```

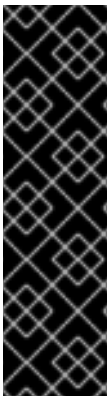
Additional resources

- **xdp-filter(8)** man page on your system
- If you are a developer and you are interested in the code of **xdp-filter**, download and install the corresponding source RPM (SRPM) from the Red Hat Customer Portal.

CHAPTER 5. USING XDPDUMP TO CAPTURE NETWORK PACKETS INCLUDING PACKETS DROPPED BY XDP PROGRAMS

The **xdpdump** utility captures network packets. Unlike the **tcpdump** utility, **xdpdump** uses an extended Berkeley Packet Filter(eBPF) program for this task. This enables **xdpdump** to also capture packets dropped by Express Data Path (XDP) programs. User-space utilities, such as **tcpdump**, are not able to capture these dropped packages, as well as original packets modified by an XDP program.

You can use **xdpdump** to debug XDP programs that are already attached to an interface. Therefore, the utility can capture packets before an XDP program is started and after it has finished. In the latter case, **xdpdump** also captures the XDP action. By default, **xdpdump** captures incoming packets at the entry of the XDP program.



IMPORTANT

On other architectures than AMD and Intel 64-bit, the **xdpdump** utility is provided as a Technology Preview only. Technology Preview features are not supported with Red Hat production Service Level Agreements (SLAs), might not be functionally complete, and Red Hat does not recommend using them for production. These previews provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

Note that **xdpdump** has no packet filter or decode capabilities. However, you can use it in combination with **tcpdump** for packet decoding.

Prerequisites

- A network driver that supports XDP programs.
- An XDP program is loaded to the **enp1s0** interface. If no program is loaded, **xdpdump** captures packets in a similar way **tcpdump** does, for backward compatibility.

Procedure

1. To capture packets on the **enp1s0** interface and write them to the **/root/capture.pcap** file, enter:

```
# xdpdump -i enp1s0 -w /root/capture.pcap
```

2. To stop capturing packets, press **Ctrl**+**C**.

Additional resources

- If you are a developer and you are interested in the source code of **xdpdump**, download and install the corresponding source RPM (SRPM) from the Red Hat Customer Portal.