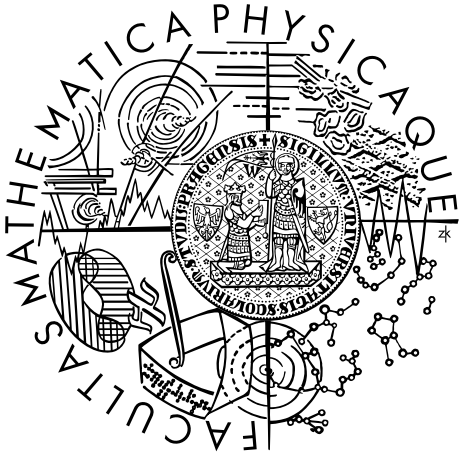
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Jan Buchar

HelenOS packet filter

Department of distributed and dependable systems

Supervisor of the bachelor thesis: Mgr. Martin Děcký

Study programme: Computer science

Study branch: General computer science

Prague 2015

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: HelenOS packet filter

Author: Jan Buchar

Department: Department of distributed and dependable systems

Supervisor: Mgr. Martin Děcký, Department of distributed and dependable systems

Abstract: Packet filtering is an essential feature of any operating system that aims to function as a network router or gateway. This thesis aims to extend the HelenOS operating system to support this functionality.

We analyze packet filters present in modern operating systems and implement a HelenOS service that provides a configurable and extendable packet filter. We also modify the HelenOS networking stack so that it allows an arbitrary service to function as a packet filter. We demonstrate the extendability of our implementation on a basic variant of the NAT mechanism.

Keywords: HelenOS packet filter networking

I'd like to express my sincere gratitude to my supervisor for his advice and guidance, and to my friends and family for their support and motivation.

Contents

Introduction

Network traffic filtering based on a configured set of rules is a core feature of software-based firewalls – machines that serve as a barrier between a secure internal network and another network that cannot be trusted, for example the Internet.

The HelenOS project aims to become a complete, multi-purpose operating system based on a microkernel architecture [1]. The goal of this thesis is to extend HelenOS so that it can function as a basic firewall that filters traffic flowing to and from an internal network. To fulfill this goal, we must:

- Modify the HelenOS networking stack to forward packets from an internal network to external hosts and vice versa
- Design an interface that allows a packet filter to hook into the network stack and provides a way for the network stack to query the packet filter
- Implement the packet filter itself. The packet filter is a standalone service that supports filtering based both on properties of the packets themselves, and on the state of connections (stateful packet inspection). It should support both IPv4 and IPv6. Apart from telling the network stack which packets can be accepted, the service can execute arbitrary actions, such as forwarding traffic from a specific port or NAT masquerading.

The thesis also contains a toolkit for automated testing of the packet filter in a virtualized network environment.

Thesis overview

In chapter 1, we analyze existing packet filters in modern operating systems and the way they are configured. Chapter 2 analyses the requirements of the thesis. Chapter 3 introduces the HelenOS packet filter from a user's point of view. Next, chapter 4 describes its internal mechanisms and the details of their implementation. Finally, chapter 5 describes the virtualized testing environment. It also presents some of the scenarios that are a part of the testing process and which showcase the capabilities of the packet filter.

1. Related work

This chapter contains a comparison of existing firewall software and an overview of their configuration.

1.1 Netfilter (GNU/Linux)

Netfilter is a set of hooks in the Linux kernel that makes it possible for kernel modules to register callback functions in the network stack [2]. It features both stateless and stateful packet inspection (with respect to connection state), NAT and other kinds of address translation.

The basic configuration unit of netfilter is a rule, which specifies a target and criteria. If a packet satisfies the criteria of a rule, netfilter takes action corresponding to the target. Rules are organized in chains and chains are divided into tables. The most notable tables are `filter` and `nat`. The former is consulted for every processed packet, the latter when a packet that creates a new connection is encountered.

The `filter` table contains three built-in chains, `INPUT` (used for packets for local addresses), `FORWARD` (for packets being routed) and `OUTPUT` (for packets sent from the machine running netfilter). Chains are traversed from the first rule to the last and when the processed packet matches, the traversal stops and the next rule is determined by the target of the matched rule. The target can be either jumping to another (possibly user defined) chain, or a definitive action such as dropping the packet.

Rules and chains can be manipulated using dedicated userspace utilities such as IPtables.

By default, netfilter doesn't use connection state when evaluating rules. This has to be enabled explicitly, for example with the `-m conntrack` option of the `iptables` command.

It is intended that netfilter will be replaced with nftables [3], a simpler packet filtering system based on a virtual machine that executes bytecode to inspect network packets.

1.1.1 IPtables

IPtables is a command line tool used to manage netfilter rulesets. It directly manipulates netfilter tables and chains. The following example creates three rules for inbound traffic. The first rule lets ICMP echo requests pass, the second lets TCP connections on port 80 pass and the third discards anything else.

```
# iptables -A INPUT -p icmp --icmp-type 8 -j ACCEPT
# iptables -A INPUT -p tcp --dport 80 -j ACCEPT
# iptables -A INPUT -j DROP
```

IPtables rules are not persistent – any configuration is lost after shutting down the machine. Persistent configuration can be achieved by writing the current configuration to a file with `iptables-save`. The configuration is then loaded at boot.

1.1.2 ferm

Ferm is a command line utility that loads rules from a structured file and translates them to iptables commands [4]. The configuration file basically describes the resulting netfilter chains. It also allows the use of variables, functions, arrays and blocks.

```
chain (INPUT OUTPUT) {
    proto (udp tcp) ACCEPT;
}
```

This example configuration translates to the following iptables commands:

```
# iptables -A INPUT -p tcp -j ACCEPT
# iptables -A OUTPUT -p tcp -j ACCEPT
# iptables -A INPUT -p udp -j ACCEPT
# iptables -A OUTPUT -p udp -j ACCEPT
```

1.1.3 UFW - uncomplicated firewall

UFW is an alternative tool for managing netfilter rules which aims for ease of use. Its configuration language is similar to that of OpenBSD PF.

```
# ufw default deny
# ufw allow proto icmp
# ufw allow to any port 80
```

The rules set up with ufw are stored in a file. This means no further actions are needed to make them persistent.

1.2 PF (OpenBSD)

PF[1] is a stateful packet filter that was originally written for OpenBSD [5]. It was intended to replace IPfilter, which was considered non-free due to its license [6]. Over time, it was ported to Mac OS X, FreeBSD, NetBSD and DragonFlyBSD. The ports have minor differences in configuration and implementation, which we won't cover. Instead, we will focus on the OpenBSD implementation.

The central point of PF's configuration is the configuration file `/etc/pf.conf`, which is loaded at boot and can be reloaded with the `pfctl` utility. The configuration is a list of rules that are evaluated from top to bottom. The last matching rule determines the taken action (the `quick` keyword can be used to end evaluation immediately).

Apart from stateless and stateful packet inspection, NAT and logging, PF has numerous notable features, such as

- address tables – a construct that allows efficient matching of packets against long lists of IP addresses

[1] The packet filter in OpenBSD shares its name with the one implemented in this thesis. Outside of this section, "pf" means the HelenOS packet filter. When referring to the OpenBSD packet filter, "OpenBSD PF" is used.

- packet scrubbing – de-fragmentation, dropping packets with invalid TCP flag combinations and other normalization procedures
- pfsync – synchronization of the state table over multiple machines that allows redundant firewalls

The rules are written in a language similar to English. It is worth noting that the word order is fixed. Variables (called macros) and lists are supported.

Following example blocks all traffic through the machine, except for HTTP requests from local networks. The HTTP responses will be passed too, because `pass` rules keep state by default. The `localnets` variable is a list, which means that a rule containing this list will be expanded into two separate rules, one for each item of the list.

```
localnets = "{192.168.1.0/24 192.168.10.0/24}"
block in  all
block out all
pass out log from $localnets to any port 80
```

This example ruleset also implicitly allows responses to the requests on port 80. This is because PF keeps tracks of connections by default and lets packets that belong to an existing connection pass automatically.

1.3 IPFW (FreeBSD)

IPFW is a stateful packet filter that is a part of the FreeBSD project [7]. It contains multiple components, such as a rule processor that is a part of the kernel, a NAT facility, a traffic logger and a traffic shaper called dummynet.

The packet filter is configured using a command line utility called `ipfw`. By default, the configuration is stored in `/etc/ipfw.rules`. There are also preset firewall types such as `workstation`, which protects the machine running `ipfw` using stateful rules (incoming traffic is only accepted if it belongs to an open connection).

The rules themselves are composed of keywords in a fixed order. The keywords make the syntax resemble English sentences. For example, the following command allows TCP connections to the machine on port 80.

```
# ipfw -q add 00400 allow tcp from any to me 80 in via dc0
```

The number after the `add` subcommand is used for ordering the rules. This enables inserting rules into arbitrary positions on a running system.

The packet filter enables filtering based on all common network and transport layer properties, such as addresses and ports. It also features more advanced filtering methods, for example limiting the number of connections allowed by a rule or matching based on the network interface that receives or sends the packet.

Although IPFW features stateful packet inspection, it doesn't track connections automatically. This has to be enabled for any rule that should create a new connection for packets that match it.

1.4 Microsoft Windows

There are two packet filtering solutions on the Windows platform – Windows Firewall and Static packet filtering.

Windows Firewall is primarily meant for desktop systems. It offers pre-set profiles based on the users network. For example the "Public" profile is the most restrictive one and is meant for networks anyone has access to.

Static packet filtering, available on Windows Server, provides a GUI to set up static per-interface whitelists or blacklists [8].

2. Analysis

In this chapter, we analyze the requirements of adding a packet filter to HelenOS and of its configuration.

2.1 Adding a packet filter to HelenOS

Because HelenOS is a multiserver operating system, it follows that a packet filter should be implemented as a standalone service that communicates with the network stack via IPC. We now outline the considerations related to integrating such service with the rest of the HelenOS network stack.

2.1.1 Packet interception

All of the packet filters covered in section 1 work with rules that are based on network and transport layer properties. Thus, it seems obvious that we should intercept network layer packets. However, this approach makes it difficult to obtain link layer properties, such as MAC addresses of the endpoints.

An alternative approach would be intercepting traffic on a lower level, such as the network interface layer. An advantage of this would be direct access to the raw packet data. However, we would have to reassemble fragmented packets and thus duplicate functionality which is already implemented in the `inetsrv` service.

We chose to implement packet filtering in the `inetsrv` service. This way, we can work with complete (unfragmented) datagrams, which are represented by the `inet_dgram_t` structure type. This isn't the only simplification of the overall design – `inet_dgram_t` also abstracts us from IP protocol versions, making it easy to support both IPv4 and IPv6. On the other hand, this approach makes it difficult to access the raw data of the packets, which might be useful for the filtering rules. Also, ICMP error messages are required to contain the first 64 bytes of the packet that caused the error [9]. This requirement isn't satisfied by this thesis.

2.1.2 IP datagram reassembly

Reassembling fragmented packets is indeed necessary for a packet filter that uses transport layer properties or performs modification such as NAT [10]. Without reassembly, we wouldn't be able to extract information such as port numbers from fragments of an IP datagram.

A negative implication of this is that the firewall needs to be the only point of entry into the network it protects. Otherwise, some fragments might be routed through a different machine, which would result in a loss of the whole datagram or bypassing the firewall. This might be resolved by introducing a mechanism similar to `pfsync` from OpenBSD.

2.1.3 Transport layer criteria

There are two basic ways of interoperating with the transport layer. We can either register our packet filter with the `tcp` and `udp` services and respond to their validation requests, or decode the transport layer headers of the IP datagrams we receive from `inetsrv`.

An advantage of the former approach is that it separates filtering on the network and transport layer. However, it introduces additional complexity into communication between the network stack and the packet filter and some packets would be passed between the services multiple times. Also, all the parts have to share the configuration (maintaining rules separately for every layer would be difficult and confusing). Moreover, mechanisms such as NAT also work the same for multiple transport layer protocols and implementing them for every protocol would mean duplicating the code or making the packet filter parts depend on shared code.

Considering this, we chose to implement all packet filtering functionality in a single service. This makes it straightforward to share common mechanisms and also doesn't obstruct extending the packet filter to support e.g. new transport layer protocols. A drawback of this is that we duplicate the logic of decoding transport layer PDUs and make the packet filter service operate on multiple network layers at once.

2.2 Configuration

It is necessary for a general-purpose packet filter to support configuration, even though some personal packet filters such as Windows Firewall only support a limited number of preset configurations (IPFW also offers preset profiles). This section analyses the way HelenOS pf will be configured.

2.2.1 Persistent configuration

It is important that packet filtering can be configured so that it persists after rebooting the firewall. This can be achieved in two ways – with a shell script that sets up the packet filter and is run on startup, or with a dedicated configuration file that is loaded when the packet filter is started. The former is used by IPtables and IPFW, the latter by OpenBSD PF.

An advantage of the first approach is that it doesn't introduce more complexity into the filtering rules – features such as variables and list expansions known from OpenBSD PF can be replaced with variables and cycle commands provided by the shell.

The second approach offers a cleaner, well-defined rule format that is independent of the shell it is run in. It also doesn't require support from the operating system (other than reading files), as opposed to the first approach, which needs the operating system to run some files on startup.

For HelenOS pf, we chose the second approach, because scripting support in HelenOS is currently in an early stage of development and doesn't support features such as variables. Moreover, it is easier to debug the configuration parser when we have full control of the loading process.

2.2.2 Runtime configuration

Because we decided to use a dedicated configuration language, it might seem superfluous to have a tool for adding rules dynamically. However, this is useful when we need to change the configuration for a limited period of time or for testing purposes.

On the other hand, it is useful to know all the rules when the packet filter is starting, as it enables us to perform optimizations of the rules and detect possible logical errors in the ruleset. This can be of course done also when adding the rules on runtime, but it might impair the performance of the packet filter.

We decided not to implement runtime configuration, because the benefits it brings aren't very important in the current stage of development.

2.2.3 Configuration file format

The rule syntax of IPFW and IPtables is designed with respect to rules being added with shell commands. Iptables accomplishes this by using command line options such as `-A` or `-J`, while IPFW uses a language that is more reminiscent of a natural language. Iptables commands look more like the majority of shell commands. The order of arguments is not fixed, which makes it easier to write the commands.

However, the IPtables commands are hard to understand without documentation, which might be the reason tools such as ferm or UFW exist.

Although OpenBSD PF is primarily configured with the `pf.conf` file, the rules' syntax is similar to those of IPFW or UFW.

The configuration syntax of HelenOS pf should use English words to describe rules, so that they are readable even for less experienced users. The order in which criteria and other parts of a rule are written shouldn't matter to the parser. This will help avoid unnecessary errors.

3. Usage

In this chapter we describe HelenOS pf from a user's point of view. We introduce the basics of configuration and control of the packet filter. We also outline some of its more advanced features such as stateful packet inspection and NAT.

3.1 Configuration syntax overview

Packet filtering can be enabled by running the `pf` command. However, before this can be done, we have to write a configuration file and save it to `/data/pf.conf`.

The configuration file is a list of rules (a *ruleset*) that determine the packet filter's behavior during packet validation. The rules are evaluated from the first one to the last. When a packet is validated, the first rule that matches it determines the action that should be taken.

A rule is written in the following general form[1] (square brackets denote an optional value, * means any number of repetitions from zero to infinity):

```
<action> <direction>[ <flag>]* {
    [<criterion>[,
    <criterion>
    ]*]
}
```

Actions

The <action> placeholder can be replaced with the following:

`pass` – let the packet pass,
`block` – just drop the packet without any notice to the sender
`reject` – drop the packet and notify the sender via ICMP

Traffic direction

<direction> can be either `in` or `out`, where `in` is used for any packets whose destination is the machine that runs `pf` and `out` is used for any packets that are forwarded or originated from the machine.

Flags

`continue` Using this flag, it is possible to override the default matching policy. When a rule with this flag matches, the evaluation continues and the rule is used only if no other rule matches. This is useful for specifying the default filtering policy at the beginning of the file.

`log` Any packets matching the rule are logged by the packet filter. This applies even when the rule has a `continue` flag.

`nostate` Forbid the rule to open a connection (see section 3.3).

[1] This is a simplified example. For a detailed description of the configuration language, see appendix C.

Criteria

A <criterion> is an atomic filtering condition in the following form:

```
<type> [!]<match_type> <arg>[ <arg>]*
```

<type> A criterion type such as dest_addr or src_port.

<match_type> The way the criterion matches against a packet property, e.g. = or between. It is possible to negate the matching type by prepending a ! sign to it – !between means anything outside given range.

<arg> The value(s) the criterion compares with the corresponding properties of the packet. For example, a dest_addr criterion with matching type = requires one argument such as 192.168.1.254.

When a rule has multiple criteria, they are evaluated as a conjunction (logical "and"). It is possible to express a disjunction (logical "or") with multiple rules.

Some criteria only apply to certain protocols, for example src_port will only work for TCP or UDP. Rules containing such criteria do not match packets of other protocols (a rule that contains a src_port criterion won't match an ICMP message).

Basic criteria and matching types

src_addr, dest_addr The source or destination IP address. Possible matching types are = or equal (the address in the IP header is equal to the argument) and subnet (the address in the IP header belongs to a subnet specified by a CIDR block in the argument). Both IPv4 and IPv6 addresses are supported.

src_port, dest_port The source or destination port. These criteria can only match packets sent over protocols that use ports. Available matching types are = (or equal), > (or greater), < (or smaller) and between. The argument (or arguments, for the between matching type) is required to be a number between 1 and 65535.

proto Transport layer protocol. Only the equal (or =) matching type is allowed. The argument can be either the protocol number, or its name protocol (e.g. TCP or ICMP). The protocol names are case insensitive.

interface Substitute the network interface name in the argument (e.g. eth1) with its configured subnet address. The only allowed matching type is =. The criterion is interpreted with respect to the direction of the rule. For in, the criterion translates to src_addr subnet <address>, for out to dest_addr subnet <address>.

Example configuration

The following is a complete example of a configuration file. It blocks any traffic by default, with the exception of HTTP communication with 10.0.10.20.

```
block in continue {}
block out continue {}

pass out log {
```

```
    dest_addr = 10.0.10.20,
    dest_port = 80
}

pass out {
    src_addr = 10.0.10.20,
    src_port = 80
}
```

3.2 Ruleset evaluation

When `pf` validates a packet, it traverses the ruleset and matches the packet against the criteria of each rule. The validation uses a first-match policy: the first rule that matches the packet determines the action to be taken. This means that the sooner a rule appears in the ruleset, the higher priority it has. This behavior can be fine-tuned using the `continue` flag.

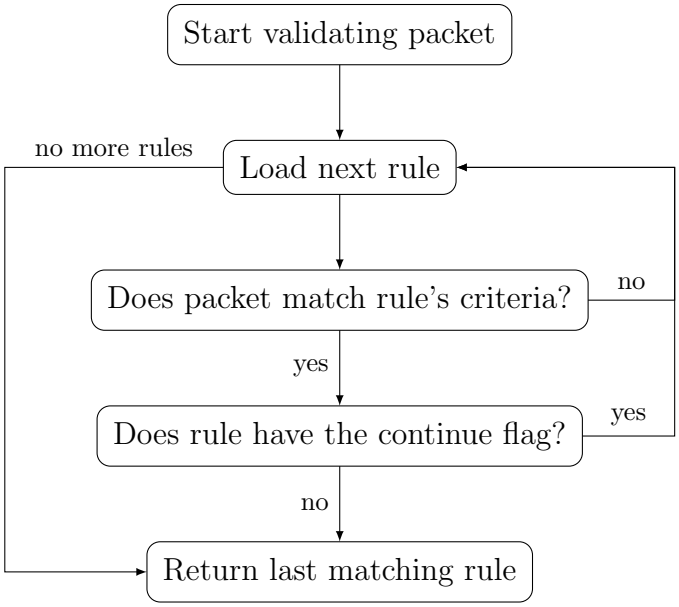


Figure 3.1: A diagram of the ruleset evaluation mechanism

3.3 Stateful packet inspection

`pf` is capable of keeping track of the network connections whose packets it filters. That means that whenever a packet capable of opening a connection matches a rule that lets it pass, an entry is created in `pf`'s state table. When a packet that belongs to a connection already present in the state table is encountered, the rule that opened the state is applied. This mechanism is a variant of stateful packet inspection – packets are filtered with respect to connection state.

An obvious advantage of this behavior is performance – the packet filter doesn't have to evaluate the ruleset again for every packet in a connection.

Also, the connections are bidirectional. When a reply to a packet arrives, it is passed automatically, which adds security and helps writing shorter rulesets. We illustrate this on the following ruleset:

```
block in continue {}
block out continue {}
pass out {
    dest_port = 80,
    proto = tcp
}
```

This ruleset blocks everything except for TCP connections to port 80. The ruleset itself, without state tracking, blocks responses to our TCP requests. We could address this by adding another rule that allows traffic from TCP port 80, but that could also allow outside hosts to initiate TCP connections to our machine, which might not be the intended behavior.

Keeping track of TCP connections also enables `pf` to check that the packets in the connections don't violate the protocol. For example, connections that haven't been initialized with a three-way handshake are blocked.

In some cases, this behavior is undesirable – for example when we only want to enable communication in one direction. In such situations, a rule can be forbidden to create a new connection using the `nostate` flag.

3.3.1 State keeping for stateless protocols

From the previous section it might seem that `pf` only performs SPI for TCP. It is true that protocols such as UDP or ICMP have no concept of a connection, but it is still possible to make records in the state table for them. These records have to expire after certain time, because there is no other way to know that the communication has ended.

3.4 Advanced usage

This section deals with packet filtering actions that alter the processed packets, namely NAT and port forwarding.

3.4.1 Network address translation

Network address translation (NAT) is a process of remapping a network address space into another by modifying the network layer headers of forwarded packets. This can be used to conserve public IP addresses and to secure internal networks by hiding the computers in them behind a machine that performs NAT.

With these goals in mind, `pf` provides the `masquerade` action, that transforms packets originating from an internal network so that they appear to come from the firewall itself. Any replies are then translated back and passed thanks to stateful packet inspection. The following is an example of a basic NAT configuration:

```
masquerade(10.0.20.15) {
    src_addr subnet 10.0.10.0/24
}

block {
    dest_addr subnet 10.0.10.0/24
}
```

Any traffic originating from the internal network (subnet `10.0.10.0/24`) will be masqueraded using NAT. The translation is transparent to the machines in the internal network – from their point of view, they communicate directly with the remote host. However, to the remote host it will appear that the traffic is coming from the firewall (address `10.0.20.15`). Additionally, any traffic destined directly to the internal network is blocked to ensure its isolation.

In case we need to filter packets going out from the masqueraded network, it suffices to add a new rule before the `masquerade` rule. This ensures that packets blocked by the new rule will never match the `masquerade` rule.

Also note that `pf` ensures that the response to a request from a masqueraded network originates from the correct address and port (which is the destination of the request). This prevents computers outside of the internal network from initiating a connection with computers inside.

3.4.2 Port forwarding

Port forwarding is a mechanism that allows forwarding traffic from a dedicated port of the machine running `pf` to another machine. This is useful for example when we need to expose a service running on a machine in a NAT-protected network to the outside world.

This functionality is provided by the `forward` action. A rule with this action must containtwo mandatory criteria, `dest_addr` and `dest_port`, and optionally a `proto` criterion, such as in this example:

```
forward(10.0.10.17, 8080) {
    dest_addr = 10.0.20.15,
    dest_port = 80
}
```

This example rule forwards all traffic from port 80 of the firewall to port 8080 of the machine with address `10.0.10.17`, and makes sure that replies from that machine appear to originate from the firewall.

In case it's necessary to filter the forwarded packets, we can use the special `through` direction flag as follows:

```
block through {
    src_addr subnet 98.154.87.0/24,
    dest_port = 8080
}
```

This rule prevents traffic originating from a specified subnet from reaching port 8080 of a machine protected by a firewall via port 80 of the firewall (supposing we also use the `forward` rule from the previous example). Note that the `through` rules are evaluated after the forwarded packets are transformed, i.e. the actual address and port are filled in.

3.5 Disabling packet filtering

In some situations, it might be necessary to disable the packet filter. This can be done using the `pfctl` userspace utility. To deactivate `pf`, simply run the following command:

```
# pfctl disable
```

It is also safe to simply stop `pf` using the `kill` command. However, it might cause some error messages to appear in the log files. This is because the network stack services have no way to tell if `pf` was stopped or if it terminated due to an error.

3.6 Traffic logging

When a packet matches a rule with the `log` flag, this fact is recorded in the `/log/pf` log file. Note that if a rule has both the `log` and `continue` flags, any packet matching it is logged, even though the packet is matched later by another rule without the `log` flag. This way, it is possible to log all packets using only one rule.

The messages about matched packets are logged with level 3 ("note"). This means that they are logged by default and can be disabled by using a less verbose log level for `pf`.

A report of a matching packet contains its source and destination address, protocol number, source and destination ports for protocols that use ports and the name of the action that was used to dispatch the packet (not necessarily the one with the `log` flag).

4. Implementation

The following chapter deals with modifying the HelenOS network stack to support packet filtering by passing IP packets to an external service and with the design of such service (`pf`). The former is covered in section 4.2, the latter in section 4.3 and following sections.

Both `inetpf` and `pf` are written in the C programming language, which is well suited for relatively low-level tasks such as packet filtering. It is also the main programming language used in HelenOS.

4.1 Modifications of the network stack

Although an interface with the packet filtering service (referred to as "packet filter") is provided by `inetpf`, we had to modify the network stack to use this interface. Because we decided that `inetpf` and the packet filter will use the `inet_dgram_t` data structure, we will refer to the units of filtered traffic as datagrams.

We now describe the modifications we made to the `inetsrv` service (referred to as "original `inetsrv`").

4.1.1 Forwarding datagrams

A machine that is intended to work as a firewall must be able to forward packets sent from one network to other network. Because the original `inetsrv` doesn't support forwarding at all, we implemented a basic variant of this mechanism: whenever a datagram is received that doesn't match any of the firewall's addresses, it is re-sent with the `inet_route_packet` function. Before that, its TTL value is decremented.

4.1.2 Filtering outgoing datagrams

All outgoing datagrams, regardless of their origin, are processed by the `inet_route_packet` function, where they should be validated. However, if the datagram was sent as a result of rewriting an incoming datagram, it is not necessary to validate it again – the packet filter already processed it.

We chose to move the code that actually sends the datagram to a new function called `inet_do_route_packet`. The `inet_route_packet` function uses the `inetpf` interface to validate the datagram. If it passes, it is sent using our new function. It is also safe to use this function to send datagrams without validation.

4.1.3 Filtering incoming datagrams

All received IP packets have to go through the `inet_recv_packet` function, which makes it a good place for filtering incoming traffic. However, this function works with packets which might have to be reassembled. In the original `inetsrv`, reassembled datagrams are immediately delivered locally. This behavior is not acceptable, as it bypasses validation.

To address these issues, we replaced the `inet_recv_dgram_local` function called by the reassembly module with `inet_recv_dgram`. This new function decides if the datagram is destined to the firewall or if it should be forwarded. In the first case, it is validated as an incoming datagram. In the second case, it is forwarded using `inet_route_packet`, and therefore validated as an outgoing datagram.

There is an additional issue in the first case: the packet filter might rewrite the datagram so that it becomes an outgoing datagram. This happens for example after port forwarding. If such transformation is made, `inetpf` returns a special value. Thus, we can check if the datagram is still destined to the firewall (in this case we deliver it locally), or if it should be forwarded (then we send it using `inet_do_route_packet` as it has already been validated).

4.2 inetpf interface

The `inetpf` interface is a layer between the IP networking server (`inetsrv`) and a packet filter. Principially, any application can serve as a packet filter if it implements the `inetpf` protocol.

4.2.1 Packet filter registration

A packet filter has to register itself with the `inetpf` service on startup before it can start filtering traffic (see figure 4.1). Only one service can act as a packet filter – it would be hard to deal with different responses and arbitrary modifications by the services. If there is no registered service, `inetpf` lets everything pass.

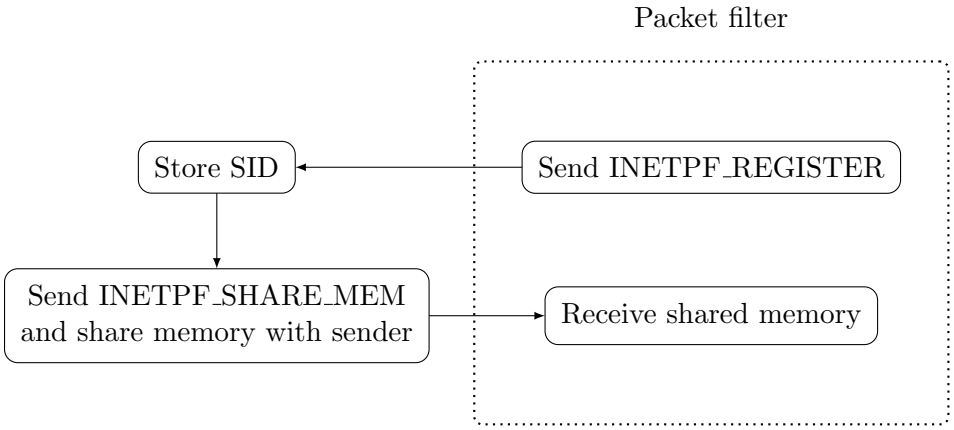


Figure 4.1: Registering a packet filter with inetpf

4.2.2 Datagram validation

When `inetsrv` processes an incoming or outgoing datagram, it uses the `inetpf_validate` function to decide whether the datagram should be blocked. This function transfers the packet to the packet filter service and returns its response.

See figure 4.2 for a detailed description of this process.

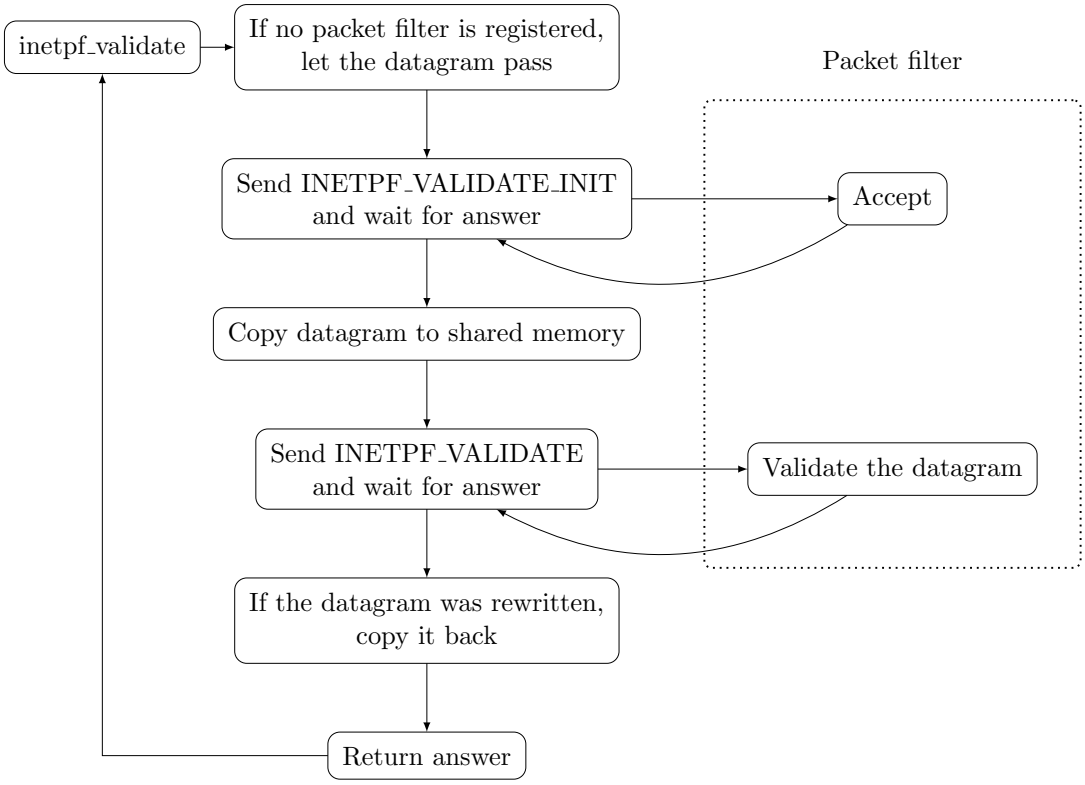


Figure 4.2: The inetpf validation process

`inetpf` passes the datagrams that are to be validated via shared memory. This implementation is faster than copying them between the services and also makes the task of receiving modified datagrams trivial. On the other hand, the protocol has to prevent data races such as `inetpf` rewriting the datagram in shared memory with a more recent one while the packet filter is still validating it.

4.2.3 Response types

After a datagram is validated, the packet filter service returns an answer that determines the action `inetsrv` should take:

`PASS` – the datagram has passed the check and can be delivered
`BLOCK` – drop the datagram – it didn't pass
`PASS_REWRITE` – deliver the datagram, but reload it from shared memory first (this response means that the packet filter made some modifications to the datagram)

Any actions apart from these, such as rejecting a datagram with an ICMP message, have to be performed by the packet filter itself. This decision was made to ensure that the `inetpf` interface stays simple and doesn't provide services that are out of the scope of `inetsrv`. A drawback of this is that every potential filtering service will have to implement the common functionality (such as aforementioned ICMP reject messages) again.

4.2.4 Packet filter termination

When the packet filter exits, for example because of a user's request, it should notify `inetsrv`. This is implemented by the `INETPF_UNREGISTER` method.

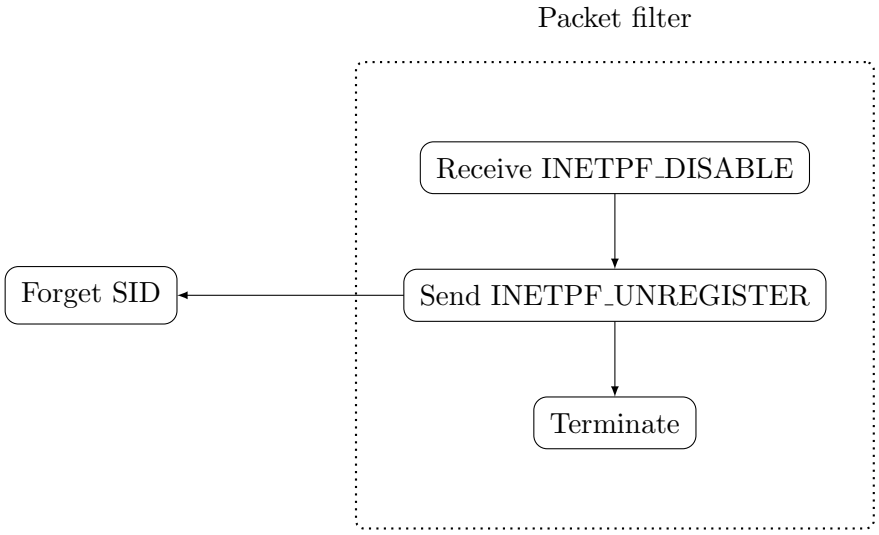


Figure 4.3: Notifying inetsrv of the packet filter's termination

Note that `inetpf` also unregisters the packet filter if it crashes or hangs up for some reason, but such event is logged as an error.

4.3 PF service

The `pf` service is the main subject of this thesis. It implements static and stateful packet filtering on the network layer, based on a list of configured rules. It also features an extendable system of datagram transformations such as NAT masquerading or port forwarding.

This section describes the core components of the service.

4.3.1 Filter

The `filter` module is the centerpoint of packet filtering functionality. It uses a list of configured rules to validate packets received from the `inetpf` interface.

The rules are represented by a data structure that contains an action callback and a set of criteria. An action callback is a function (with certain metadata) that returns an `inetpf` response. Apart from this, there aren't many constraints on the function – it can for example send ICMP messages or manipulate the datagrams' IP headers. The criteria are a conjunction of atomic conditions, such as "destination address equals 10.0.0.138". A rule as a whole specifies the action to be taken for packets matching its criteria.

When `pf` starts, it loads rules from a configuration file. The filter then divides them into multiple lists, called rulesets. Currently, there are separate rulesets for inbound traffic, outbound traffic and traffic on mapped ports. Because it is trivial to decide which ruleset should be evaluated during validation, this division gives us a performance advantage, as we don't have to evaluate irrelevant rules.

Rulesets are evaluated sequentially, using a first-match policy – the first rule in the ruleset whose criteria are satisfied by the datagram determine the action to be taken. This policy was chosen because it is (subjectively) easier to understand and optimize.

Extendable actions

While the set of possible criteria is rather restricted and it is not difficult to add a new one, the amount of possible actions is virtually unlimited. Moreover, actions are more complicated than simple boolean expressions – they can have potentially any effect. Therefore, we designed actions as modules that are as independent as possible. We demonstrate this by describing how to implement a new action.

The source file of the new action must contain a structure of type `action_t`. The structure has following members:

`name` – a string that is used to distinguish the action during configuration loading

`fnc` – the function that is called when a datagram matches a rule. It must return a valid `inetpf` response.

`init` – a function that initializes the action's global state, e.g. initiates a connection to a service. It is called during the filter's initialization.

`load_rule` – a function that parses the action's arguments and returns a pointer to the rule's configuration. This pointer is passed to `fnc` whenever it's called and enables it to maintain rule-specific context.

`destroy` – a function that is called when `pf` is going to exit and cleans up the action's global state

This structure has to be declared in the `actions.c` file, and included in the `filter_actions` array in the same file. To ensure that our new action is compiled and linked with the rest of the filter, we must include its source file in the `SOURCES` variable in `pf`'s Makefile.

This way of extending the filter encapsulates the individual actions well, without restricting their access to the rest of `pf` and external libraries (provided that they're included in the Makefile).

It would be possible to eliminate the need to list the actions in `actions.c` using a linker script that defines a new section in the resulting ELF binary and putting the `action_t` structures into this section. However, this would require a modification of HelenOS' own linker script, which could prove more difficult to maintain than a plain C file.

4.3.2 Port map

The `forward` and `masquerade` actions require `pf` to map a port on the firewall to a remote address and port. The mapping can optionally have a timer, whose expiration means that the entry is no longer valid. This functionality is provided by the `portmap` module.

The module's interface

For a given mapping (local port and protocol to remote address and port), `portmap` stores a data structure (`portmap_callback_t`) that contains a function

that should be invoked on datagrams received through the local port. Inserting port mapping is implemented in the `portmap_insert` function that also takes the current timestamp as an argument. The timestamp is used to check the mapping for expiration.

The port map itself doesn't validate any traffic. Instead, the `filter` module calls the `portmap_find` function when validating a datagram to see if it should be forwarded somewhere else. This function takes a protocol and port number used to search for the mapping, along with a timestamp used for expiration checks. If a mapping is found and hasn't expired, its timestamp is updated with the current one and its callback structure is returned.

A reason for these functions to work with timestamp arguments instead of getting the timestamp themselves is simpler testing – in unit tests, we can easily supply them any value and check if their behavior is correct.

Implementation details

Mappings are stored in a hash table that uses the protocol number and local port as a key. This abstract data structure is a part of the HelenOS project, which is one of the main reasons we decided to use it – it is unnecessary to implement our own data structure for such a simple task.

Another reason for using hash tables is that they only require us to supply a hash function for our key (a tuple of numbers). This is not as difficult as for example defining an ordering on the keys, as required by search trees.

`portmap` also uses a reverse table that maps remote addresses and ports to local ports. This is used in checking for duplicate mappings.

Reserving ports

Because `pf` operates predominantly on the network layer (it receives datagrams from `inetsrv`), it has no concept of sockets. This becomes a problem when we need to reserve a port that is used for a mapping. Binding a socket to such port from some other program than `pf` would result in erroneous behavior – anything that should be received by the socket would be instead forwarded somewhere else.

To address this issue, we open a new socket whenever a port mapping is registered and bind it to the mapped port. The socket is in fact never used, but it prevents other applications from using the port. This solution might have an impact on `pf`'s performance, but it is currently the only feasible way of reserving a port.

4.3.3 State table

Apart from static packet inspection (only based on the datagram's properties), `pf` also features stateful inspection. This means that it keeps track of connections between hosts – once a datagram matches a rule, it creates a connection. From this point, datagrams that belong to this connection are accepted without evaluating the rules.

An obvious advantage of stateful inspection is the performance gain of not having to evaluate the rules for every datagram. Another one is that the connections are bidirectional, which simplifies configuration – we don't have to explicitly allow

replies to allowed requests. As this might not always be the intended behavior, it is possible to set a "nostate" flag for a rule, which prevents it from opening connections.

A connection is identified by its protocol, source and destination address and source and destination port. The state table uses these values as keys to stored data about the connection – most importantly its state, a timestamp used to check it for expiration and the rule responsible for its creation. The storage logic is implemented in the `statetable` module.

The module's interface

The module offers the following functions:

`statetable_insert` Takes a rule, protocol number and a datagram. The addresses and ports are extracted from the datagram and used to record a new connection.

`statetable_check` Takes a protocol number and a datagram. These are used to look up a connection in the state table. If a connection is found, its state and the rule responsible for its opening are returned by output parameters. If the datagram is invalid in context of this connection (for example it tries to open a new connection even though one already exists), the function itself returns false. If it's valid, the function returns true. If there is no connection present in the state table, the function returns true if the datagram can be used to open a new connection.

Both of these function also require the current timestamp as a parameter – it is used to check whether the connection expired.

Implementation details

Data about connections is stored in structures of type `connection_t`, which are in turn stored in a hash table.

We chose to use a hash table because it is rather straightforward to compute a hash out of the values that identify a connection (protocol, source and destination address and port), in contrast to for example implementing a comparison function on tuples of these values.

Connection state tracking

At this state of development, `pf` only validates the initial phase of a TCP connection. When a new connection is being opened, we ensure that it is initiated with a three-way handshake by only accepting datagrams with the correct flags. After the connection is initiated, only packets without the `SYN` flag are accepted. The connection remains open until it expires, because it is very difficult (if not impossible) for the firewall to determine if the connection was closed [11].

Although stateless protocols such as UDP and ICMP have no explicit start and end of a connection, it is still possible to track their state in some measure – any accepted datagram can open a new connection, which remains open until expiration. This gives us the advantage of both simpler configuration and better performance.

4.3.4 The validation process

In figure 4.4 we can see how the `filter` module proceeds when it validates a datagram. Datagrams destined to a mapped port must be transformed before looking up their state so that the state table knows the actual endpoints of a potential connection.

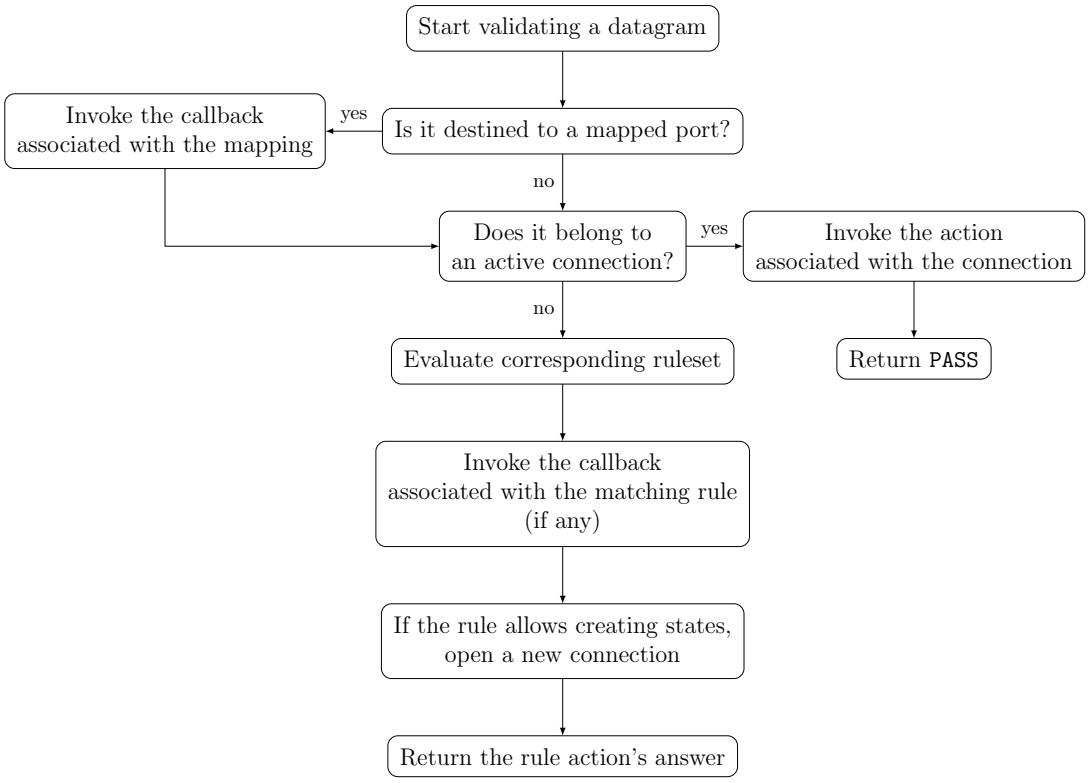


Figure 4.4: A diagram of datagram validation

4.3.5 Configuration parsing

Because the configuration language is relatively simple, we chose to implement a custom parser. It reads the configuration file character by character, treating continuous strings of non-whitespace characters as tokens – lexical units of the configuration file. The parser only keeps track of the current token.

The parsing process is initiated by calling the `config_load` function. This function takes a function as an argument that returns lines of the configuration file. It uses this function instead of reading the file directly to enable unit testing of the configuration parser.

During the parsing process, `config_load` calls the `read_rule` function repeatedly. The `read_rule` function advances the current token cursor when necessary. If the parser reaches the end of the input without errors, it returns a list of `rule_prototype_t` structures.

The prototype list is handled by the `filter` module, which preprocesses it and transforms it into the rulesets that are used for validating packets.

4.4 NAT masquerading

`pf` implements a basic variant of the NAT mechanism based on port forwarding (NAPT). It maps an unused port number of the firewall to an internal address and port. For outgoing traffic, packets are modified so that it looks like they originated from the mapped port of the firewall. The address and port of any packet destined to the mapped port are rewritten to the associated internal address and port. See figure 4.5 for an illustration of this mechanism.

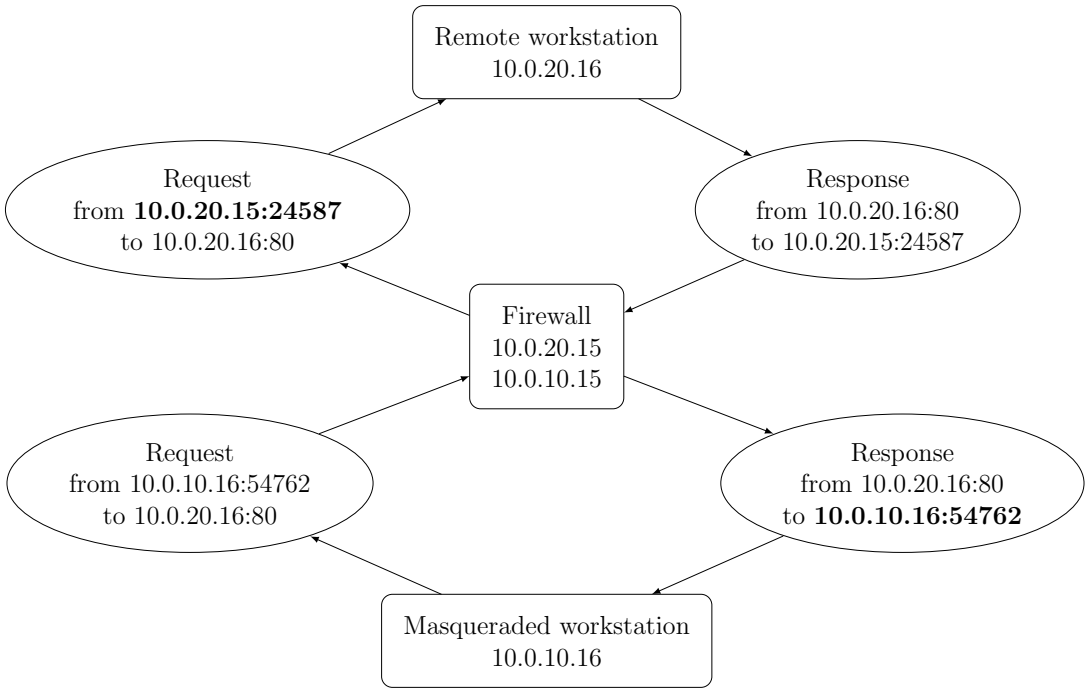


Figure 4.5: A demonstration of the NAPT mechanism

4.4.1 NAT for ICMP

Although ICMP messages don't use ports, it is still possible to perform NAT for some of them, most notably echo requests and replies [12]. Where ports would be used for TCP or UDP, we use the message identifier. The identifier is a 16-bit number, exactly like TCP and UDP port numbers. Thanks to this, we can easily use the `portmap` module.

A drawback of this solution is that we cannot reserve these "ports", which means that there is a chance that a collision occurs. Also, messages that do not contain an identifier, such as "Destination Unreachable" [9] have to be dropped.

5. Evaluation

In this chapter we describe the environment that was used to test the packet filter. We also present some of the test scenarios that show its important features.

5.1 Testing environment

At this stage of development, HelenOS is mainly tested using virtualization software (namely qemu [13]), which is also the case of HelenOS pf. This approach was chosen because it allows automated testing and makes the testing environment portable and easily deployable, as opposed to building a physical network.

For a guide on using the testing environment, see appendix B.

5.1.1 Virtual network implementation

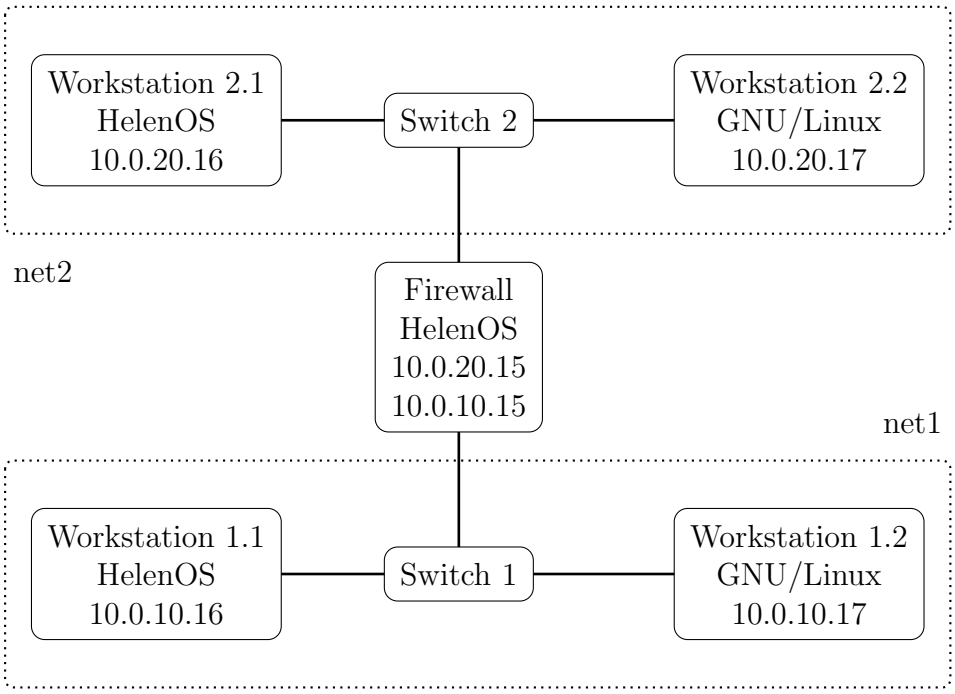


Figure 5.1: A diagram of the virtualized testing environment

To test the capabilities of HelenOS serving as a firewall, we set up two virtual networks, `net1` and `net2`, that are connected by a router running HelenOS pf. The guests in these networks are qemu virtual machines that boot from a live CD image. This helps to ensure that no configuration persists from previous tests. Also, during development, the images are updated frequently and it would be difficult to update a HelenOS installation on a virtual hard drive.

The networks contain both HelenOS and GNU/Linux guests. One reason for this is that GNU/Linux ships with networking utilities that are far more mature than their HelenOS counterparts. For example, `netcat` is a priceless tool for testing network communication.

We connect the guests using VDE[1], which is a robust virtual networking solution shipped with qemu. The VDE networks are entirely separated from the network of the host system, which prevents any broadcast messages originating from the host's network from entering the testing network.

It is possible to bridge the testing network with the host's network. However, this would require extensive, platform dependent configuration. For example, on GNU/Linux systems, setting up a network bridge requires root access, as we need to create a virtual network interface.

On the other hand, having no direct network connection to the testing environment means we have to find another way of controlling the virtual machines remotely.

5.1.2 Controlling the virtual machines

With network connection to the guests, we could use e.g. `telnet` to connect to HelenOS' `remcons` service and obtain a basic shell. A major drawback of this solution is that the virtual machine running `pf` would have to be aware that it's being tested to allow traffic belonging to `remcons` pass. To control the GNU/Linux guests, we could use `ssh`. That would require us to send commands to these guests using different utilities than with the HelenOS guests.

We chose an alternative aproach that eliminates the dependency on network configuration and the guest OS: we translate commands into series of key events that are sent to the virtual machines via qemu monitor, which allows us to connect any OS that can be controlled by a keyboard to the network. Of course, the commands themselves are still different for the operating systems, but we can enter them using a uniform interface.

5.1.3 Monitoring network communication

Because of our approch to controlling the guests, we have no information about the results of the commands we ran[2]. This would be an issue for example if we tested interactive commandline utilities. However, for testing network communication, it is more suitable to observe traffic on the guests' network interfaces and check if it is filtered correctly by the packet filter.

Qemu can be configured to capture traffic on any network interfaces in the virtual network and save it to pcap files. These files are then examined using the ngrep[3] utility, which filters the capture using the same rule syntax as libpcap (used in tcpdump). After running the tests, the captures can also be examined manually, using a program such as Wireshark.

[1] `http://vde.sourceforge.net/`

[2] We can of course check the results on the display of the virtual machine. However, it is very difficult to automate this.

[3] `http://ngrep.sourceforge.net`

5.2 Scenario: Restricting traffic to specified ports

The following is an example of a highly restrictive setup. Workstations in the internal network are only allowed to make HTTP requests and query one specific DNS server. The workstations could be used for example to provide people with web access.

To facilitate remote administration, SSH login and sending ICMP messages are enabled for a single machine (which might be for example the administrator's workstation).

```
# Block everything by default
block in continue {}
block out continue {}

# Only enable DNS queries to 10.0.20.17
pass out {
        interface = eth1,
        proto = udp,
        dest_port = 53,
        dest_addr = 10.0.20.17
}

# Enable HTTP(S) requests from the internal network
pass out {
        interface = eth1,
        proto = tcp,
        dest_port = 80
}

pass out {
        interface = eth1,
        proto = tcp,
        dest_port = 443
}

# Enable SSH connections to machines in the internal network
# from 10.0.20.16
pass in {
        interface = eth1,
        proto = tcp,
        dest_port = 22,
        src_addr = 10.0.20.16
}

# Enable sending ICMP messages from 10.0.20.16
pass in {
        interface = eth1,
```

```
        proto = icmp,
        src_addr = 10.0.20.16
}
```

5.3 Scenario: Workstations behind a NAT

In this setup, the workstations in `net1` are hidden using NAT, which means they can't be reached from the outside. Because there is a HTTP server running on Workstation 1.1 and we need to be able to access it from the outside, we forward port 80 on the firewall to port 8080 on address 10.0.10.16. We restrict access to this web server to a single machine. This can be achieved with following configuration:

```
# Block traffic destined to the firewall by default
block in continue {}

# Block non-HTTP traffic from internal network
block out {
        interface = eth1,
        dest_port != 80
}

# Block any traffic destined directly to the internal network
block in {
        interface = eth1
}

# NAT for the internal network
masquerade(10.0.20.15) {
        interface = eth1
}

# Only allow 10.0.20.17 to access our web server via port 80
block through continue {}

pass through {
        src_addr = 10.0.20.17,
        dest_port = 8080
}

# Forward everything on port 80 to an internal web server
forward(10.0.10.16, 8080) {
        dest_addr = 10.0.20.15,
        dest_port = 80
}
```

Conclusion

The main goal of this thesis was to develop a packet filter that is configurable and easily extendable. We fulfilled this goal by implementing `pf`, a full-featured filtering service. Its architecture allows adding actions that are executed on packets that match a rule. This is demonstrated by the `reject`, `forward` and `masquerade` actions.

Furthermore, we analyzed the filtering mechanisms present in the packet filters shipped with major operating systems and the methods of their configuration. We used the results of this analysis to design a configuration file format that is easy to write as well as to understand. The analysis also helped us to choose an established matching strategy (first match wins), which is easy to understand and allows optimizing the ruleset by placing the most frequently used rules near its start.

Our packet filter also features stateful packet inspection. It doesn't allow using the connection state as an explicit matching criterion, and performs SPI automatically instead with the option of disabling it.

Lastly, we modified the HelenOS networking stack so that it supports packet filtering for both IPv4 and IPv6 by querying a separate packet filter service. We also added support for packet forwarding to the Internet Protocol service.

In conclusion, these features allow HelenOS to be used as a firewall or NAT machine that protects either a single computer or a small local network.

Additionally, we created a simple virtualized testing environment that can be used to test packet filtering, as well as other networking-related features.

Future work

Although we fulfilled the thesis' goal, there are still many ways in which the packet filter could be improved:

- We could modify the way the networking stack and packet filter communicate, so that the filter also receives the raw packet data. This would allow several new features:
  - The filter could log packets that match a rule in the PCAP format, which is supported by network diagnostics tools (e.g. Wireshark).
  - The `reject` action could send error messages that contain the header of the packet that caused the error, as requested by RFC 792 [9].
  - It would be possible for the `interface` criterion to match actual MAC addresses of the interface that sends or receives the packet.
- We could proxy TCP handshakes in a manner similar to that of OpenBSD PF. This would improve the security of the firewall and reduce vulnerability of the machines in the internal network to certain kinds of attacks.
- Another useful feature of OpenBSD PF are address tables - groups of subnet addresses that are stored in data structures that allow efficient lookups. These are useful for matching packets against large numbers of IP addresses.

- Compiling the ruleset to a pseudo-virtual machine bytecode like in nftables would lead to a performance boost. We could even integrate the pseudo-virtual machine into the network stack itself. This could reduce the amount of necessary communication with the packet filtering service, as it could just copy the bytecode after loading the configuration file and then just receive notifications of events such as logged packets.

Bibliography

[1] *HelenOS website*. 2015. URL: http://www.helenos.org (visited on 05/15/2015).

[2] *Netfilter homepage*. 2015. URL: http://www.netfilter.org/index.html (visited on 05/15/2015).

[3] *Nftables project homepage*. 2015. URL: http://netfilter.org/projects/nftables/ (visited on 05/15/2015).

[4] *Ferm - firewall rules made easy*. 2015. URL: http://ferm.foo-projects.org/download/2.2/ferm.html (visited on 05/15/2015).

[5] *OpenBSD PF documentation*. 2015. URL: http://www.openbsd.org/faq/pf/ (visited on 05/15/2015).

[6] *OpenBSD commit: Remove IPF*. 2001. URL: http://marc.info/?l=openbsd-cvs&m=99118909527873 (visited on 06/28/2015).

[7] *IPFW documentation*. 2015. URL: https://www.freebsd.org/doc/handbook/firewalls-ipfw.html (visited on 05/28/2015).

[8] *Configuring Static Packet Filters*. 2009. URL: https://technet.microsoft.com/en-us/library/dd469754(v=ws.10).aspx (visited on 05/14/2015).

[9] *RFC 792 - Internet Control Message Protocol*. 1981. URL: https://tools.ietf.org/html/rfc792 (visited on 06/23/2015).

[10] *RFC 2993 - Architectural Implications of NAT*. 2000. URL: https://tools.ietf.org/html/rfc2993 (visited on 06/26/2015).

[11] *RFC 2663 - IP Network Address Translator (NAT) Terminology and Considerations*. 1999. URL: https://tools.ietf.org/html/rfc2993 (visited on 06/28/2015).

[12] *RFC 3022 - Traditional IP Network Address Translator*. 2001. URL: http://tools.ietf.org/html/rfc3022 (visited on 06/23/2015).

[13] *QEMU processor emulator*. 2015. URL: http://wiki.qemu.org/Main_Page (visited on 07/26/2015).

[14] *Compiling HelenOS from source*. 2015. URL: http://trac.helenos.org/wiki/UsersGuide/CompilingFromSource (visited on 07/26/2015).

List of Figures

List of Abbreviations

IPC	Interprocess communication. 8
NAPT	Network address and port translation. 25
NAT	Network address translation. 3–6, 8, 9, 11, 14, 15, 20, 25, 29, 30, 39
PDU	Protocol data unit. 9
SPI	Stateful packet inspection. 14, 30

Attachments

A. Contents of the attached CD

The attached CD contains following items:

source/

A folder with the complete Bazaar source code branch.

image.iso

An ia32 HelenOS live CD image.

changes.patch

A patch file containing all the changes made to HelenOS in this thesis.

virtlab/

The testing environment. The `tests` directory contains the two test scenarios presented in chapter 5, as well as other, less complicated tests.

thesis.pdf

This thesis.

B. Running the test environment

The following is a step-by-step guide on building the HelenOS CD image and running tests of packet filtering functionality.

B.1 Building HelenOS

This step is only needed to test architectures other than IA32 - there is already an IA32 image on the attached CD.

1. Copy the sources from the attached CD to your hard drive and open a terminal in the source folder

2. Choose a folder where the HelenOS toolchain should build a cross-compiler:

   ```
   $ export CROSS_PREFIX=/home/user/helenos-build
   ```

3. Build a cross-compiler (HelenOS can only be built with a supported version of GCC [14]):

   ```
   $ cd tools
   $ ./toolchain.sh ia32
   ```

   Depending on your OS, it might be necessary to install some dependencies.

4. Build HelenOS:

   ```
   $ cd ..
   $ make PROFILE=ia32
   ```

This will produce an `image.iso` file in the source directory. We can either boot this image directly or use the testing environment.

B.2 Test environment usage

The testing environment is a virtualized network of HelenOS and GNU/Linux machines. It can be run with

```
$ ./virtlab.sh [--headless] [--debug] /path/to/image.iso \
  [tests/test_dir]
```

This command starts the virtual machines and sets up networking. If a `test_dir` is supplied, it also runs the test in the directory. This means it creates a hard disk image with a packet filter configuration file and starts `pf` on the machine that serves as a firewall. Then it runs a series of commands to test if the packet filter behaves as expected.

If the `--headless` flag is used, QEMU's graphical output is disabled and the environment is shut down after the test finishes. The `--debug` flag sets `pf`'s logging level to "debug", which is useful for debugging tests.

The `virtlab.sh` script uses the Finnix live CD on its GNU/Linux machines. Its image is included in the attached CD. If necessary, it is possible to use a different image by setting the `LINUX` environment variable.

The HelenOS images are run on i386 systems by default. To test other architectures, we must use the `Q` environment variable to supply a command to run the corresponding emulator, for example:

```
$ Q=qemu-system-x86_64 ./virtlab.sh
```

There is also a script that runs `pf`'s unit tests and prints their results:

```
$ ./unit.sh /path/to/image.iso
```

It is also possible to run the unit tests and all the feature tests:

```
$ ./run_all.sh /path/to/image.iso
```

B.2.1 Dependencies

- QEMU
- VDE2
- ngrep
- socat

C. Configuration reference

The `pf` configuration file consists of rules that follow one after the other, with any number of whitespaces between them. A rule consists of a header and criteria as follows (square brackets denote and optional value, an asterisk means zero to infinity repetitions of the string preceding it):

```
rule_header {
    [criterion1[,
    criterion2]*]
}
```

The configuration file format uses whitespace to separate its tokens (lexical units). These are only mandatory when they divide two alphabetic strings (for example, the tokens `dest_addr` and `equal` require a whitespace between them, but `dest_addr` and `=` don't). While the whitespaces are optional, it is recommended to use them for better readability. It is allowed to use spaces, tabs and newlines with any number of repetitions.

C.1 The rule header

The header's purpose is to specify the rule's action and other parameters that are more related to the packet filter than to the packet itself. It must start with an action and can be followed with flags separated by whitespaces. Some actions have arguments that modify their behavior. These are listed in parentheses after the action's name and separated by commas.

The following is the general structure of a rule header:

```
action[(argument1[, argument2]*)][ flag]*
```

C.1.1 Actions

Action	Description
`block`	Drop the packet.
`pass`	Accept the packet.
`reject(code, code6)`	Reject the packet with an ICMP message. Uses type 3 and code `code` for IPv4 and type 1 and code `code6` for IPv6. Both arguments are optional.
`forward(addr, port)`	Forward all packets matching this rule to address `addr` and port `port`. Rules that use this action must contain a `dest_addr` and `dest_port` criterion.
`masquerade(addr)`	Masquerade packets matching this rule (coming from an internal network) using NAT, so that they appear to originate from `addr` (which must be an address that belongs to the firewall).

Action	Description
	The rule itself doesn't block traffic destined to the internal network.

C.1.2 Flags

Flag	Description
`in` (`out`)	Match inbound (outbound) traffic. Anything forwarded by or sent directly from the firewall is considered outbound traffic. Packets destined to one of the firewall's addresses are inbound traffic. This behavior changes slightly when the rule contains the `interface` criterion. The default direction is `out`.
`through`	Match traffic on ports that are mapped to a different combination of address and port. The packet is validated after its header is altered. This flag cannot be used together with `in` and `out`.
`continue`	Only use this rule for packets that don't match any other rule.
`log`	Log any packet that matches this rule.
`nostate`	Packets accepted on behalf of this rule don't open a connection (they are not recorded by the state table).

C.2 Criteria

Criteria are conditions a packet must satisfy for a rule to match. They are specified by a type (the property the criterion matches), one or more aguments (the value to which the criterion compares the corresponding value in the packet) and a matching type (the way the property is compared to the arguments).

A criterion has the following form, where `!` means a negation of the criterion:

```
criterion_type [!]match_type argument1[ argument2]*
```

C.2.1 Criterion types

Name	Matching types	Allowed protocols	Description
`interface`	`equal`	All	If the rule has the `in` flag, translates to `src_addr subnet MASK`, where `MASK` is the subnet address set on the interface whose name is supplied as argument to this criterion.

Name	Matching types	Allowed protocols	Description
			If the rule has the `out` flag, the criterion translates to `dest_addr`. When the address is reconfigured on the interface, this criterion is **not** reloaded.
`src_addr` (`dest_addr`)	`equal`, `subnet`	All	Matches the source (destination) IP address of the packet. Supports both IPv4 and IPv6. The argument must be a textual representation of an IP address. The `subnet` matching type requires a subnet address, such as `10.0.10.0/24`.
`src_port` (`dest_port`)	`equal`, `greater`, `between`	TCP, UDP	Matches the source (destination) port in the transport layer header of the packet. The argument is a numeric value between 1 and 65535.
`proto`	`equal`	All	Matches the transport layer protocol. Allowed arguments are either protocol numbers or their textual representations - TCP, UDP, ICMP or ICMPv6 (case insensitive).
`size`	`greater`	All	Matches the size of the data in the IP packet (including e.g. the transport layer header). The argument must be a numeric value.
`icmp_type`	`equal`	ICMP	The ICMP type. The argument must be a numeric representation of the type.
`icmp_code`	`equal`	ICMP	The ICMP code. The argument must be a numeric representation of the code.
`icmp6_type`	`equal`	ICMPv6	The ICMPv6 type. The argument must be a numeric representation of the type.
`icmp6_code`	`equal`	ICMPv6	The ICMPv6 code. The argument must be a numeric representation of the code.
`tcp_flag`	`set`	TCP	Matches if the packet has the TCP flag specified by the argument, which should be either a numeric value (the bit offset in the flags field, counting from bottom - 0 means SYN), or a textual representation such as SYN, ACK or RST.

C.2.2 Matching types

The following is a list of possible matching types. Some matching types have alternate forms that can be used as shortcuts. The forms only differ in appearance and possibly readability.

Name	Alternate forms	Description
`equal`	`=`	Checks whether the argument equals the value in the packet.
`greater`	>	The criterion is satisfied if the value in the packet is greater than the argument.
`between`	None	True if the value in the packet is between the criterion's arguments.
`subnet`	None	Checks if the address in the packet is part of the subnet supplied as the argument.
`set`	None	True if specified flag is set in the packet.