Université catholique de Louvain
École Polytechnique de Louvain
Département d'Ingénierie Informatique

Supporting IPv6 host-based multihoming (shim6) in Linux firewalls

Promoteur:
PROF. OLIVIER BONAVENTURE
Lecteurs:
PROF. GILDAS AVOINE
SÉBASTIEN BARRÉ
DAMIEN LEROY

Mémoire présenté en vue de l'obtention du grade de **master ingénieur civil en informatique** option **networking and security** par
CHRISTOPH PAASCH

Louvain-la-Neuve
Année académique 2008-2009

Contents

1 Introduction **1**

2 Overview on firewalls and shim6 **4**
- 2.1 Firewalls 6
  - 2.1.1 Stateless Firewalls 6
  - 2.1.2 Stateful Firewalls 7
  - 2.1.3 Stateless vs. stateful Firewalls 8
- 2.2 Shim6 9
  - 2.2.1 Protocol description 11
  - 2.2.2 Message format 16
  - 2.2.3 Security considerations in shim6 16
  - 2.2.4 Firewall considerations 17

3 Shim6 and firewalls: Problem statement **18**
- 3.1 Design of the shim6-firewall 19
  - 3.1.1 Shim6 support in a stateless firewall 19
  - 3.1.2 Switching locators with a statefull firewall 19
  - 3.1.3 Tracking a shim6 flow 20
- 3.2 Problems with the shim6 protocol 27
  - 3.2.1 ULID-option in the initiation messages 27
  - 3.2.2 No shim6 state available 28
  - 3.2.3 Context collision 30
- 3.3 Conclusion 34

4 Description of the implementation in the Linux firewall **35**
- 4.1 The netfilter architecture 35
  - 4.1.1 Connection Tracking 36
  - 4.1.2 IPtables 37
- 4.2 Implementation design 38
  - 4.2.1 Tracking shim6 as a layer-4 protocol 38
  - 4.2.2 Tracking shim6 independently from the netfilter connection tracker 39
  - 4.2.3 Final solution 39

4.3 Implementation details 41
4.3.1 Extension header framework 42
4.3.2 Layer-4 shim6 handler 44
4.4 Conclusion 50

5 Evaluation of the shim6-firewall in the Linux Kernel **52**
5.1 Setup 52
5.2 Test description 52
5.3 State creation 54
5.3.1 Comparing the initiation messages 55
5.3.2 Comparing subsequent messages 56
5.4 Sending shim6 payload messages 58
5.5 Firewall stressing 59
5.6 Conclusion 60

6 Configuring a shim6-firewall **61**
6.1 Firewall configuration for multihomed hosts using shim6 61
6.1.1 How to allow shim6 traffic 62
6.1.2 Filtering on a shim6 flow 64
6.1.3 Firewall blocking the new locator pairs 65
6.1.4 Conclusion 66
6.2 Adoptions to the iptables implementation 66
6.2.1 IPv6 extension header match 67
6.2.2 shim6-specific module in ip6tables 67
6.2.3 Matching on the ULIDs 67
6.3 Conclusion 68

7 Conclusion **69**

A Code of the shim6-firewall **74**
A.1 Patches to Linux Kernel 2.6.24 74
A.2 Files modified 75

B Contributions to related projects **76**
B.1 LinShim6 76
B.2 Linux Kernel 77
B.3 iptables 82

List of acronyms

CGA Cryptographically Generated Addresses

CT Context Tag

DNS Domain Name System

DoS Denial-of-Service

FII Forked Instance Identifier

FTP File Transfer Protocol

HBA Hash Based Addresses

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

ISP Internet Service Provider

MTU Maximum Transmission Unit

NAT Network Address Translation

nonce Number used once

RAM Random Access Memory

RCU Read-copy Update

REAP Reachability Protocol

RFC Request for Comments

TCP Transmission Control Protocol

UDP User Datagram Protocol

ULID Upper Layer Identifier

Chapter 1

Introduction

The last years the Internet has become the most important communication system. People want to be online anywhere and at any time. More and more devices, from the usual desktop computers and notebooks to cell phones and PDAs, come with the possibility to connect to the Internet.
However, the current Internet is not able to handle the growing number of devices connecting to it. Thus, a newly designed Internet Protocol is necessary. This new Internet Protocol, called IPv6, allows the increasing number of devices to connect to the world wide web. As IPv6 is a complete redesign, it has been enhanced with several new features.

Multihoming consists in the connection of a device to several access points. These access points are connected to an Internet Service Provider who gives access to the Internet. A host can be multihomed in two different ways.
The host itself can connect to two different access points. For example, a notebook can be connected to the wired home network and a public WiFi. Thus, each of the two interfaces (the wired and wireless network cards) have access to the Internet. The different access points provide each an IP address to the interfaces. Thus the host owes several IP addresses and can connect to the Internet using both addresses.
Another way of multihoming is the multihomed site. A site can connect to several Internet Service Providers. This is often done to have a more reliable Internet connection. If a Provider has problems to provide Internet access, it is possible to use one of the other Providers to reach the Internet. Each Provider assigns an IP address prefix to the site. The site will provide these prefixes to the hosts. With the different prefixes the hosts have several IP addresses that can be used.
Multihoming allows the hosts to use their different IP addresses for a connection. A mechanism is needed to control the usage of these different IP addresses. One of these is the shim6 protocol, that uses the different IP addresses for a connection of a host. Thus, it provides a more reliable connection for multihomed hosts.

With the growth of the Internet, security issues are also becoming more important. Several mechanisms are deployed to secure the Internet. A firewall is one of these.
A firewall can be seen as the moat of a medieval castle. It has to separate the inside of the castle from the external world. It restricts the access to the castle and also controls the people who want to leave it. It can also prevent the castle to be *"flooded"* by people or accessed by evil guy's. Thus, the purpose of an Internet-firewall is to prevent unwanted traffic to pass.
Firewalls need to be configured to prevent this traffic. Rules need to be expressed to prevent different connections to the Internet. The firewall creates a state for every connection that hosts make. With this state the traffic can be controlled. The state maintains several informations about the connection. One of these is the IP addresses used by the communicating hosts.

It appears clear that the combination of multihoming and firewalls will create problems. As previously mentioned, the state of the firewalls maintain information about the IP addresses of a connection. However, in a multihomed environment, hosts use different IP addresses for a connection. Thus, the usage of these different addresses for the same connection will be a problem for the firewall. The firewall will need to maintain correctly its state, even if the IP addresses of a connection are changing.

This Master Thesis aims at supporting the host-based multihoming protocol (shim6) in the Linux firewall.

Contributions of this Master Thesis

Up to now, there exists an implementation of shim6 in the Linux Kernel, called LinShim6 and maintained by Sébastien Barré. However, there is still missing shim6-support in firewalls. The goal of this Master Thesis is to support IPv6 host-based multihoming (shim6) in the Linux firewalls.
With this document comes a prototype that enables shim6-support in the Linux firewall. The problems that shim6 poses to firewalls are studied in detail. The shim6-firewall implementation provided with this document solves these issues. The protoype is designed to fit nicely in the Linux firewall framework. Attention has also been granted to the efficiency of the firewall. The prototype is required to not introduce an unacceptable delay into the handling of the packets.
This document also provides recommendations to properly configure the prototype for filtering in a multihomed environment. As hosts are using shim6, specific filtering rules are required.

Structure of this document

This Master Thesis describes the work that has been done, by starting with the theoretical parts and then going more and more into the technical details of the implementation, while ending with the practical deployment and configuration of the shim6-firewall in a real world environment.

Chapter 2 describes the basics of the shim6 protocol and gives an overview of firewalls and their functions.
The following Chapter 3 first gives an overview of the problem that a multihoming protocol like shim6 poses to a firewall and suggests the design of the solution. In the second part of this chapter we also discuss modifications to the shim6 protocol that could aid firewall developpers.
Chapter 4 goes into the details of the implementation, starting with an overview of the netfilter design in the Linux Kernel and continuing with the presentation of the different solutions that were considered to implement the design. But will end with the detailed description of the final implementation architecture for the shim6 support in the netfilter connection tracker.
The evaluation of the implementation will be described in Chapter 5 where a complete performance test of the firewall will be described.
The final chapter will present some recommendations for network administrators who need to configure a shim6-firewall in a mutihomed environment.

Chapter 2

Overview on firewalls and shim6

As the IPv4 address space is getting smaller and smaller, the new network protocol IPv6 is being more and more employed in the current Internet. This new network protocol uses a larger address space by using addresses of 128 bits. But that is not the only novelty in IPv6. IPv6 is a complete redesign of the old IPv4 protocol and so has several new features to bring up.

The IPv6 protocol facilitates the multihoming of a site with the IPv6 address autoconfiguration protocol [Donz 04]. Multihoming can be based on hosts or on sites. A multihomed host has several IP addresses, even when using only a single interface. A multihomed site has several links to the Internet and at least one IP address space. The purpose of multihoming is mainly to provide a more reliable Internet connection because the use of a different IP address may result in a different route taken by the packet. Thus the broken link may be bypassed and so the Internet connection is more robust to failures of the network. There are different protocols that can use these multiple IP addresses or links. In the case of a multihomed site, the goal is achieved by BGP and routing protocols. The multihomed host needs also a mechanism that manages the different IP addresses. A protocol still in development, and that will be the subject of this document, is the shim6 multihoming protocol [Nord 09]. Shim6 gives the ability to a host to use different IP addresses in the case of link-failure, without disrupting the established connection. The shim6 protocol will be explained in more details further in Section 2.2.

Another important point in networking is the security. As the Internet gets bigger and bigger, the number of potential attackers also increases. Several protection mechanisms are employed. These secure the Internet from attacks intended to disrupt the correct functioning, or they protect confidential information. One of these protection mechanisms is the well known firewall. The purpose of a firewall is to control access to a site or host and thus, to prevent unwanted packets from passing and avoid denial-of-service attacks. The firewall will be explained in more details in Section 2.1.

Firewalls can be located at different points in a network. They can be installed on a host to

control the access to the host. They can also be placed in border routers of a site to secure the access to the site.
In Figure 2.1 we can see a multihomed site that has access to the Internet by using two different links, which may be due to the fact that the site is connected to two Internet Service Providers. The site contains several workstations and a server, interconnected by a router. The firewalls in the site are represented by brick walls. The server and a workstation are protected by firewalls. The border routers, that give access to the Internet are also equipped by a firewall to control the general access to the site.

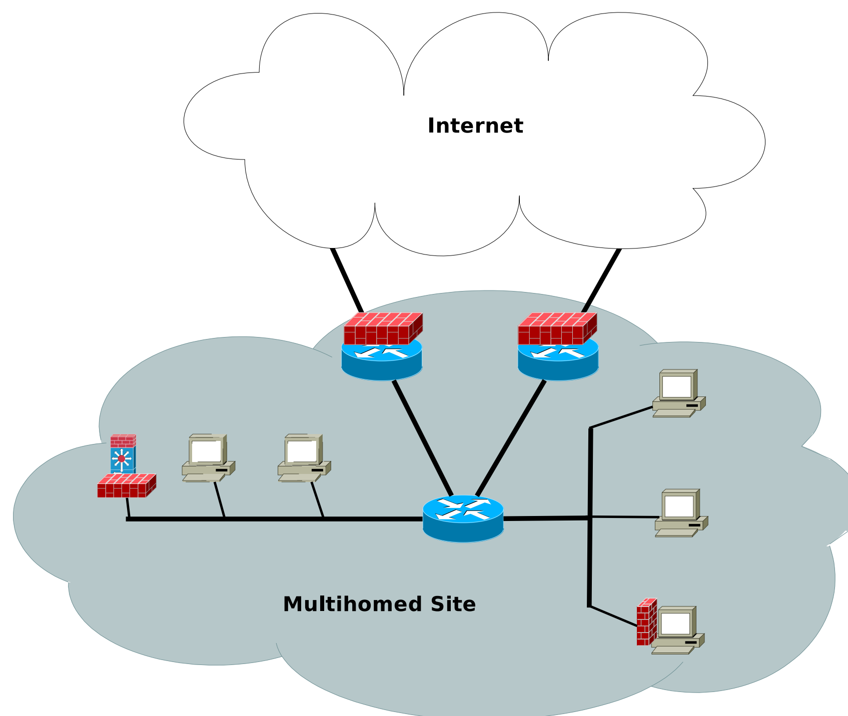


Figure 2.1: Example of a multihomed site with two firewalls on border routers and two on hosts

With the unstoppable and necessary deployment of IPv6, and the upcoming multihoming of sites and even hosts, it is also necessary to secure those accesses of the multihoming protocols (like shim6). Firewalls need to be adopted to support those new features. The next sections will explain the basics of the different concepts coming up in this document.

2.1 Firewalls

The function of a true firewall is to prevent the propagation of fire between two areas. A more convenient way to define a software firewall would be to view it like the moat of a medieval castle. It has to separate the inside of the castle from the external world. It restricts the access to the castle and also controls the people who want to leave it. It can also prevent the castle to get *"flooded"* by people or accessed by evil guy's. So, its purpose is to prevent the passing of malicious packets/information, that will cause some trouble on one of the two sides [Zwic 00, Ches 03].

There exists several kinds of firewalls, particularly the stateless and stateful packet filter:

Stateless packet filter This kind of firewall uses basic filtering on information available in the IP packet. Some packet filters also look at the layer-4 protocols, like TCP and UDP. So, the packet filter drops or accepts packets based on their IP addresses, port numbers, and other information contained in the packet.

Stateful packet filter With the stateful packet filter, the firewall assigns each packet to a flow of packets. That way, the firewall maintains a notion of connection for every packet (e.g. the well-known TCP connection for TCP-packets).

So, the general purpose of a firewall is to prevent packets passing to or from a specific IP address, port number, to a specific host name, or other information contained in the packet (e.g.: `iptables` allows to filter based on TCP-flags, and many other [Kis 09]). Another goal of a firewall is to prevent denial-of-service attacks (e.g. TCP-syn-flooding) and to prevent malformed packets to pass the firewall.

2.1.1 Stateless Firewalls

A stateless firewall, also called packet filter, sees the packets passing its network architecture. For every packet it applies the rules that got specified by the firewall administrator. Based on these rules the firewall decides to drop or allow the packets. The firewall administrator needs to express the security policy of the site in firewall-rules, so that the last one can meet the security needs of the site.

The rules can be expressed based on static information contained in packets:

- Source and destination IP addresses
- IP protocol version

- Source and destination ports
- Transport protocol
- Packet size
- ...and many other, based on the information contained in the different headers.

So, with a stateless firewall we could filter on any information contained inside a packet. We can easily block packets with spoofed IP addresses going outside to the Internet. Spoofing an IP address implies that the sender forged the source IP address, which means that the sender does not hold the IP address he is pretending to. We can also block incoming spoofed packets which have a source address that belongs to the site's address space.
Configuring a packet filter is a difficult task which is quite error prone. First, one has to identify what should and should not be permitted. Then, the rules must be specified in terms of the different packet fields. This is the difficult task, requiring deep knowledge of the different protocol headers for specific rules. After that, the rules can be expressed in the specific syntax of the used firewall[Zwic 00].

2.1.2 Stateful Firewalls

In general, a stateful firewall has the same capabilities as a packet filter. We can express a bunch of rules that define which packets are allowed or will be dropped. But the stateful firewalls, also called dynamic packet filters, have a new ability added to their behavior.

A stateful firewall associates each packet to a state. The firewall makes this decision based on several fields of the packet, namely the IP addresses, IP protocol version, Transport protocol and the port numbers (if present). That way a TCP-connection will have a state representation inside the stateful firewall. But also UDP-flows or ICMP packets will create a state inside the firewall.
The stateful firewall stores some useful information for each state, namely the status of the connection (e.g.: `NEW`, `ESTABLISHED`), number of bytes or packets exchanged, ...So the firewall administrator can also filter on these informations that are associated to the packet and thus has a much more modular tool to express the security policy of the site. The next part will show the benefits of a stateful firewall in the case of a TCP-connection.

2.1.3 Stateless vs. stateful Firewalls

One obvious question would be, what are the benefits of a stateful firewall, regarding the more complex implementation due to the maintenance of a state. For TCP, we can compare stateless vs. stateful firewalls for the case of the netfilter firewall.

With a stateless firewall we can only filter based on the information contained in the TCP-header. So, it is possible to block packets of a specific size, with a certain set of TCP-flags set, and so on. We could imagine to block all incoming SYN-packets without the ACK-flag set, to prohibit incoming connection requests. Of course, we can also filter based on the port numbers and IP addresses to prohibit a certain service or spoofed IP addresses. With the time, the number of possible attacks on the TCP-protocol increased. For example we could imagine to make a port-scan without the SYN-flag set as described by U. Maimon [Maim 96]. But also, as the policy of a firewall should be to only let pass those packets that are essential for the correct functioning of the protocols, a stateful firewall is needed.

With the stateful firewall, we can have a more precise filtering for incoming or outgoing TCP-connection requests. The netfilter connection tracker creates a state for an established TCP connection, and tracks useful information about this connection. That way, the firewall is able to better fulfill the purpose of a firewall. As the policy of a firewall should be to drop everything that is not explicitly allowed, the stateful firewall can better protect the protocol that is used by the tracked flow.

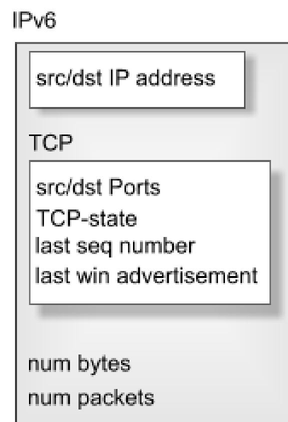


Figure 2.2: Simplified view of the state maintained by the connection tracker in the case of a TCP connection

In the case of TCP, the netfilter firewall tracks the sequence numbers and window size of the TCP connection to have maximal information about the running TCP connection. Thus being able to mark packets that are out of sequence or don't respect the window

size [Rooi 00] (a summarized view of the state-information stored by the connection tracker can be viewed in Figure 2.2). So the firewall increases the protection of the underlying protocols, by only letting pass those packets, that are absolutely necessary for the correct functioning of the protocol and so reduces the window of possible attacks.

2.2 Shim6

Shim6 is a protocol that offers the ability to use several IP addresses of the multihomed hosts, so that the connection to a peer has failover and load sharing properties. As got expressed by David D. Clark in *"The Design Philosophy of the DARPA Internet Protocols"*:

> *"The most important goal on the list is that the Internet should continue to supply communications service, even though networks and gateways are failing"* [Clar 88]

Shim6 tries to meet this goal as a layer-3 protocol that supports multihoming on top of IPv6. Shim6 manages the usage of the different IP addresses assigned to the host on which it is employed. With the handling of the different IP addresses, shim6 is able to provide a reliable connection if one of the IP address pairs fail. Shim6 is designed to have minimal impact on upper layer protocols, like TCP, UDP, ...
Currently there exists a shim6 implementation named *LinShim6* [Barr 06, Barr 08a]. *LinShim6* supports most parts of the shim6 protocol definition. But there are some minor parts of shim6 which aren't yet implemented and so these parts won't be covered in the following [Barr 08b].

An ordinary TCP connection uses several identifiers to identify a connection. Those identifiers are the IP addresses and port numbers used in this TCP connection. Those IP addresses will be called Upper Layer Identifiers (ULID), as they will represent the identifiers for the upper layer protocols. The purpose of shim6 is to ensure reliability of the connections identified by those ULIDs. This goal is achieved due to the multihomed host, that has several IP addresses which will be called locators. The locator is in fact the real IP address that will be carried by the packet, and the ULID is the identification for the upper layer protocol.
It is those locators that will be used by the shim to change the IP addresses of the connection. This may change the route of the packet and thus maybe bypass the broken link on the path. In Figure 2.3 we can see the general use case of shim6. The two hosts (A and B) are communicating over an established connection using their ULIDs (respectively `2009:BBBB::1234` and `1111:BBBB::1234`). At a certain point in time, the Provider 2, giving access to the `2009:BBBB` subnet, experiences a link failure, and host A can-

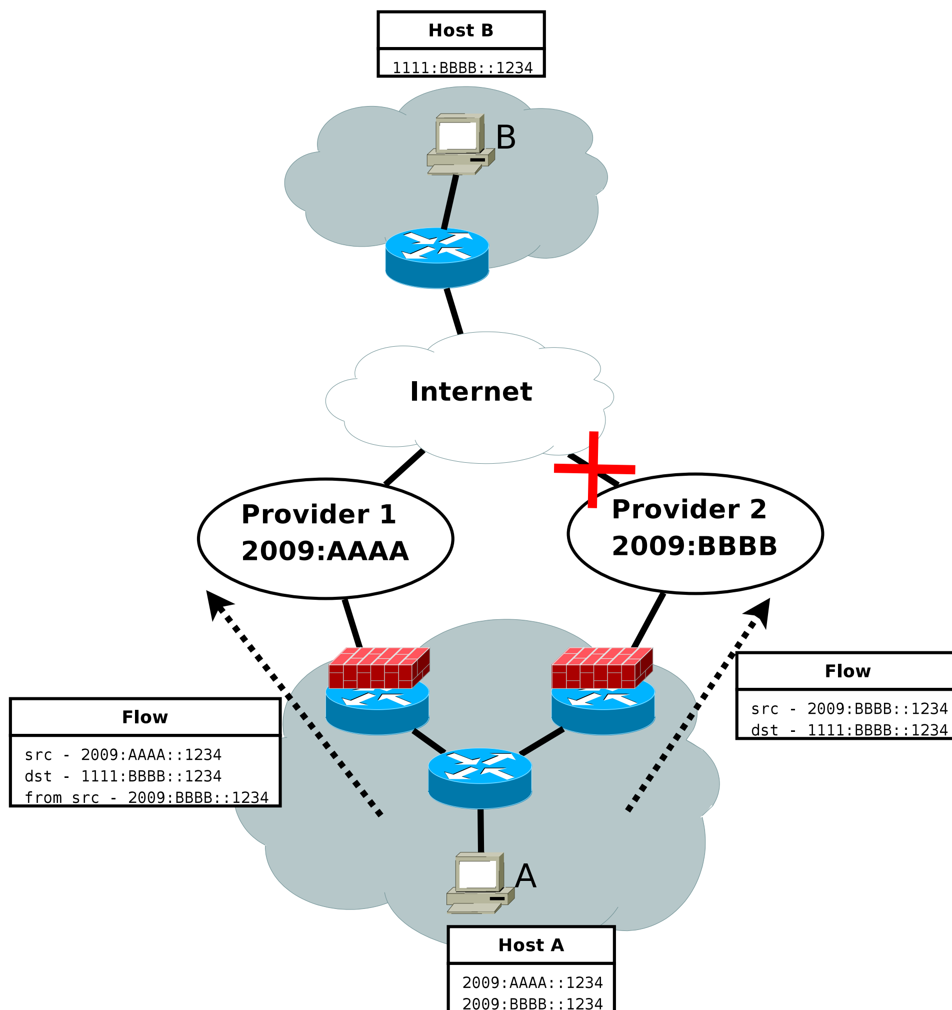


Figure 2.3: Shim6 - locator change due to a linkfailure

not anymore use Provider 2 to contact host B. At that point, the shim6 layer detects that linkfailure and decides to change the locators. As the host A is connected to two ISPs, shim6 can choose another IP address and switches the communication to using the locators A:`2009:AAAA::1234` and B:`1111:BBBB::1234`. But, as the goal of shim6 is to have minimal impact on the upper layer protocols, and so, the established connection should not be interrupted, the newly exchanged packets need to have an indication of the previous ULIDs that were assigned to that flow (`from src` in the figure). This indication is the so-called context tag. It is a 47-bit value included inside the shim6-protected flow.

2.2.1 Protocol description

The previously described goal is achieved by shim6 by establishing the so-called shim6 state. This state regroups certain informations to properly switch the locators while preserving the upper layer flow of packets. The functioning of the shim6 protocol will be explained in this part of the document.

Shim6-context establishment

A shim6 state is established when there exists a flow of packets between two hosts and one of the hosts decides to establish the shim6 state. The shim6 state is established using a four-way handshake, which means that no state is created upon reception of the first shim6 state-establishment packet. Figure 2.4 shows the four-way handshake of the shim6-context establishment.

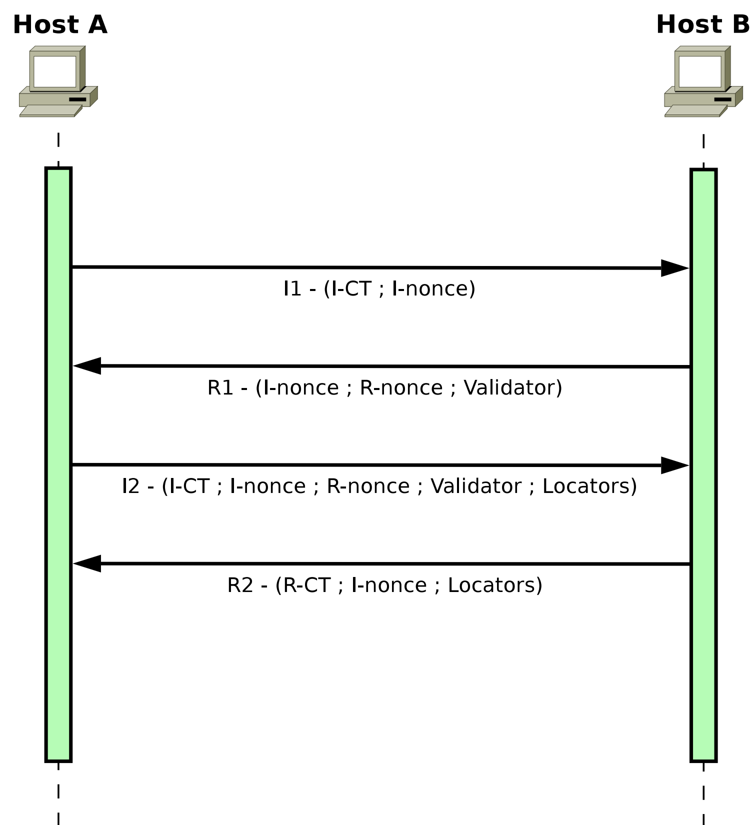


Figure 2.4: Four-way handshake shim6-context establishment

I1 The first message, that is starting the shim6-context establishment contains the context tag (`I-CT`), chosen by host A. The context tag will be the identifier for the shim6-

context for packets going from host B to host A. The `I-nonce` is used to prevent replay attacks and to prevent an offpath attacker from interfering with the establishment.

R1 The reply-message due to an incoming I1 message, will contain the given `I-nonce`, a chosen `R-nonce` (which is a counter) from host B, and a validator, which is a hash of the received context tag, `R-nonce` and a secret of host B. At this time, host B does not create a state for the shim6-context establishment.

I2 In this message, host A will provide its set of locators to host B. Host A also chooses an `I-nonce` and provides the given `R-nonce` to host B. The validator is a copy of the one received from host B. With the validator, host B can check if the I2 message is in fact the response to an outgoing R1 message.

R2 The final message of the context establishment includes the context tag as chosen from host B (`R-CT`), and the locators it wants to provide to host A. It still includes the nonce from host A, to prevent the above mentioned attacks.

The context tags chosen by each host are the one they expect to be included inside the shim6 protected flow. If a host needs to send TCP-traffic using the locators, he needs to include the context tag that the destination has chosen.

After the context establishment, both hosts know the locators provided by their counterparts, and the context tags they are “listening” to. The four-way context establishment assures, that host B only creates a state after receiving the I2 message and successfully checking the provided validator and `R-nonce` (as the nonce is a counter, host B can check, if the nonce is in the valid range of possible nonces). This is a way to avoid half-open contexts on the receiver-side (as TCP-connections use a three-way handshake, they are vulnerable to those attacks, well known as the SYN-flooding attack).
We could imagine, that two hosts would decide at the same time to start a shim6-context establishment. And so, the hosts will receive incoming I1 messages, while they have already send an I1 message. In that case, the two hosts will immediately reply with an R2 message to provide their locator list. Figure 2.5 shows the concurrent context establishment of a shim6 connection.

For shim6 it is also possible to establish a state using the locators as the IPv6 addresses in the packet. This can be useful, if the host does not manages to establish a context using its ULID. So, the shim6 context establishment messages support the ULID-option, which specifies the ULIDs that should be associated to this shim6 state. The ULID-option will be included in the I1 and I2 messages. The R1 and R2 messages do not contain this ULID-option.

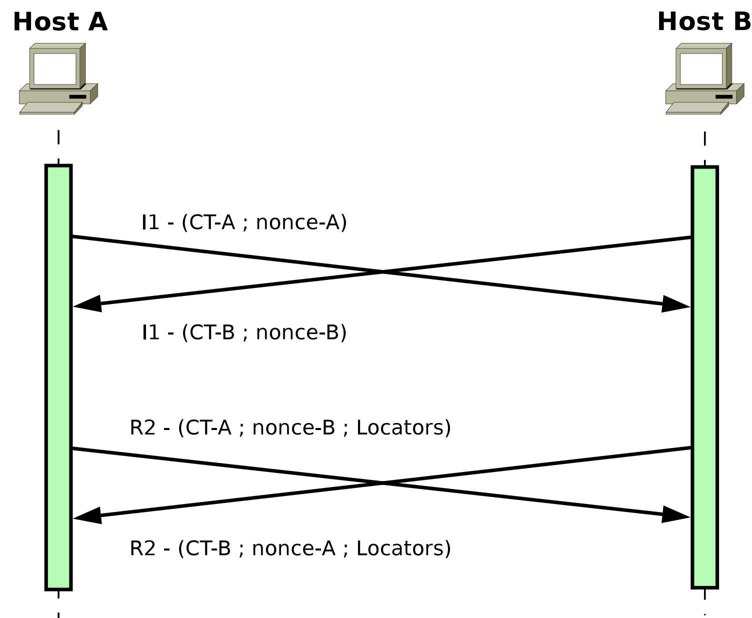


Figure 2.5: Concurrent context establishment of a shim6 context

Locator updates

It is possible for a host, to update its currently used locators or the locator preferences. This is done by the update-request messages. A host sends an update request message, containing the receiver context tag, a nonce, and the new locator list to the corresponding host. The receiver of the request will answer with an update acknowledgment message, containing the receiver context tag and the received nonce.

Forked Instance Identifier

If two hosts want to establish two shim6 sessions using the same ULIDs they can do it using the so-called FII-option (Forked Instance Identifier). This way, the hosts can protect different upper layer protocols by different shim6 sessions (if e.g., they want to protect the VoIP by another locatorset than the TCP). The I1 and I2 messages do contain this FII to seperate the different contexts.
The current LinShim6 implementation does not supports the FII-option [Barr 08b].

Failure detection

Detecting a path failure is done by forced bidirectional detection. In fact, this assumes, that for every incoming packet, there should also be some reply traffic. That way,

the protocol can conclude, that the connection is bidirectionally working [Arkk 06]. So, the reachability detector can be in several states:

Bidirectional traffic If the host can see bidirectional traffic, it is obvious that everything is fine.

Unidirectional traffic In the case, that the host only sees traffic in one direction (e.g.: UDP-streams), for example incoming traffic, it needs to ensure that the reply direction is also working. This is done, by sending keepalive messages (containing the receiver-context tag) in the "idle" direction. So, the host receiving those keepalives knows that the connection is still bidirectional.

No traffic at all As there is no traffic, the hosts don't expect reply traffic, and no keepalives are sent.

Failure recovery

When one of the hosts has detected a failure, it will start the failure recovery protocol as defined in the IETF reap-draft [Arkk 06].

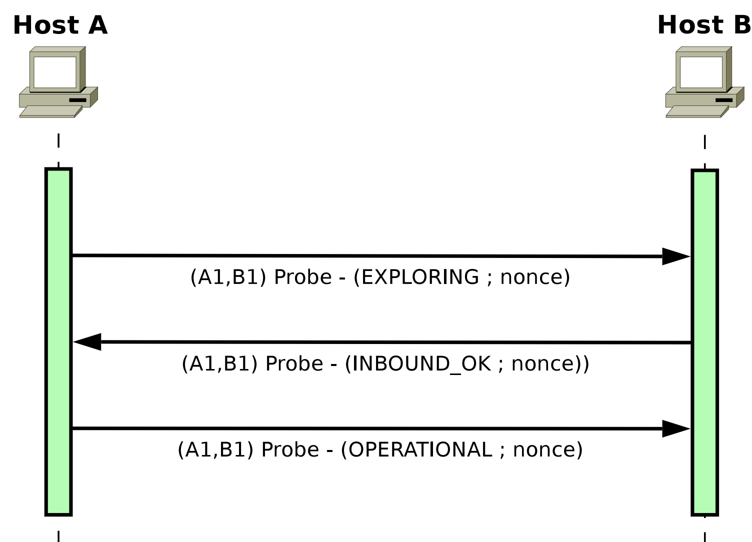


Figure 2.6: Successfull path exploration

In the Figure 2.6 is shown a simplified view of the reachability protocol. First, host A chooses another locator pair (A1,B1 in this case) and sends a probe message to host B. The nonce in the robe message is used to protect against the attacks mentioned in Section 2.2.1 and identify the robe. In each robe, a host puts the nonce of the current probe as well as the list of received and sent nonces since the beginning of the exploration. The state of

the probe message is set to EXPLORING. When B receives a probe message it will reply to it with another probe message, containing the received nonce, and setting the state to INBOUND_OK, to indicate to host A, that it correctly received the previously sent probe message. As host A receives the last probe message, it knows that a bidirectional working locator pair was found, and so answers to host B by sending an OPERATIONAL probe message. After that, the shim6 context switches to the newly allocated locator pair.

State recovery

Shim6 allows the hosts to recover from a lost state. A state might be lost due to an early timeout of the shim6 state on one of the hosts. Thus, if a host receives a shim6 message for which it does not have a state associated it can request the necessary information from the peer to recreate the state.

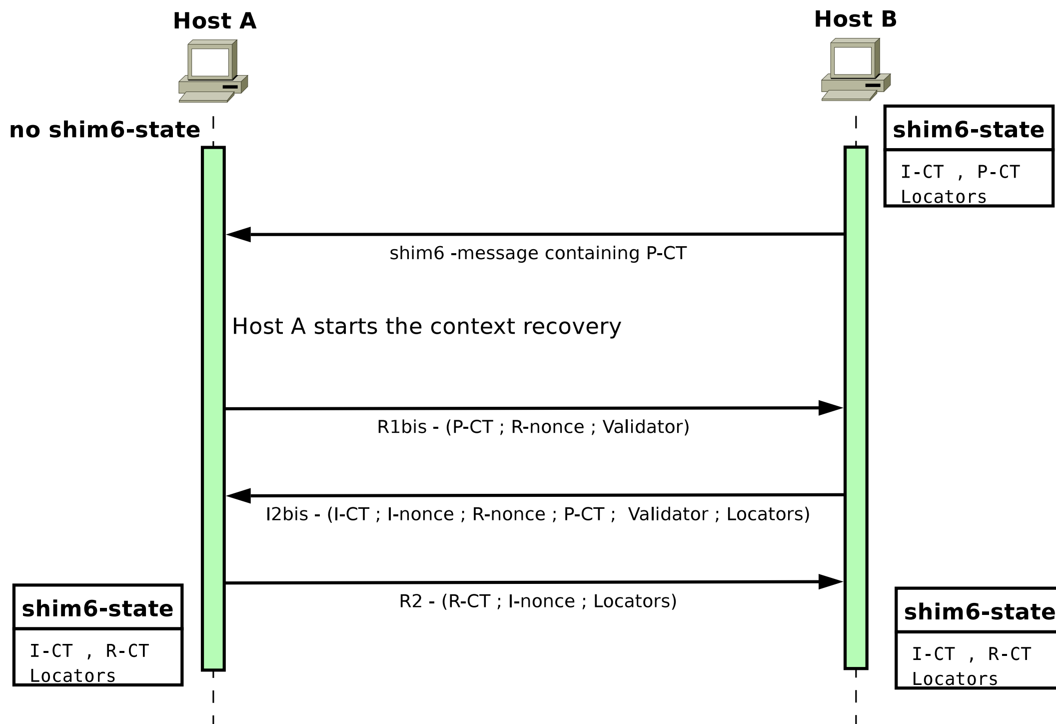


Figure 2.7: Shim6 state recovery

Figure 2.7 illustrates the state recovery of host A where host B did not lost the state. This host received a shim6 message from host B containing the context tag P-CT who is unknown for host A. So, this one asks to recreate the state by sending the R1bis message to host B. This one answers with an I2bis message that contains the context tag chosen by

B and the P-CT (so, these two context tags may be the same). Host A answers with an R2 message, and so the state got successfully recreated on host A.

Shim6 payload messages

When the REAP protocol successfully changed the locators, the underlying layer-4 traffic, that is "protected" by shim6 needs to be redirected by the shim6 layer. This is done by rewriting the source and destination IP addresses of the communication to the newly allocated locators. The context tag needs to be included in the message, to enable the receiving host to identify the received packet to the original ULIDs. The included context tag is the one that has been chosen by the destination.

2.2.2 Message format

As previously mentioned, shim6 uses the generic IPv6 extension headers, to identify itself as a shim6 payload or control message. There is one unique protocol number for the both types of the shim6 messages, and one bit inside the header to identify if it is a payload or a control message. Figure 2.8 shows the format for the generic shim6 extension header. If `P` is set to 1, the packet belongs to a shim6 payload message. If it is 0 it is a shim6 control message.

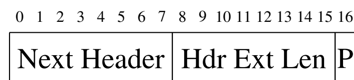


Figure 2.8: Format of a shim6 payload extension header

In the case of a shim6 payload message, the 47-bit context tag is directly included after the `P`-bit. That way, the overhead of the shim6 extension header is set to the minimum extension header size of 64 bits.

For the shim6 control messages, the octet following the `P`-bit identify the type of the control message. The rest of the extension header are specific to each different shim6 message type. Further details on the specific message formats of each shim6 message type can be seen in the IETF shim6-draft [Nord 09].

2.2.3 Security considerations in shim6

Shim6 tries to resolve the security threats described in RFC4218 *"Threats Relating to IPv6 Multihoming Solutions"* ([Nord 05]). This section describes how the shim6 pro-

tocol meets with the security considerations from the RFC (Section 16 of the shim6-draft [Nord 09]).

Off-path attackers To prevent off-path attackers to interfere with the shim6 protocol, shim6 uses the context tag and nonces to secure its shim6 control messages. In fact, the context establishment messages all use nonces and subsequent messages use the context tag to prevent the off-path attacker from sending shim6 messages. The attacker needs to know the nonce and context tag to inject messages. This is only possible if he can sniff shim6 messages and thus needs to be on the path. But it is an explicit non-goal of shim6 to protect against on-path attackers as every on-path attacker can highly interfere with any protocol used.

Protection against DoS attacks As got previously described, the four-way handshake of the shim6 context establishment protects against DoS attacks aimed to exhaust the state of the receiving host (like a SYN-flooding attack).

Locator protection To prevent an attacker from modifying the locator list, shim6 uses Hash Based Addresses (HBA, [Bagn 08]) or Cryptographically Generated Addresses (CGA, [Aura 05]).

HBA These type of addresses link each others prefixes by using a hash. This hash is included inside the addresses, so each address contains the information about the prefix-set that got advertised. An attacker cannot add an IP address to the list as its prefix is not included in the hash of the other addresses.

CGA Cryptographically generated addresses link a public key to the addresses. With the locator list of CGA addresses comes a signature that ensures the integrity of these addresses. As for HBA, the attacker cannot change the list of CGA addresses as he does not possesses the correct private key.

2.2.4 Firewall considerations

As shim6 changes the locators of the packetflow, this may disturb the working firewall on the host or edge of the local network. The connection tracker will need to switch the established TCP connection state to another pair of locators. This is where we will consider to change the netfilter implementation. The next chapter describes in more details the problems that shim6 will bring to the firewalls.

Chapter 3

Shim6 and firewalls: Problem statement

Shim6 enables hosts to change the locators they are using for a flow. Another IP address in the packet flow may have the effect that the packets take another path through the network. If a site has several access points, there may be separated firewalls on each of these access points. Thus, these firewalls will not see all the packets corresponding to a specific flow. This kind of configuration is also known as distributed firewalls [Lero 06]. It is out of the scope of this document to go into the details of the distributed firewalls and their problems with shim6.
This document will focus on the case where the firewall either is the only one to protect the site, or is installed on the shim6-host. Thus, this firewall will see all packets passing.

A stateful firewall creates a state for each packetflow. As shim6 modifies the locators that communicating hosts are using for a flow, the stateful firewalls need to be adopted. To understand the necessary change, a small description of the architecture of stateful firewalls is needed: Firewalls are divided in two parts. The connection tracker and the packet filter.

The purpose of the connection tracker is to create and manage the state of each packetflows. The packet filter usually filters based on the information contained in the packetheaders (IP addresses, . . .). But it can also filter on information that is part of the state associated to the packet. This kind of filtering is the difference between a stateless firewall and a stateful firewall.
As a flow is associated to a pair of ULIDs, these must be stored in the state of the connection tracker. Thus, the state in a shim6-firewall will no more be associated to the locators inside the packet but rather to the ULIDs associated to the flow.

This chapter will explain in details the problems and changes necessary to firewalls. The first section justifies the changes that were necessary to the firewall to support the shim6 protocol. It will describe the problems related to stateful firewalls and how the shim6

session is tracked by the stateful firewall.
The second section will describe in details the problems that remain for the firewall to fully support the shim6 protocol. In particular several changes to the shim6 protocol are suggested to allow firewalls to handle shim6.

3.1 Design of the shim6-firewall

This section explains our design decisions to implement support for shim6 in a stateful firewall. The different problems that a stateful firewall poses to the shim6 protocol (and vice-versa) are explained and how these lead to the final design decision.

3.1.1 Shim6 support in a stateless firewall

The stateless firewalls, which were explained in Section 2.1.1, filter based on the information contained in each packet. Shim6 generates new types of packets, which must be taken into consideration by the firewall administrator. This depends on how, the firewall processes the packet and whether it recognizes the shim6 extension header or not.

The firewall must be able to recognize the upper layer protocol encapsulated by the shim6 payload extension header and filter on this protocol.
As the shim6 control messages will not be followed by any other extension header or transport protocol, the firewall should allow the administrator to filter on different parts of the shim6 extension header (e.g.: specific shim6 message types, . . .)

3.1.2 Switching locators with a statefull firewall

This is depicted in Figure 3.1. Host A and B have an established TCP connection and a shim6 session between their ULIDs A1 and B1. At a certain moment in time, shim6 triggers the reachability protocol and changes the locators of the shim6 session, so that the IP address fields of the packets from the TCP connection now contain A2 and B2. Shim6 adds to each packet the shim6 payload extension header containing the context tag for that shim6 context.

The problem is, that if the firewall does not support the shim6 extension header it will not associate the packet containing A2 and B2 as locators to the original TCP flow. If the firewall supports state recovery in the middle of a TCP connection it will create a new state for these locators A2 and B2. Otherwise the firewall will simply drop these packets. Thus the TCP connection is broken.

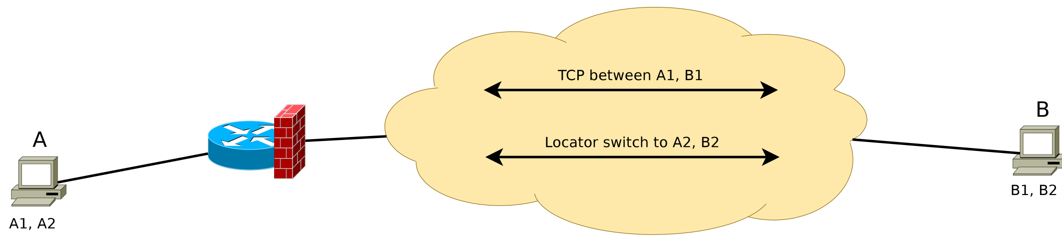


Figure 3.1: Standard use case of a firewall facing a shim6 locator switch

All the information the stateful firewal will associate to the transport protocol will be out of synchronization with the original flow. These may be number of bytes or packets exchanged, state of the connection (e.g., `NEW`, `ESTABLISHED`, . . .) and sequence numbers and window size in the case of TCP.

The last point is the most substantial issue. Let us imagine that shim6 switches back to the original locators of the flow. If the TCP connection tracker of the firewall has not already discarded the original state for the TCP connection (due to a timeout), the packets will be out of sequence and so will be dropped if the connection tracking subsystem of the firewall filters on out-of-sequence acknowledgment and sequence numbers. Thus, the firewall must associate the flow of packets that contains a shim6 payload extension header to the original flow of packets. Solutions to these problems will be presented in the next section.

3.1.3 Tracking a shim6 flow

The goal of a shim6 aware connection tracker is to associate a shim6-protected flow to its original IP addresses (ULIDs), in order to continue to correctly track the state of the associated protocols.

Figure 3.2 represents how a shim6-firewall should handle a shim6 payload message. The received packet contains IP addresses A2, B2 but the shim6-firewall will match these to the corresponding ULIDs (A1, B1) that are associated to the context tag ct_B. This tuple (containing the ULIDs) that the firewall created from the packet will be used to retrieve the associated state. As the firewall retrieves the ULIDs based on the context tag from the shim6 header, the firewall needs to track the shim6 context establishment to be aware of the context tag for the specific flow of packets.

To properly handle this, the firewall needs to create an explicit state for the shim6 context in its state machine. This state needs to carry all the necessary information to track

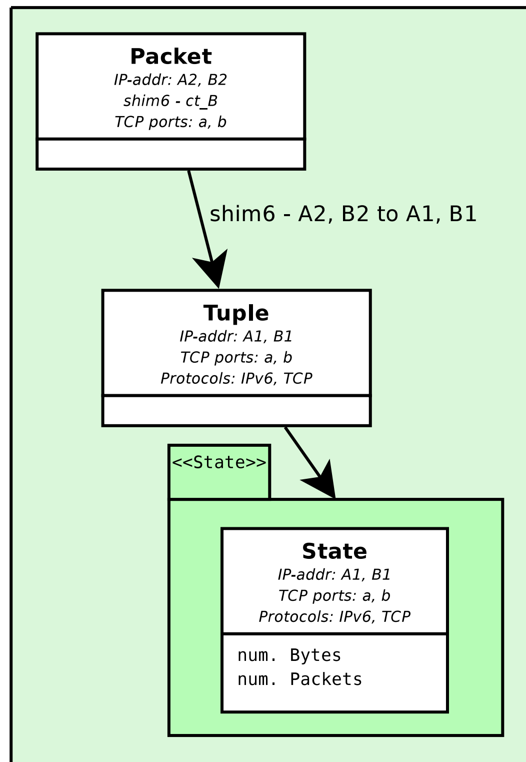


Figure 3.2: State representation of the shim6-firewall

the locator switches and properly associate a context tag to the right ULIDs. Following is a discussion on how to track the shim6 session.

Tracking the shim6 context establishment

Figure 3.3 shows the shim6 context establishment who has been explained in detail in Section 2.2.1.
It uses a four-way handshake to limit the risk of DoS attacks on the hosts (SYN-flooding attack on TCP). Thus, the firewall should also create its shim6 state upon the reception of the I2 message on.
If the state was created upon arrival of the I1 message, the firewall could track and verify the nonces of the first messages. For this, the firewall would need to perform the same work as the end-hosts. However the goal of the shim6-firewall is not to reimplement the complete shim6 protocol, but mainly to map the context tags to a specific ULID-pair.

If the connection tracking of the shim6 context establishment starts upon reception the I2 message, we can imagine several scenarios which could pose a problem for the firewall. An attacker could try to perform a Denial-of-Service attack, by sending context es-

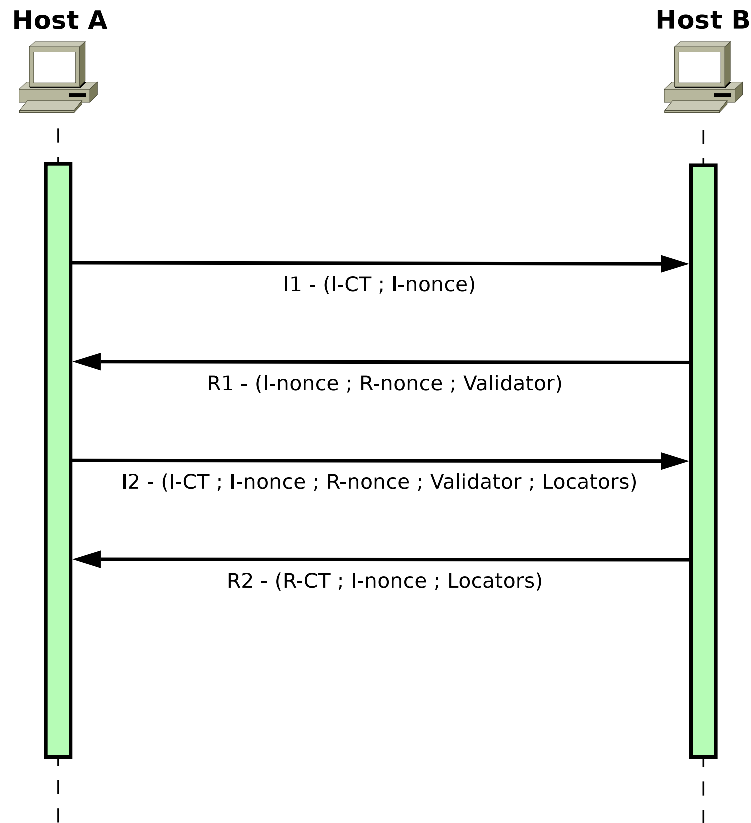


Figure 3.3: Four-way handshake shim6-context establishment

tablishment messages which will put the firewall on a different state than the endhosts. It is obvious that this will be a problem for the information exchange between the original hosts, as the firewall will not have the same information (context tag, locator list, ...) as the hosts. Following are three of the main problems that will arise if the firewall starts its connection tracking from the I2 message on.

Host privacy for the site It is common practice from the network administrators of a site to prevent all incoming connection requests from passing the firewall. This is often done to protect the hosts from unsolicited packets, but also for privacy issues. The administrators want to prevent attackers from gaining any information on the topology of their network. But also they want to prevent the attackers to discover which software runs on which machines.
An easy way to make such kind of attacks harder is to block all incoming packets that do not belong to an existing flow. In the Linux netfilter firewall this can be achieved by using the following rule (assuming that `eth0` is the interface connected to the Internet):

```
ip6tables -t filter -A FORWARD -i eth0
          -m state --state NEW -j DROP
```

As explained earlier, the state machine will recognize any packet that does not belong to an existing connection as being in the `NEW` state. Thus, the firewall will correctly meet the security goal by dropping all incoming, `NEW` packets.

If a firewall starts its connection tracking of the shim6 context establishment upon reception of the I2 message, the previous messages (I1, R1 and R1bis) will not be tracked and will be in the `UNTRACKED` state (see also Figure 3.4). As one can see in the figure, the I1 and R1 messages will pass the firewall, even if this one is blocking incoming `NEW` packets. Thus the attacker could discover, which machine inside the site is up, and that it has a shim6 implementation running. The firewall administrator could find this annoying, and a security concern.

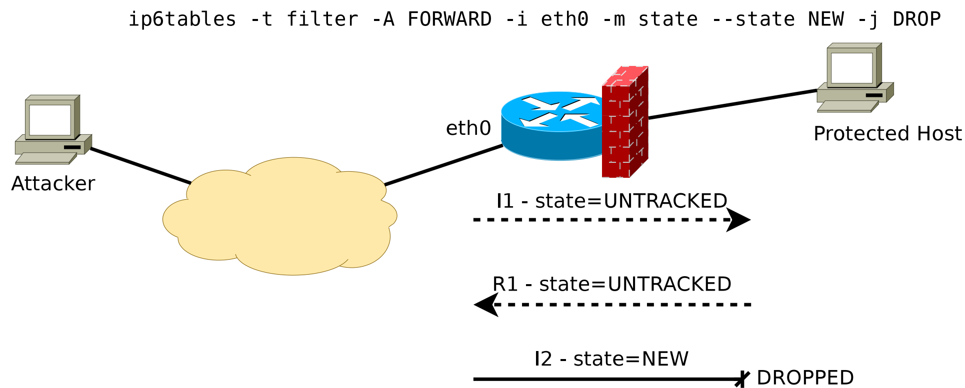


Figure 3.4: Privacy issue of a firewall blocking NEW packets

A solution to this might be to drop not only the incoming `NEW` packets, but also incoming I1 messages. That way it is ensured that no unwanted traffic passes the firewall. So, only hosts inside the site can initiate a shim6 context establishment.

Problems with concurrent context establishment A problem may arise with the previously described configuration of a firewall that sees the shim6 packets passing. If the firewall blocks incoming `NEW` packets and I1 messages, the concurrent context establishment (as described in Section 2.2) will be blocked. This is illustrated in Figure 3.5 .

Figure 3.5 reveals that no shim6 context will be established, due to blocked I1 and R2 messages. Both hosts start the shim6 context creation at the same time by sending an I1 message. The incoming I1 from host A will be blocked by the firewall rule that prevents incoming connection establishment requests. However the outgoing I1 will reach host A.

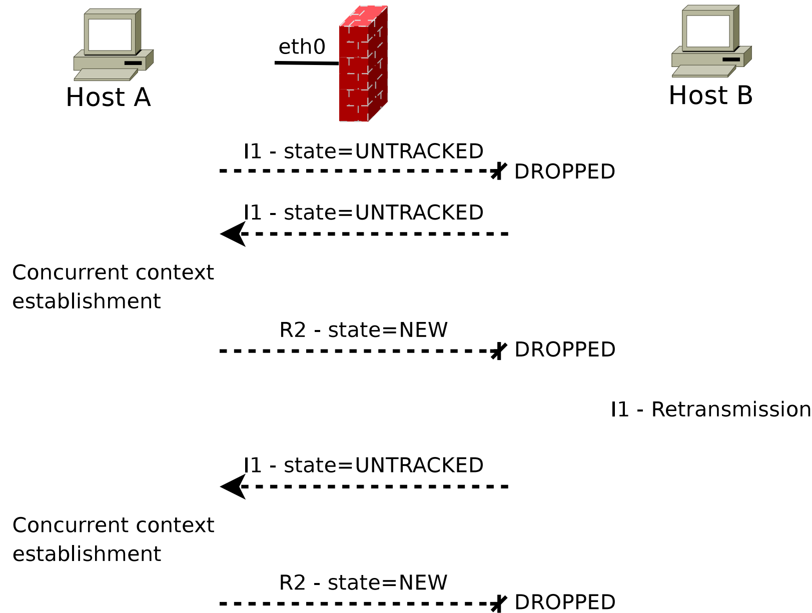


Figure 3.5: Blocked I1 message prevent concurrent context establishment

As host A is in the `I1-Send` state it will think that it must trigger the concurrent context establishment procedure. In this case, host A will send a R2 message in response to the incoming I1 message. These R2 message will be blocked by the firewall, because they will be in the `NEW` state as they are the first shim6 context establishment messages that will be tracked (I1 messages are untracked).
On the other side, host B has sent an I1 message and is waiting for the R1 reply. As it does not receive this R1 message, nor the I1 messages coming from host A, it will retransmit its initial I1 message. This will be performed several times until host B recognizes host A as not being part of the shim6-enabled hosts and will cache this information.

One condition for this to happen is a similar timeout configuration for both shim6 hosts. The context triggering heuristic must be similar and the I1 maximum retransmission timeout also needs to be similar. This case could be quite common in practice, as the shim6 draft gives recommendations about the timer configurations.

The previously described problem for the context establishment applies only for a specific configuration of the firewall and the two concerned hosts. The firewall must block `NEW` packets and I1 messages. The hosts need similar timer configurations and shim6 context triggering heuristics. Another argument is that if the firewall administrator blocks incoming I1 messages he knows that this will interfere with the shim6 context establish-

ment and so must be conscious about this fact. So, this problem might be seen as rather uncommon and needs to get reevaluated with the appearance of more hosts and firewall administrators applying shim6.

Denial-of-Service attacks on the firewall Denial-of-Service attacks on the firewall imply that the attacker manages to put the firewall on a state that differs from the state of the communicating hosts in such a way that the firewall would block subsequent packets that should pass the firewall. This way the firewall will block or interfere with the communication between the original hosts.

One possible attack on the firewall is due to the fact, that the shim6 draft says that upon the reception of an I2 message, a host should look for a corresponding pair of ULIDs in its state machine and then update the information of the state, *even if the state is already ESTABLISHED* (see section 7.13 of the shim6 draft, [Nord 09]). Of course the I2 message must respect all the other security checks (nonces, responder validator).
Shim6 allows a host to change the context tag of an established shim6 context (as long as the I2 message respects all verifications on the security parameters).

For an attacker to be able to execute this kind of attack, he needs to sniff the R1 message, so that he can place the right nonces and responder-validator in the I2 message. Thus, the attacker needs to be on the path between the two communicating hosts. But it is an explicit non-goal to protect shim6 against on-path attackers. And so, the attack is of no concern for the end-hosts.

But, as the firewall will only start its connection tracking after having seen the I2 message, the firewall will not be able to check if the I2 message contains the right nonces and responder-validator. The firewall needs to observer the I2 message and update the context tag and locator list according to the I2 message. Even if there exists already a shim6 state for these ULIDs.
An attacker can simply put the firewall on a different state than hosts by sending an I2 message. This I2 message needs to have the correct ULIDs in the IPv6 header. However the rest of the shim6 extension header can be arbitrary. Especially the modified context tag will be a problem for the firewall. The host who receives the I2 message will simply discard it, as it does not contain the right nonces. The firewall however has a different context tag in its shim6 states than the communicating hosts.

Tracking from the I1-message

With the previous announced problems when tracking the shim6 context establishment only from the I2 message on, it appears clear that it is impossible for the shim6-aware firewall to create the shim6 state starting at the I2 message. So, the firewall must track the state of the shim6 context from the I1 message on. This way the task for the firewall administrator is much easier, to express the rules for protecting the site (see Host privacy for the site, Section 3.1.3) and prevent the firewall from being in a different state than the communicating hosts (see Denial-of-Service attacks, Section 3.1.3).

Tracking from the I1 message on implies that the firewall needs to check the nonces of the shim6 context establishment messages. The firewall needs to always store two nonces during the context establishment phase (one for each direction). This is because messages might need to be retransmitted.

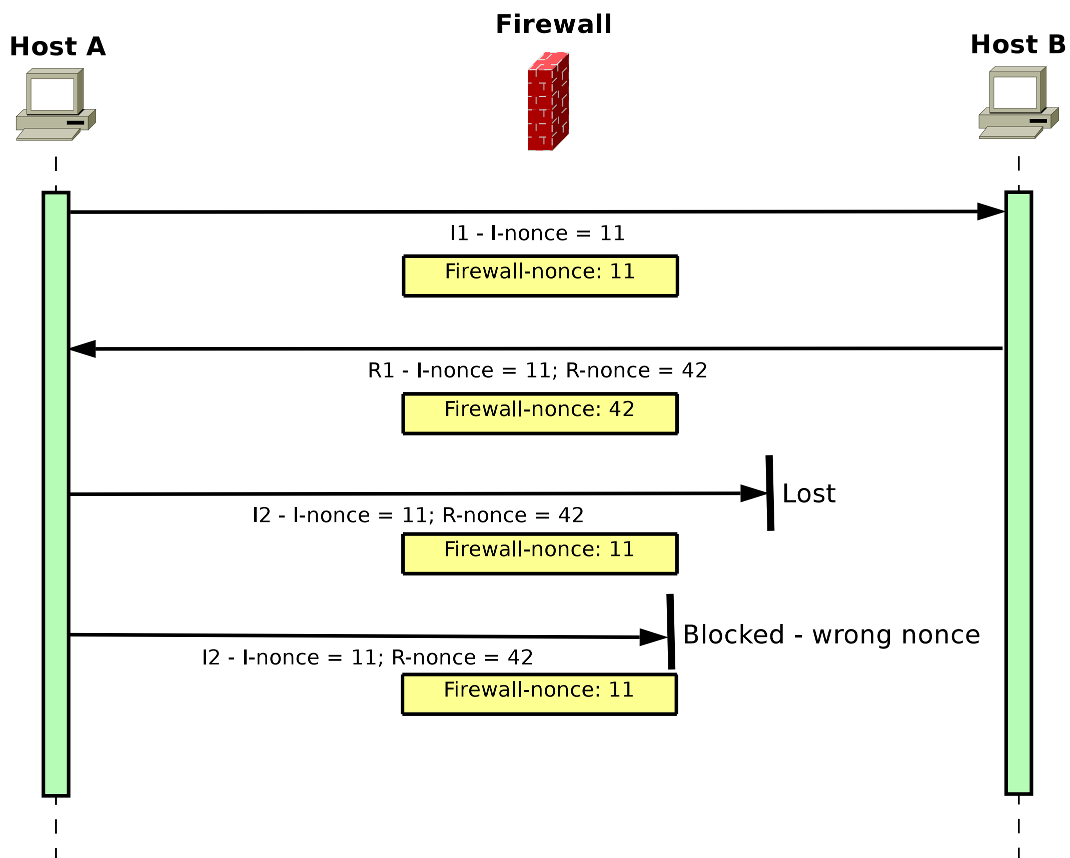


Figure 3.6: Example of a firewall only storing one nonce

In Figure 3.6 we can see a firewall that only stores **one** nonce in its statemachine. The first I2 message will be lost after having passed the firewall. The firewall correctly checked

the nonce for this I2 message (it was 42) and stores the new one (from the I2 message - I-nonce = 11). When host A does not receive the expected R2 from host B, it retransmits the I2 message. This one will be blocked by the firewall, because the nonces do not match. This is due to the fact that the firewall does not maintain nonce directions. This explains why the firewall needs to store bidirectional nonces.

Thus, the stateful firewall creates the shim6 state from the I1 message on. Verifying the nonces in each direction to prevent the previously mentioned problems. With this design, the basic shim6 protocol is supported. However some parts of the shim6 protocol need to be adopted to allow full support. This will be explained in the next section.

3.2 Problems with the shim6 protocol

This section discusses several problems that cannot be solved by the firewall implementation and which may require a change in the shim6 protocol specification.

3.2.1 ULID-option in the initiation messages

The firewall needs to map the locators to their corresponding ULIDs. In the shim6 context establishment messages (I1, R1, . . .), there is no context tag to map to the ULIDs. Usually the ULIDs are the IPv6 addresses included in the IPv6 header. But the shim6 protocol specification also allows the so-called ULID-option, which specifies the ULIDs associated to this context establishment (see Section 2.2.1).

As the R1 message does not contain the ULID-option it is impossible for the firewall to correctly map the R1 message (containing the locators in the IPv6 header and not the ULIDs) to the corresponding ULIDs. Hosts associate the receiving R1 message to the ULIDs using the included locators and nonces. But for the firewall this will be more difficult (but not impossible). The firewall would need to maintain a mapping between pairs of (locators,nonce) and the corresponding shim6 state. This will complicate the work for the firewall, as it is difficult to decide when to garbage-collect the mappings to save memory.

A solution to this problem is if the responses (R1 and R2) would also contain the ULID-option field. With this, the firewall can associate these responses to the right ULIDs. This change in the the shim6 protocol specification is only a small additional work for the hosts. Hosts receiving an I1 message (or I2), needs to insert an ULID-option inside its R1 message to inform the firewall of the correct ULIDs.

It should be noted that the same problem is present for the FII (Section 2.2.1). This one should also be included inside the responses to allow the firewall to maintain several shim6 states for the same pair of ULIDs.

3.2.2 No shim6 state available

Another possible problem for the firewall may be the case when the state machine of the firewall looses the state about an existing shim6 session. This can be due to several reasons:

- An early timeout of the firewall state: This may be due to different configurations of the shim6 timeouts on the firewall and on the communicating hosts.
- Firewall start up: The firewall starts up and so does not have any state yet.

If the firewall has lost the state of the shim6 session it can not anymore associate the shim6 payload packets to its original ULIDs. One might think that the firewall could associate this flow to the current locators, and so using these locators as ULIDs. But this will pose a problem as soon as the connection switches back to its real ULIDs. This packet flow will not have the shim6 payload extension header included in the packets, and so cannot be associated to the already existing flow.

For the case of a TCP state, the Linux firewall allows the connection tracker to recreate a valid state. This is possible because all the necessary information can be extracted from a TCP packet. For the case of shim6 this is not possible. If the firewall sees a shim6 payload message it cannot deduce the corresponding ULIDs. Thus another mechanism is needed to recover the shim6 state.

Shim6 state recovery on the firewall

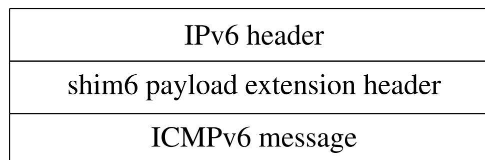
The problem is that the firewall will receive a packet containing a shim6 extension header with a specific context tag. The shim6 extension header can be a payload extension header or a control messsage. The firewall will not be able to associate the context tag to a specific pair of ULIDs, because it does not have any information about such an association. So it needs to recover this information. This can easily be done by asking one of the hosts to start a context recovery (see Section 2.2.1).

The shim6 protocol specification [Nord 09] should add support for this kind of error messages from the middleboxes to enable state recovery. This message can be sent by the firewall. Several types of messages could be possible:

Shim6-message The firewall could generate a new type of a shim6-message. This message should contain the context tag that the firewall found in the previous shim6 packet that it received. The source IP address should be that of the firewall interface and the destination should be the source address of the previous shim6 packet for which the firewall did not had a matching shim6 state. Since the firewall includes the context tag in the shim6 message, it proves to the receiving host that it is in fact on the path. So the host can simulate a context recovery exchange with the corresponding peer, that belongs to this context tag.
For this to work, an additional shim6 message format must be defined. The advantage of this solution is that it is completly integrated inside shim6 and does not use any other protocols.

ICMPv6 Another solution could be to create a new ICMPv6 message type. This ICMPv6 type would be used to indicate that a middlebox is having a problem with a specific context tag. So, the ICMPv6 message needs also to include the context tag that the firewall has seen. As ICMPv6 messages include the headers of the packet that caused the ICMPv6 [Cont 98], the context tag will be encapsulated by the ICMPv6 message. Upon the reception of this ICMPv6, the host extracts the context tag, and starts a context recovery with the associated peer by sending an R1bis message.
A drawback of this solution is that a new, ICMPv6 type must be defined. The hosts need also to listen on incoming ICMPv6 messages and check whether the type belongs to the new type of ICMPv6 messages. An advantage of this solution is that it is quite common for firewalls to send ICMPv6 messages. And so, this solution might more properly fit in the concept of a firewall. Another important advantage for this solution is that it is a generic solution. Thus this might also be used by other protocols (e.g., MIPv6, . . .). The new ICMPv6 message type can be used by any other protocol to indicate to the hosts that the firewall has lost the corresponding state. Thus the protocol needs to start a context recovery upon the reception of this ICMPv6 message.

Shim6-ICMPv6 A trade-off between the previously described solutions could be to send a shim6 payload message, with an encapsulated ICMPv6 message. So, packets generated from the firewall will have the following format:



The firewall will need to put the context tag inside the shim6 payload extension header. The ICMPv6 message type should be *Destination Unreachable* with the code *Communication with destination administratively prohibited.* When a host receives

this packet it will have to start the context recovery based on the context tag included in the shim6 payload extension header. The advantage is that no new message types need to be defined. The only change is that the shim6 protocol needs to start a context recovery upon the reception of such packets.

The goal of each of these different messages is to launch a context recovery so that the firewall can recreate its state based on the content of the R1bis, I2bis and R2 messages. As the host does not send the packet explicitly to the middlebox that announced the problem but rather to the peering host, it is ensured that this exchange can not be intercepted by an off-path attacker. So the hosts are protected against off-path attackers.

3.2.3 Context collision

One problem might be that shim6 hosts may choose the same context tag for different states and so the firewall will map the wrong ULIDs to the corresponding context tag.

In fact, as a firewall may be at the border of the site, it may protect the access of a huge number of different machines. Every machine may create shim6 states with different correspondent hosts, each one choosing a specific context tag. It may be possible that two different host-pairs may choose the same context tag. As the firewall will map a shim6 message to its ULIDs based on the context tag, it can not differentiate between two shim6 sessions with the same context tag. These shim6 messages will be mapped to the same ULID pair.
Figure 3.7 shows two host-pairs that have a shim6 state established with the same context tags. The firewall will only have one reference (context tag → shim6-state). For example, the firewall will match the context tag to the shim6-state associated to the hosts A and B. So, the shim6 payload messages from C-D will be associated to the corresponding state from A-B.
For example if a shim6 payload message coming from C is passing the firewall, it will replace the locators by those associated to the shim6-state of A-B. Thus the firewall will associate this message to the wrong state and possibly make wrong filtering decisions. A detailed analysis of the security concerns is explained later in this section.

Security Concerns It is important to analyse in detail the problems that a context collision will bring to the firewall. First there are concerns about the states maintained for each host pair.

- As the firewall cannot associate the flow to the ULIDs C-D, it might create a new state for the flow with the colluding context tag. This is because the new flow probably

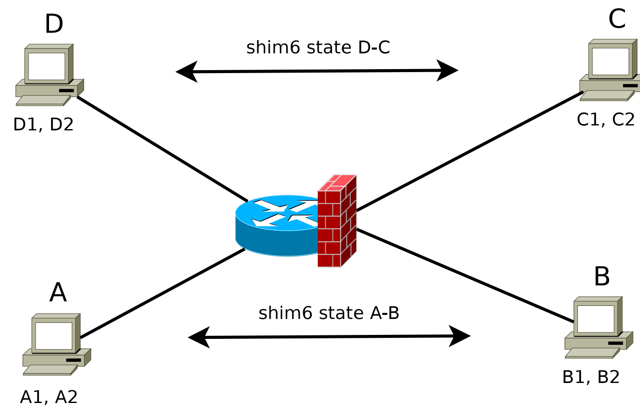


Figure 3.7: Two host-pairs having the same context tag for their shim6 states

uses other port numbers. The new state will be assigned to the ULIDs A-B. Thus the connection tracker is in a corrupted state. This might lead to further uncontrolled behaviour that is based on the different packets that might pass.

- In the case of TCP, the checksum verification in the firewall will fail. This is because the firewall will use A-B to compute the checksum rather than C-D. If the firewall is configured in such a way that invalid TCP checksums will be dropped, the flow will be interrupted by the firewall.
- The original state between C-D might be garbage-collected due to a timeout. Thus, when the shim6 session switches back to using the ULIDs the firewall will need to recreate the state for C-D.

Another concern is due to the shim6 protocol specification. Section 12.2 of the shim6-draft [Nord 09] says that the hosts do not check the locators in receiving shim6 payload messages. This means that any host can interfere with an existing flow of packets just by knowing the shim6 context tag. The attacker does not need to spoof its IP addresses. Thus, even if the site protects itself from spoofed IP addresses the attacker can inject traffic. The receiving host will simply map the context tag to the associated pair of ULIDs, without checking the locators. Thus accepting the spoofed packet.
The shim6 protocol justifies itself about this by saying that it does not protects itself from an on-path attacker. Which means that the attacker needs to be on the path to know the context tag.

By not verifying the locators on a shim6 payload message, shim6 simplifies packet spoofing.
Assuming that the attacker is on a site that filters on spoofed IP addresses. The attacker would not be able to spoof packets in a world without shim6. If the attacker can sniff a

packet and thus discovers the shim6 context tag, he is able to inject traffic. Hosts receiving this traffic will think that it comes from the valid peer.
TCP uses a checksum and sequence numbers. These will make it more difficult for the attacker to inject traffic. However it is not up to shim6 to rely on the security provided by upper layer protocols. There is only a minimal overhead for the shim6 hosts to verify the locators of shim6 payload messages. Shim6 should protect itself from packet spoofing, because shim6 allows spoofing even if the site filters on spoofed IP addresses.

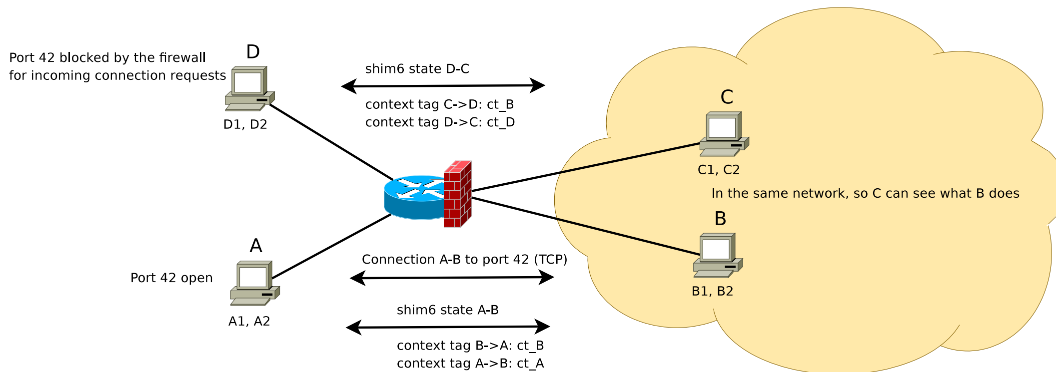


Figure 3.8: Firewall allowing the traffic that should be blocked

Another problem with context collision is the case depicted in Figure 3.8. In this example the firewall blocks the access to port 42 of the host D for incoming connection requests. Host C establishs a shim6 session with D, using the same context tag as host B has chosen for its shim6 session with host A. Host B has established a connection to port 42 of host A. With shim6, host C is able to establish a connection to host D's port 42. It simply sends a packet with a shim6 payload extension header to host D by using the same layer-4 protocol and port numbers as the connection betweend host A and B. As the firewall does not handles the context collision, it will associate this packet to the original connection between host A and B. Thus it will allow the packet to pass.
Again, one might argue that with TCP and the usage of sequence numbers, this attack is hard to execute. However again it is not up to shim6 to rely on the security that might be provided by upper layer protocols as these might change in time.

Solution A solution to the previously described problems might be to verify the IP addresses contained in the shim6 messages and check these locators to belong to the previously negotiated set of locators for the shim6-state. Thus the firewall will need to track the locators of the shim6 context establishment. Currently this is not done, due to several reasons. First, due to time-constraints there was not the time to implement this feature. But also due to efficiency concerns for the firewall.

The shim6 protocol specification does not give a limit on the number of advertised locators. But there are several mechanisms that limit this indirectly. For example the length of the locator list is coded into a 8-bit field. So, there is already a maximum of 256 locators. But when considering the MTU of a standard IPv6 packet (1280 bytes) it turns out that the maximum number of locators is 70.
One thing that also needs to be considered is that the reachability protocol uses exponential backoff for sending its probes. So, having a huge number of locators does not has any sense, because the shim6 connection would have timed out before having tried all the possible combinations of the 70 locators. The LinShim6 implementation applies an upper bound of 15 locators per host to the shim6 protocol.

With this bound of 15 locators it would be possible for the firewall to track the locators. To avoid the firewall of allocating memory at every locator update-message it should allocate the memory statically at the shim6 state creation. So, we need to foresee the maximum amount of memory needed.
First there needs to be place for 30 locators due to the standard shim6 context establishment (locators from the I2 and the R2 message). Secondly there needs to be place for the locator update messages. These messages allow the shim6 hosts to update their set of locators. The locator update mechanism is based on a request-acknowledgment mechanism. Due to this, the firewall cannot apply the update request directly and must store this one in a temporary memory and acknowledge this when it sees the update-acknowledgment.

So, this results in a total of $15 \times 2 \times 2 = 60$ locators to store. As was previously mentioned, the firewall should allocate this memory at the state creation, to avoid the expensive memory allocation during shim6-message flow after the state creation. As IPv6 addresses have a length of 16 bytes, this results in a total additional memory usage of at least 960 bytes. Additionally there need to be stored information about this locator list, pointers, . . . Thus, the additional memory can be estimated to 1kB. This memory overhead is not a negligible amount. The shim6 state machine in the firewall will consume a major part of the machines memory for the shim6 state. This needs to be further evaluated when it becomes clear, how many shim6 states a firewall needs to manage.

Something to consider with this setup is, when a host is sending consequent update-requests without having already received the acknowledgment. The firewall needs to always store the last update-request and discard unacknowledged update-requests, because a host will always try to enforce the acknowledgment of his last update-request. So, at the end it is always the last update that will get applied to the hosts.

3.3 Conclusion

This chapter discussed the different problems that the shim6 protocol poses to firewalls. The firewall needs to map the context tag to the associated ULIDs. To protect itself from DoS attacks the firewall needs to verify the nonces from the shim6 context establishment messages (Section 3.1.2). For doing so the connection tracker needs to create the state from the I1 message on.

However some parts of the shim6 protocol can not be supported. The shim6 protocol needs to be adopted for allowing the firewall to support these features. The ULID-option should be included in the responses (R1 and R2). Shim6 needs to support a mechanism to support context recovery for the firewall. The number of locators need to be bound to a reasonable size to protect the firewall from context collision. And shim6 should verify the locators from the shim6 payload messages for not giving an easier way to spoof packets.
If these changes will be done to the shim6 protocol, the firewall can fully support the shim6 protocol.

Chapter 4

Description of the implementation in the Linux firewall

The Linux Kernel includes a firewall, named netfilter. Since release 2.4 of the Linux Kernel [Russ 02a], the firewall also includes a stateful connection tracker, that is also part of the netfilter framework. This chapter will first present the general netfilter architecture and then the details of the implementation architecture.

4.1 The netfilter architecture

Netfilter is a framework that offers functionalities to treat packets passing through the Linux networking stack. Currently it is possible to handle packets at five different points of the IPv6-stack. Those are the so-called *netfilter hooks*. For each hook several packet-handling routines can be defined in the hook-structure (`nf_hook_ops`). For each packet passing, netfilter iterates over the different handling routines and calls them with the packet as an argument (see also Figure 4.1) [Wehr 04, Benv 06]. For example, to enable the IPv6 connection-tracking in the Linux Kernel, netfilter calls the function `nf_register_hooks` with the hook-structure that specifies the function `ipv6_conntrack_in` as the packet-handler. There are lots of different handling routines in the Linux Kernel. The `conntrack` enables stateful firewalling (see also Section 2.1.2), `NAT` adds the ability for Network Address Translation and `Filter` is the firewall-routine which iterates over the firewall rules.

Using this generic design of the netfilter architecture, there are different handling routines defined in the Linux kernel. Iptables is used to filter the packets. The connection tracker is also registered at the netfilter hooks. NAT (which is also implemented by netfilter) uses this connection tracker. Even the LinShim6 implementation takes profit from the

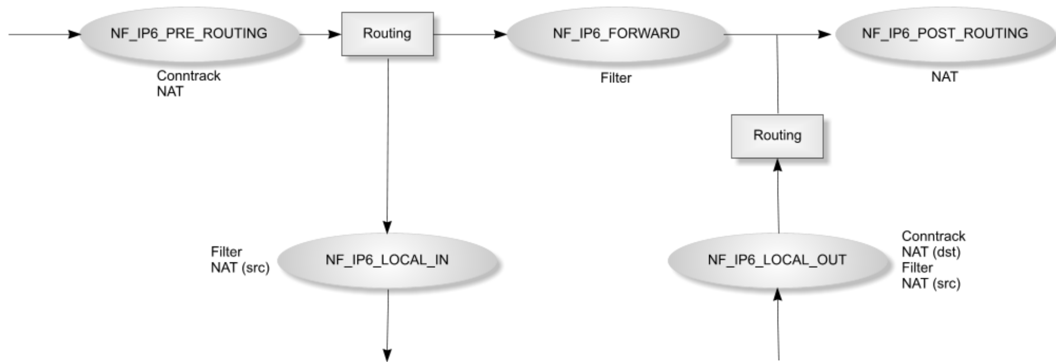


Figure 4.1: Netfilter Hooks

netfilter hooks, by registering the packet listener at the hooks `NF_IP6_LOCAL_IN` and `NF_IP6_LOCAL_OUT` [Barr 08a].

4.1.1 Connection Tracking

The connection tracker examines every packet passing the pre-routing and the local-out hooks. That way, it can create a list of connections passing through the Linux kernel. A connection in terms of the netfilter definition is not only the well-known TCP-connection. Netfilter associated to each of the layer-4 protocols a notion of state. [1]
Every connection has an initial direction, specified by the way this connection has been set up. In the case of TCP, the direction is specified by the initial SYN-packet. So, as a returning packet has source and destination addresses inverted, the connection table contains two entries for every connection. This is done, to speed up the lookup in the connection table. One entry is for the initial direction and the other one is for the reply direction.

The connection tracker registers into the netfilter pre-routing hook, by giving it the packethandler `ipv6_conntrack_in()` (see also Figure 4.2). This function is the starting point for every packet entering the connection tracker. For a non-fragmented packet it will pass the packet to the generic `nf_conntrack_in()` function.
As the connection tracker, and netfilter in general, is very generic and extensible, the specific protocol-handlers are all stored in generic structures, which must define the necessary packet-handler functions. So, the connection tracker first retrieves the IPv6 protocol-handler-structure. This structure defines a function (`ipv6_get_l4proto`), to look for the layer-4 protocol-handler. After identifying layer-3 & 4 protocols, netfilter tries to

[1] So, even the connection-less UDP protocol has an entry in the netfilter connection-table (in fact, a UDP-state is created when the first UDP-packet passes the firewall).

look for a corresponding connection in the connection table, and if not found adds a new connection-entry to the list. (`resolve_normal_ct()`)

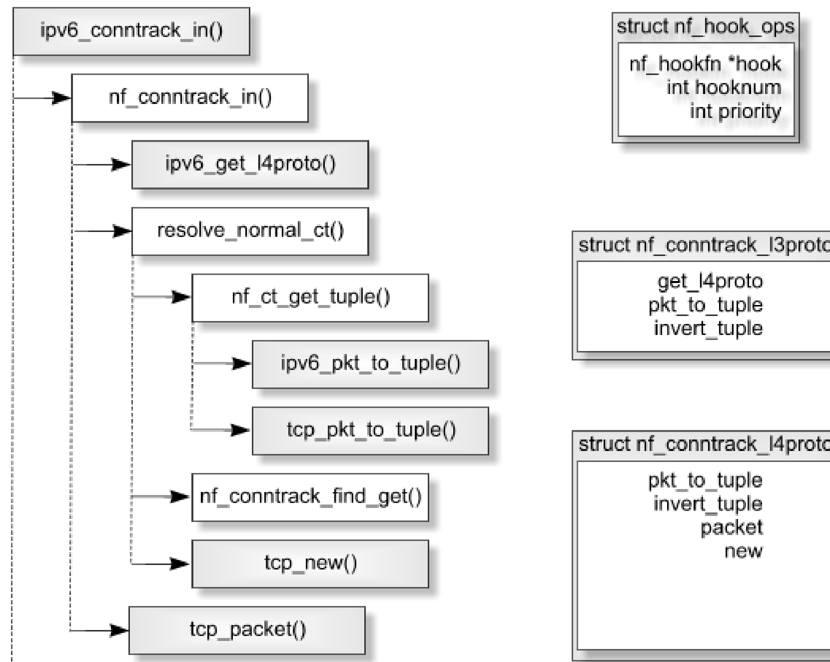


Figure 4.2: Call-graph of the IPv6 connection tracker

First of all, `resolve_normal_ct` will create a conntrack structure by calling the protocol-specific `pkt_to_tuple` functions. This new conntrack structure will be used, to find the corresponding entry in the connection table. The tracker matches over the tuple by checking the source and destination IP addresses, the layer-3 protocol and the layer-4 protocol. If there does not already exist an entry for that packet, it will create a new one, by calling the layer-4 specific `new` function, which will populate the connection entry with the appropriate data. There will also be an inverted entry in the table for the reply-direction. After having successfully created or found the existing connection, we update the connection status, and check if the packet is a valid layer-4 packet by calling the `packet` handler function of `nf_conntrack_l4proto`. The connection tracker finishes its work, by linking this connection entry to the initial packet buffer. [Boul 09, Ayus 06]

4.1.2 IPtables

The IPtables module is used for packet filtering and is divided into different parts, namely the standard iptables for IPv4, ip6tables for IPv6, arptables for the ARP-protocol, . . . The module xtables gathers the common parts of each of the previously men-

tioned modules. The structure of iptables is similar to that of the connection tracker, in that it also works with structures which define function pointers and register at the various netfilter hooks[Russ 02a, Russ 02b].

The IPv6 packet filter will start at `ip6t_do_table`. It first retrieves the table for the specific hook at which the method got invoked, and will then iterate over every entry in that table. For each entry, it first checks if the rule matches the IP-addresses, interface names and the layer-4 protocol (if specified in the rule). If those parts of the rule match, netfilter will check the rest of the rule, by calling the generic `IP6T_MATCH_ITERATE`, which will call the specific "match" function for every rule. In the case of a rule that specifies the state of a connection, the function `match` of `xt_state.c` will be called. This one checks the state of the packet and returns true if the packet's state is in an accepted state. When a matching rule has been found, netfilter will apply the specified target to the packet and return it to the initial caller of the iptables-module.

4.2 Implementation design

To implement the solution presented in the Chapter 3, several problems need to be solved. This section will first present the different implementation designs considered to support shim6 into the netfilter firewall and will finish with the final solution that got implemented.

The main problem for the implementation of shim6 support is the lack of IPv6 extension header handling in the connection tracker. IPv6 extension headers are situated between the IPv6 header and the layer-4 protocol headers. As the shim6 information is also contained in the extension headers, a solution needs to be found. But the current netfilter connection tracker just jumps over the extension headers, and there is no proper way to extract the data from the extension headers.

4.2.1 Tracking shim6 as a layer-4 protocol

One solution taken into consideration is to fully implement the shim6 connection tracking as a layer-4 protocol. To achieve this task, the previously mentioned function `ipv6_get_l4proto` (see Section 4.1.1) needs to return shim6 as the layer-4 protocol. So, the complete handling of the shim6 control or shim6 payload messages would be in that layer-4 module. The layer-4 handler would create a shim6 context to track the shim6 state and verify if the used locators in the packets are corresponding to the current locator pair associated to the shim6 state.

This solution would nicely fit into the netfilter connection tracking framework. In fact it would be enough to implement a new layer-4 protocol module for shim6, and registering the `nf_conntrack_l4proto` structure into the netfilter framework.

A major drawback of this solution is that the connection tracker will not be able to track the underlying layer-4 protocols included in the shim6 payload messages (e.g., TCP, UDP, ...). It is obvious that this would create some huge security holes, as a firewall is supposed to completely track the protocols to ensure the security for the hosts (as described in Section 2.1.3). The connection tracker should still be able to track the TCP-sequence numbers to protect the hosts from DoS attacks.

4.2.2 Tracking shim6 independently from the netfilter connection tracker

Another solution taken into consideration is to track shim6 completely independently from the netfilter connection tracker. This could be achieved by registering a new netfilter hook just before the connection tracker. The new hook would do all the major parts of the existing connection tracker, but only for shim6 messages. Thus it could properly track all the necessary parts of the shim6 extension headers and also do the connection tracking of the underlying layer-4 protocols, thus resolving the issue stated in the previous section.

The drawback of this solution is that it would be a heavy task to implement it, and so it would be quite error-prone. And as it is always good coding style to use the already existing implementations and frameworks, this solution was discarded.

4.2.3 Final solution

The final solution for the shim6 support is to create a new framework aimed at handling the IPv6 extension headers. With this framework, the shim6 extension header can be handled correclty, while still being able to track the upper layer protocols after the shim6 payload extension header.

IPv6 extension header handling

In the current version of IPv6 connection tracking, the extension headers are not explicitly treated. This presents several problems to extension headers, as stated in Kozakai et al. [Koza 07]. Especially for the Type 0 Routing Header extension, where the final destination address is included in the extension header.
The goal of the extension header framework is to parse the extension headers after the IPv6

handler but before any layer-4 specific handlers. The new framework should be called at different points of the connection tracking, as seen in Figure 4.2. As extension headers are not meant to represent a layer-4 protocol, and as the headers are a specific case of IPv6, it would not make sense to call the new framework in any other place than inside the IPv6 connection tracker. The extension header handler is called at two points in the IPv6 connection tracker. Namely at the end of the call to `ipv6_pkt_to_tuple` and `ipv6_get_l4proto`.

When receiving a call to the framework, it will iterate over the different IPv6 extension headers (actually Linux defines only the Hop-by-hop, Routing, Fragmentation, Authentication and Destination options header as extension headers) and parse the call to the corresponding extension header handler. Figure 4.3 shows the general architecture of such an extension header framework.

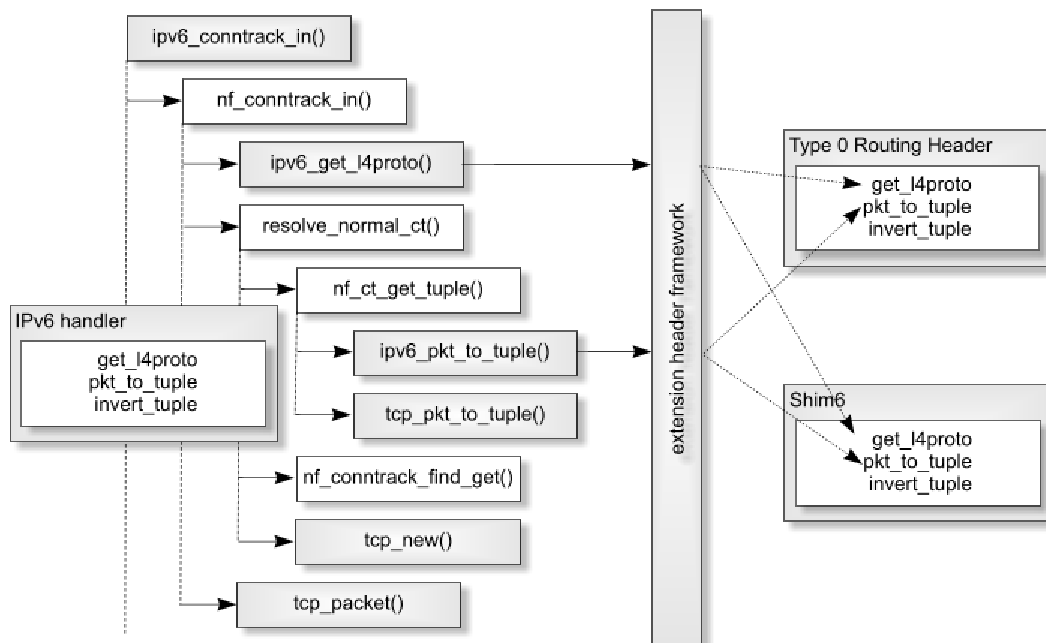


Figure 4.3: Overview of the extension header framework

Shim6 specific tracking

Having resolved the problem with the extension headers, it is time to specify the way how the netfilter extension for shim6 will achieve the goals stated in Chapter 3. As there are two types of messages for shim6 (control and payload), carrying different information and being handled differently by the hosts, those two also have to get handled in a specific manner.

Shim6 - the layer-4 protocol The shim6 control messages will be treated as a layer-4 protocol. This decision was taken, due to the fact that a real shim6 connection is established between two hosts, and so it is necessary to protect this connection establishment by the firewall to ensure maximum protection of the hosts. To be able to track the shim6 control messages like a layer-4 protocol it is necessary to register a new `nf_conntrack_l4proto` structure into the connection tracker. This new layer-4 shim6 tracker will maintain the state of the shim6 context which means that it tracks the four-way handshake of the context establishment.

Shim6 - the extension header One part of the shim6 solution will use the previously introduced extension header framework, to "preprocess" the header, before handling the underlying protocols. The first call to the shim6 handler will go to `get_l4proto`. In the case of a shim6 control message, the returned layer-4 protocol will be shim6 (as described earlier in this section). But in the case of a shim6 payload message it will be the underlying protocol carried by shim6 (TCP, UDP, . . .).
The next call will be to `pkt_to_tuple`. In this case, the handler will lookup for the context tag and replace the source and destination addresses of the connection tracking entry by the ULIDs that correspond to that shim6 context. That way the packet is associated to its original ULIDs. For an incoming shim6 payload message this change will enable the connection tracker to properly continue tracking the underlying transport connection (in the case of TCP, it can continue tracking properly its state) as it will see the packet associated to the original ULIDs.

With the above mentioned framework and shim6 trackers it is possible to correctly handle a shim6 connection between two hosts, including locator updates and changes. The next section will explain the implementation-specific issues and decisions taken to realize the previously presented design.

4.3 Implementation details

In this section, the details of the implementation will be explained. The different structures defined and the problems encountered to implement the goals specified in the previous section will be developed in details.

4.3.1 Extension header framework

As previously explained, the extension header framework is called at different places in the netfilter stack. For the implementation of the shim6 support into netfilter only two calls were needed. So the extension header framework is only called at the end of the call to `ipv6_get_l4_proto()` and `ipv6_pkt_to_tuple()` (see also Figure 4.3). But it would be easy to add other points in the IPv6 netfilter stack where calls to a new extension header handling function would have been added.

Listing 4.1 shows the structure used to register a new IPv6 extension header handler into the netfilter connection tracker. Two functions must be registered into this structure and they must obey to the specifications shown in the listing. The extension header framework iterates over the extension headers included inside the IPv6 packet. For each extension header it calls the corresponding handler function. If there is no corresponding handler function registered in the framework, the generic extension header handler is called.

Listing 4.1: Extension header handler structure for the support of another IPv6 extension header

```
struct nf_conntrack_exthdr
{
  /* Extension header protocol number */
  u_int8_t exthdrproto;

  /* Protocol name */
  const char *name;

  /*
   * Try to fill in the third arg: nhoff is offset of
   *                               current ipv6 exthdr.
   * Return true if possible.
   */
  int (*pkt_to_tuple)(const struct sk_buff *skb, unsigned int nhoff,
          struct nf_conntrack_tuple *tuple);

  /*
   * Called before tracking.
   *  nhoff: offset of current ipv6 extension header
   *  *dataoff: will be set after the call to give the length from
   *      nhoff to the the next exthdr. (if this extension
   *      header isn't the l4 protocol)
   *  *protonum: will be set after the call to be the next extension
   *       header(if this extension header isn't the l4
   *       protocol)
   * Returns 0 if the extension header will not act as a l4 protocol
   *      1 if it is the l4 protocol
```

```
 *      a negative value in case of error
 */
int (*get_l4proto)(const struct sk_buff *skb, unsigned int nhoff,
        unsigned int *dataoff, u_int8_t *protonum);
};
```

To add a new extension header handler, `nf_conntrack_exthdr_register(...)` must be called with the `nf_conntrack_exthdr` structure as an argument.
When a packet enters the netfilter stack, the IPv6 handler will make a call to `exthdr_pkt_to_tuple()`. This function iterates over the different extension headers, calls their appropriate `pkt_to_tuple()` functions, and determines with their respective `get_l4proto()` functions if it must continue its traversal through the extension header stack of the packet.

To treat the concurrent access to the extension header framework, a RCU read/write lock is used [McKe 06]. The RCU (Read-copy update) supports concurrency between a single updater and multiple readers. This goal is achieved by separating the update to a removal and a reclamation phase. The RCU update sequence is the following:

1. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it. (current users still see the old data structure)
2. Wait for all previous readers to complete their RCU read-side critical sections.
3. Free the old data structure (no other readers are accessing this data structure)

The original IPv6 connection tracker made inside the call to `ipv6_get_l4proto()` several operations to determine the layer-4 protocol. Namely for the authentication extension header. As specified in RFC2402 ([Kent 98], section 2.2) the length of the authentication header is different from the standard IPv6 extension header specification. So, a specific extension header handler for the authentication header was needed. Those extension header handlers are in `net/ipv6/netfilter/nf_conntrack_exthdr_*.c`.

Shim6 extension header handler

In this part, the two shim6 functions for the extension header framework are explained. They are implemented in `nf_conntrack_exthdr_shim6.c`.

`shim6_get_l4proto()` examines the shim6 extension headers and looks if it is a payload or a control message. If it is a control message, the function returns `1`, to indicate that this extension header should be treated as a layer-4 protocol. The pointers to

the beginning of the extension header will not be modified (see argument `dataoff` from `get_l4proto` in Listing 4.1), as the layer-4 shim6 handler will need to access the shim6 extension header. If the message is a payload message, the function returns `0` and updates the pointers to the next headers using the function `ipv6_optlen()` specified in the Linux Kernel.

The task of `shim6_pkt_to_tuple()` is to replace the IPv6 addresses in the tuple by the correct ULIDs that are associated to that shim6 context. For doing so, the context tag is extracted from the shim6 extension header and based on the context tag the ULIDs are retrieved (the context tag to ULID mapping is performed using a hashtable which will be explained later).
As Section 3.2.1 described, in the case of a present ULID-option into the shim6 context establishment messages, the firewall should extract these ULIDs from the messages (assuming that the R1 and R2 messages also contain the ULID-option). But due to time-constraints this has not been implemented in the shim6 firewall.

4.3.2 Layer-4 shim6 handler

The layer-4 shim6 handler tracks the shim6 context establishment. This need to be tracked to correctly map the context tag included inside the shim6 extension header to the ULID that is associated to this state. As described earlier in this document, the shim6 state tracking begins with the I1 message.

To obtain a fast mapping between the context tag and the ULIDs, a hashtable is used which links the context tag to the netfilter state of the shim6 connection. This shim6 state of the netfilter connection tracker needs to maintain several informations. To protect himself against the previously described Denial-of-Service attack (see Section 3.1.3), it needs to track the nonces used by the host.

Listing 4.2: Shim6 state inside the connection tracker

```
struct nf_ct_shim6
{
  uint32_t nonce[2]; /* last nonce seen from tuplehash[1].src
                * to tuplehash[1].dst for nonce[0] and
                * other direction for nonce[1] */

  /* ct[0] = context tag for packets from tuplehash[1].src
   *       to tuplehash[1].dst
   * ct[1] = context tag for packets from tuplehash[0] */
  uint64_t ct[2];

  /* Current shim6 state of that connection */
```

```
  char state;
};
```

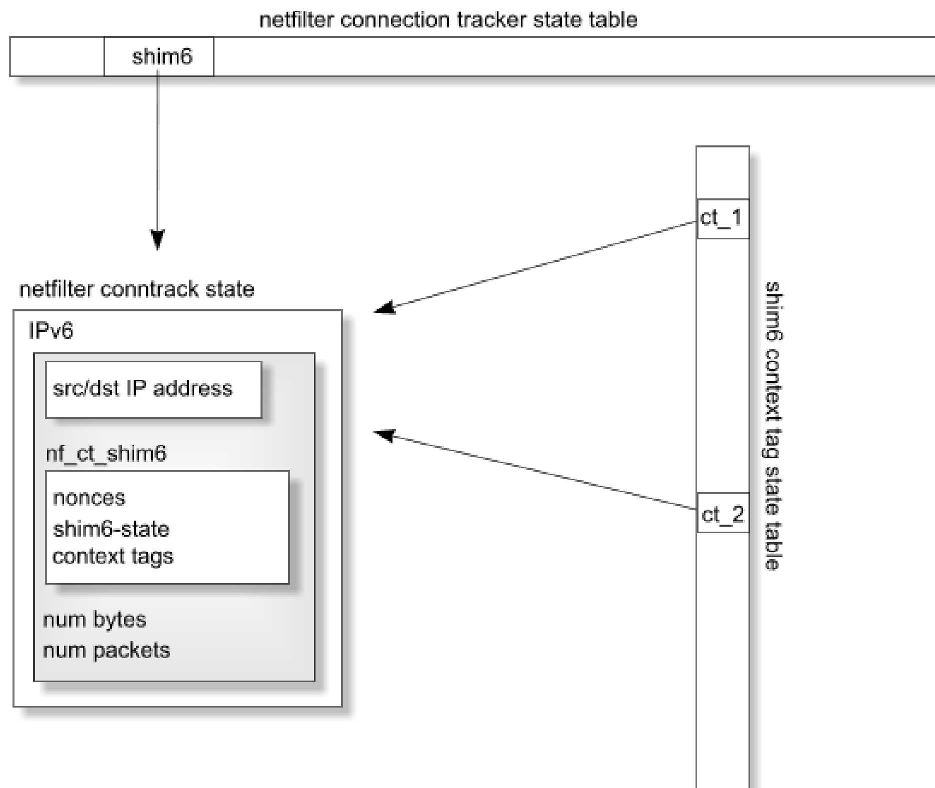


Figure 4.4: Simplified representation of the netfilter connection tracker maintaining a shim6 state

Figure 4.4 depicts the complete state maintained by the firewall to represent the shim6 state. The shim6 context tag table contains the context tags and a pointer to the existing netfilter state of the firewall. The details about this table will be explained later.
The netfilter connection tracker state table is also a hashtable. The keys for these entries are (as explained in Section 2.1.2) the IP addresses, protocol numbers and port numbers. For the case of the shim6 state, the port numbers are 0, as shim6 is only a layer-3 protocol. Due to the choice of having 0 as a port number, only one state per ULID-pair is possible. The shim6 protocol specification allows several shim6 states for the same ULID-pair by using the so-called Forked Instance Identifiers (FII). But the current implementation of LinShim6 does not supports the FII-option and so, the shim6-aware firewall does not support it either.

Context tags and nonces

Listing 4.2 shows the definition of the state maintained by the firewall. As the shim6 protocol uses two different context tags for each flow direction, these two need to be stored. The hosts choose the context tag that should figure in the shim6 payload messages for packets coming from the peer.

At the context creation the state has an initial direction. This direction is the one from the first shim6 message that the firewall has seen. `ct[0]` (resp. `ct[1]`) is the context tag that will be included in shim6 payload messages in the reply direction (resp. original direction). The reply direction is the one included in `tuplehash[1]` (see Listing 4.2). The same procedure is done to store the bidirectional nonces inside the state.

Hashtable

As mentioned earlier, a hashtable is used to map from context tags to the corresponding shim6 state. That way, on an incoming shim6 payload message, the retrieval of the corresponding ULIDs is very fast.

For the hashtable, the kernel implementation `hlist` in `linux/list.h` is used. In fact, the kernel hashtable is just a table of linked lists. The size of the table influences the efficiency of the hashtable. Access to the table is in $O(1)$, like any other c-access to elements of the table. The indexes of the table represent the hash of the key for every element in the table. In case of hash-collision, the elements are added at the beginning of the linked list, that is contained at each entry of the table (Figure 4.5 shows the representation of a kernel-hashtable). This way, the addition of elements in the hashtable has a complexity of $O(1)$. The retrieval of elements depends on the number of collisions for the specified key. In Worst case, the kernel will need to iterate over the whole linked list of a table-entry.

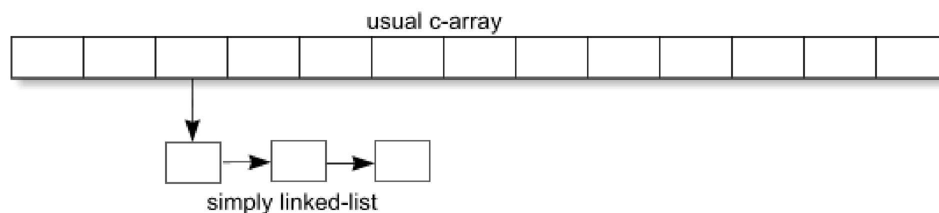


Figure 4.5: Kernel-hashtable which is a combination of a c-array of simply linked lists

The structure used for the shim6 hashtable is shown in Listing 4.3. The key for the hash-entry is the context tag. The object assigned to the hashtable-entry is the netfilter connection tracking state associated to that context tag. The hashtable has a specific size. It

was chosen to use half the size of the netfilter connection tracking hashtable. This choice is arbitrary and needs to be reevaluated with the further deployment of shim6 in an actual environment. In fact it highly depends on the number of shim6 sessions that will be established, in comparison with the other connections.
As there exists two context tags for every shim6 state (one for each direction), the hashtable contains two entries for each shim6 state. To map the context tag inside the hashtable, we just took the context tag modulo the size of the hashtable. This is valid, because the shim6 protocol specification defines that the context tag has to be a random 47-bit value [Nord 09].

Listing 4.3: Structure used for the mapping from context tag to shim6 state

```
struct nf_ct_shim6_hash
{
  struct hlist_node hnode;
  struct nf_conn *conn;
};
```

Timeout handling

Another difficult problem was to solve the handling of timeouts of the shim6 state inside the firewall.
In fact, after the end of the shim6 context establishment, the two hosts will not exchange any shim6 messages anymore, as long as there are packets exchanged between both hosts using the ULIDs as the locators in the packet (see also the Reachability Protocol [Arkk 06]). When there is only unidirectional traffic, the shim6 hosts will generate Keepalive massages.

To solve this problem, the firewall needs to monitor the traffic and look if there are any packets with IP addresses that are associated to a shim6 state. Thus, the firewall needs to do the following steps:

1. Extract the ULIDs associated to each packet passing the firewall.
2. Look in all the shim6 states if there is one with the same ULIDs.
3. Update the timer of this shim6 state.

These functionalities are placed in `ipv6_confirm(...)`. This function is called at the end of the packet's path through the networking stack, when the packet has passed the filter. This way, only packets that will be passed to the network will be accounted. The function `shim6_update(...)` is in charge to correctly update the associated shim6 state.
To efficiently retrieve the associated shim6 state for this pair of ULIDs, a special workaround is used. In fact, the packet is passed to the function `nf_ct_get_tuple(...)` which is meant to create the tuple based on the packet and

the layer-3 and layer-4 protocol handlers. But to avoid creating the real tuple (with layer-4 being TCP, UDP, ...), the function receives the layer-4 shim6 handler as an argument. So `nf_ct_get_tuple(...)` will create an IPv6-shim6 tuple, that will map to the corresponding shim6 state. Listing 4.4 shows the call to `nf_ct_get_tuple(...)`, where the shim6 protocol handler is used instead of the real layer-4 protocol included in the packet. If the original packet contains a shim6 payload extension header (and so the IP addresses are locators and not the ULIDs), the tuple will map to the ULIDs, as the shim6 extension header handler will retrieve these based on the context tag. With this tuple, the `shim6_update(...)` function can lookup for the corresponding shim6 state in the netfilter connection tracking table. To do so it will compute the hash of the previously generated tuple and lookup inside the linked-list of the hashtable corresponding to this hash. This may result in a complete lookup on all the states included inside this specific linked-list if no shim6 state is present for these ULIDs.
After retrieving the corresponding shim6 state the connection tracker can correctly update the timer of the shim6 state and so prevent the timeout of the connection tracking state of the shim6 connection.

Listing 4.4: Creation of the shim6 tuple out of a generic packet

```
nf_ct_get_tuple(skb, skb_network_offset(skb), 0, PF_INET6,
                IPPROTO_SHIM6, &tuple, __nf_ct_l3proto_find(PF_INET6),
                __nf_ct_l4proto_find(PF_INET6, IPPROTO_SHIM6))
```

TCP-checksum

Another issue is the checksum computation inside the firewall. The upper layer protocol might include a checksum that is based on the ULIDs but not on the locators that the packet is carrying. Thus, the checksum included is false, because shim6 modified the IP addresses of the packet.
In fact, the Linux netfilter firewall can be enabled to verify the checksum of the layer-4 protocols. If the checksum of the packet is wrong, the packet will not get discarded but will be marked as `INVALID` and a counter for wrong checksum calculations will be incremented.

To correct this problem, the function `nf_ip6_checksum(...)` needs to be modified. This function is used inside the netfilter framework to compute the checksum of several parts of an IPv6 packet. The checksum calculation extracts the IPv6 addresses of the packet. So, in the case of a shim6 payload packet, the locators inside the IPv6 packet will result in a wrong checksum calculation. A good solution is to add the ULIDs inside the socket buffer and so modify `nf_ip6_checksum(...)` in a way that it will use the ULIDs associated to this shim6 payload packet. The function `shim6_set_ulids(...)` will examine the

packet for a possible shim6 extension header and will add the ULIDs inside the socket buffer. `nf_ip6_checksum(...)` will then use these ULIDs for its checksum computation (see also Listing 4.5)
The choice of placing the ULIDs inside the socket buffer can be useful for supporting other protocols which also use a separation of locators and ULIDs. The next part will give a proposal on further work for more properly handling the checksum and timeout issues of those kinds of protocols.

Listing 4.5: Extract of nf_ip6_checksum(...) using the ULIDs inside the socket buffer

```
  struct in6_addr *src, *dst;
  __sum16 csum = 0;

#if defined(CONFIG_NF_CONNTRACK_SHIM6)
  if (shim6_set_ulids(skb)) {
    src = (struct in6_addr *) &skb->ulid_src;
    dst = (struct in6_addr *) &skb->ulid_dst;
  } else {
    src = &ipv6_hdr(skb)->saddr;
    dst = &ipv6_hdr(skb)->daddr;
  }
#else
  src = &ipv6_hdr(skb)->saddr;
  dst = &ipv6_hdr(skb)->daddr;
#endif
```

Further work

The current implementation for the handling of the timeout and the checksum calculation is not generic. Inside the generic netfilter code were placed shim6-specific functions like `shim6_update(...)` and `shim6_set_ulids(...)`.

A better design would be to add a *"Locator-ULID separator module"* into netfilter, that would call a generic framework which iterates over the possible multihoming handlers. This framework needs to be placed at different points in the netfilter code. These are the ones described in the previous two sections (`ipv6_confirm(...)` and `nf_ip6_checksum(...)`). The different Locator-ULID separator handlers would need to register into this framework and thus get called at these two functions of the netfilter stack.

Figure 4.6 shows the Locator/ULID separator framework. This framework should be compiled as a module into the Linux Kernel. Multihoming protocols that need to handle timeout and checksum issues should register into this framework and use the ULID-field in the socket buffer.

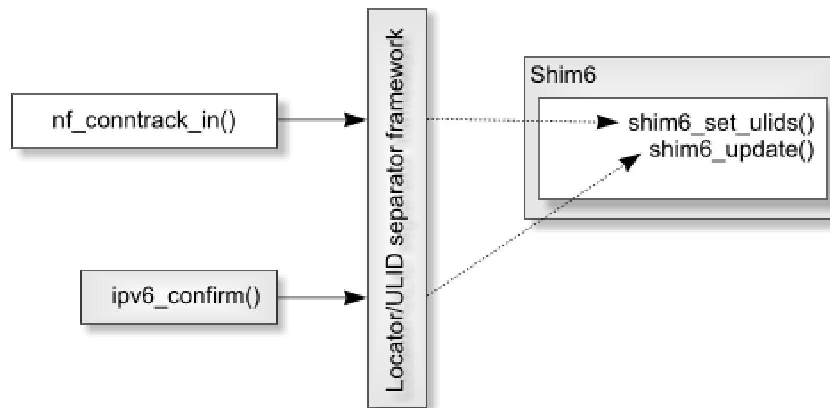


Figure 4.6: Locator/ULID separator framework

4.4 Conclusion

This chapter presented the design and implementation of the shim6-firewall. It has been designed to be as generic as possible by implementing the extension header framework. That way the netfilter connection tracker has a proper way to handle the IPv6 extension headers. The code can be found on the CD-Rom that is provided with this document. Appendix A presents a short overview of the different patches that need to be applied to the Linux Kernel 2.6.24. Some contributions have been done to related projects that are not directly related to the shim6-firewall but rather bug fixes in these projects (Appendix B)

However some parts of the shim6 specification has not been implemented. This is mainly due to time constraints, but also because the LinShim6 implementation neither supports the full protocol (FII-support is not implemented [Barr 08b]). So there was no way to test this shim6-feature. The parts of the shim6 protocol specification that are not supported are:

ULID-option As explained in Section 3.2.1, the shim6 protocol needs to be changed so that the firewall may support the ULID-option.

FII As described in Section 2.2.1, the FII allows to create several shim6 sessions for the same pair of ULIDs. But as the ULID-option, the FII is neither included in the R1 and R2 messages and so the firewall cannot track it.

Context recovery This part of the shim6 specification enables hosts to recover a lost context. It is only partially supported in the netfilter implementation. Actually, the context recovery is possible provided that the firewall did not lose the shim6 state. Then the firewall simply monitors the passing R1bis, I2bis and R2 messages. The case where the firewall loses the state was discussed much more in details in Section 3.2.2

Context collision The support to handle context collision (shim6 states with the same context tag) neither got implemented. For doing so, the firewall needs to monitor the locators. The problems related to this were discussed in Section 3.2.3.

Some futher work might be done to handle the timeout of the shim6 states (`shim6_update`) and the checksum computation in a generic manner rather than a simple call to the shim6-specific functions.

The next chapter will present an evaluation of the shim6-firewall by applying some performance evaluations on the Kernel.

Chapter 5

Evaluation of the shim6-firewall in the Linux Kernel

This chapter presents the evaluation of our implementation. The shim6-firewall is tested when facing a huge number of state creations. The time that these packets stay in the network stack of the firewall is measured in comparision to the number of states created.

5.1 Setup

As depicted in Figure 5.1, the test-environment is composed of 3 machines, interconnected to each other by a 100Mbps Ethernet Switch. The performances of S and R are of little importance. However, the firewall is running on an Intel Pentium 4 processor with 1400MHz, it has 376 MB of RAM and a 100 Mbps Ethernet card.

The network is statically configured in such a way that packets sent by S will go through the firewall, which forwards these to R. When R receives the packets it will just drop these. The purpose of R is just to have a forwarding station for the firewall. Thus, the packetflow is unidirectional, which means that for every simulation, the I1, R1, I2 and R2 packets are originated from S. This host spoofs packets so that the firewall sees requests and responses with the right IP addresses.

5.2 Test description

The firewall will be tested on the delay it introduces when handling the packets. For doing so a tcpdump will be executed on the firewall. Tcpdump monitors the packets on the network interface and logs these. With the logging, tcpdump adds a timestamp to the log so the delay between the incoming and outgoing packet can be computed. That way the

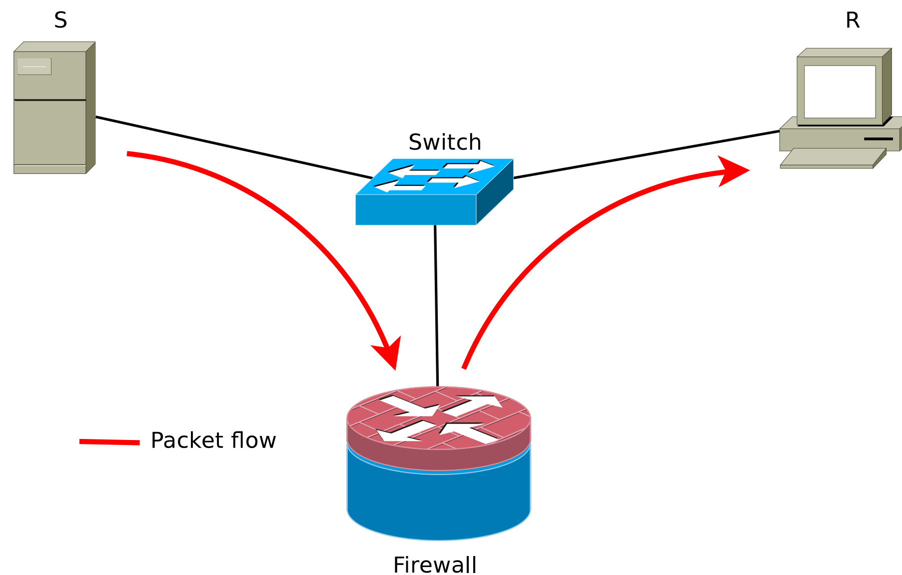


Figure 5.1: Setup of the performance-evaluation test-lab

firewall is evaluated like a "blackbox" as our only interest is the delay that the connection tracking introduces.

The firewall does not contain any filtering rules. Its only purpose is to handle the connection tracking. Packets that go through the firewall will either create a new state or will be associated to an existing state. This introduces the delay in which we are interested. To evaluate the performance we created a large number of states on the firewall and observe how this influences the delay introduced to the packets.
The default Linux firewall limits the number of possible states during boot-time in function of the memory present on the machine. So this value has been increased manually after the boot to allow the creation of up to 300000 states.

Three different evaluations were done on the firewall and are explained in the next sections.

State creation The firewall will be tested for the case when a large number of states will be created. Every packet that is part of a state creation (for shim6: I1, R1, ...) will be logged. So we can observe if the delay of these packets varies with the number of states in the firewall.

Payload messages Another test for the shim6 payload messages. As the previous test, these messages will be evaluated in relation to the total number of shim6 states present in the firewall.

Firewall stressing A good test for deadlock problems is to try to stress the firewall by sending fast subsequent messages. This was done in the final test.

To send the shim6 messages, the so-called shimulator was used [Mekk 07]. The shimulator allows to send shim6 messages that have spoofed IP addresses. It is possible to specify the context tags and nonces included in the shim6 messages.

5.3 State creation

In the first performance tests, a large number of states are created on the firewall to analyze the delay of these state creation packets as the firewall's state table grows. For these tests, the machine S (Figure 5.1) sends the packets necessary to create the state on the firewall. S is sending the packets with an interval of 1ms between each packet so that the firewall has enough time to properly parse each packet. To create a large number of states, S sends these with originating source address `4242::4242` to the destination starting at `1111::1111` upwards. S also needs to generate the replies to its connection establishment messages by spoofing the addresses. As the destination IP address is increasing, different states are created on the firewall. To prevent the firewall from deleting these states, the timeout configuration of the states needs to be increased.

To compare, three tests are done:

TCP-state creation on a clean Kernel As a reference, TCP-state creation is evaluated on a clean Linux Kernel. The setup consists of a Linux Kernel 2.6.24 with a similar Kernel configuration than the shim6-firewall. Machine S will send the necessary messages to create a TCP connection (namely Syn, Syn/Ack and Ack). As described earlier, the delay of each of these messages will be monitored while the total number of TCP-states is increasing on the firewall.

TCP-state creation on the shim6-firewall The shim6-firewall will be flooded by the TCP connection establishment messages on the shim6 firewall. No other states than these TCP states are on the firewall.

Shim6-state creation The shim6-firewall is flooded by shim6 session establishment messages. The firewall will have more and more shim6 states in its state table.

As described in Section 5.2 tcpdump is used to obtain the time that the packets take to go through the firewall. Tcpdump is launched with the command `tcpdump ip6 proto 61 -i eth0 -w tcpdump.log`. The parameter `ip6 proto 61` ensures that all the packets that have `61` as the next header after IPv6 will be logged. Thus, tcpdump monitors only the shim6 messages. To log on TCP packets, `61` needs to be replaced by `6`.

5.3.1 Comparing the initiation messages

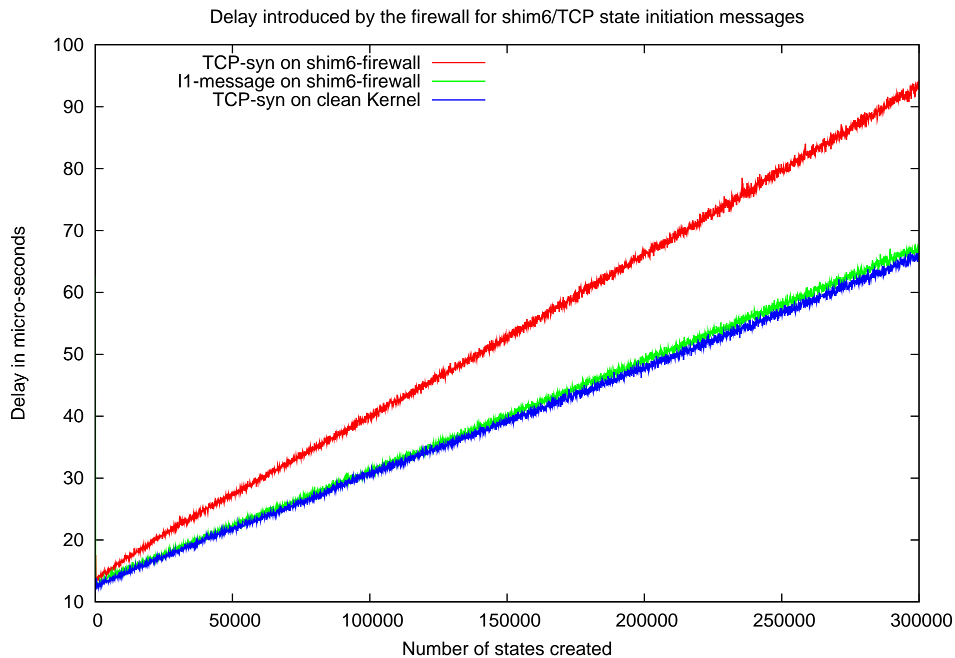


Figure 5.2: Shim6-TCP state creation

Figure 5.2 shows the delay in μs of the initial state creation messages for TCP and shim6 (namely TCP-syn and the I1 message). To improve the readability, these delays are an average of 100 values. It is important to note, that the standard deviation is of no concern. For a small number of states it is around 2-4μs and for about 300000 states it increases to 5-7μs. This shows that the hashtables are equally filled (equally distributed collisions) and thus the lookup is nearly constant for a given number of states.
The TCP-syn test was executed on the two different Kernels. In red is shown the TCP state creation on the shim6-firewall and on blue for the clean Kernel.

The first observation is that the TCP state creation on the clean Kernel is slightly better than the shim6-state creation on the shim6-firewall. This is due to the overhead of the extension header framework of the shim6-firewall.

We can observe that the delay of TCP-syn packets on the shim6-firewall increases faster than on the clean Kernel and than the I1-messages. This is because the shim6-firewall must update the shim6-states that are associated to a specific pair of ULIDs (see

Section 4.3.2, Timeout handling). When calling `shim6_update(...)` the firewall will iterate over the entries in the netfilter state table to find a shim6-state that matches the ULIDs of the TCP-syn packet. As the netfilter state table is a hashtable, it consists of a c-array of simple linked lists (see Section 4.3.2, hashtable). So `shim6_update` iterates over the linked list in the c-array that corresponds to the hash it computed. As the firewall does not contain any shim6-state during this test, the iteration continues until it reaches the end of the list. Thus, this scenario is a worst-case test.
Due to this results, it might be worth discussing, whether the choice of iterating through the list of netfilter states rather than iterating through the list of context tags was a good choice. This depends highly on the number of shim6-states that are present. If the firewall iterates through the list of context tags, it needs to examine **every** context tag, because this one is not known when the firewall receives a TCP packet. For every context tag it finds in the hashtable it needs to retrieve the associated netfilter state and then compare the IP addresses. When the firewall does not contain a lot of shim6-states this solution will be faster than the implemented one. So, this needs to be further examined, based on information of the number of shim6-states that are present on real firewalls that handle shim6 packets.
It might also be possible that the firewall would dynamically change its lookup procedure based on the number of shim6-states in comparison to the total number of states in the firewall. That way the firewall could optimise its lookup procedure and always take the best decision.

We also have to mention that the I1-message on the shim6-firewall Kernel and the TCP-syn on a clean Kernel have the same slope. This shows that the shim6 state creation on the shim6-firewall is comparable to a state-creation on a clean Kernel. The extension header framework and the shim6 protocol handler do not add a considerable delay to the firewall. Upon the reception of an I1-message the firewall does a similar job as a clean Kernel that receives a TCP-syn packet.

5.3.2 Comparing subsequent messages

It is also interesting to analyze the delay introduced by the firewall after the initial state-creation messages. Especially the I2 and R2 messages are of interest, because they will insert the context tag into the hashtable described in the previous chapter. It can be expected that this insertion should be rather fast, because the hashtable implementation in the Linux Kernel uses a linked list, and simply adds the element at the top of this list.
This test was done in the same manner than the previous state-creation comparision. Based on the previous state-creating messages, the subsequent messages were send and their delay measured by tcpdump. Like the previous test, the values shown in Figure 5.3 are an average of 100 values. The standard deviation is comparable to the one from the previous test.

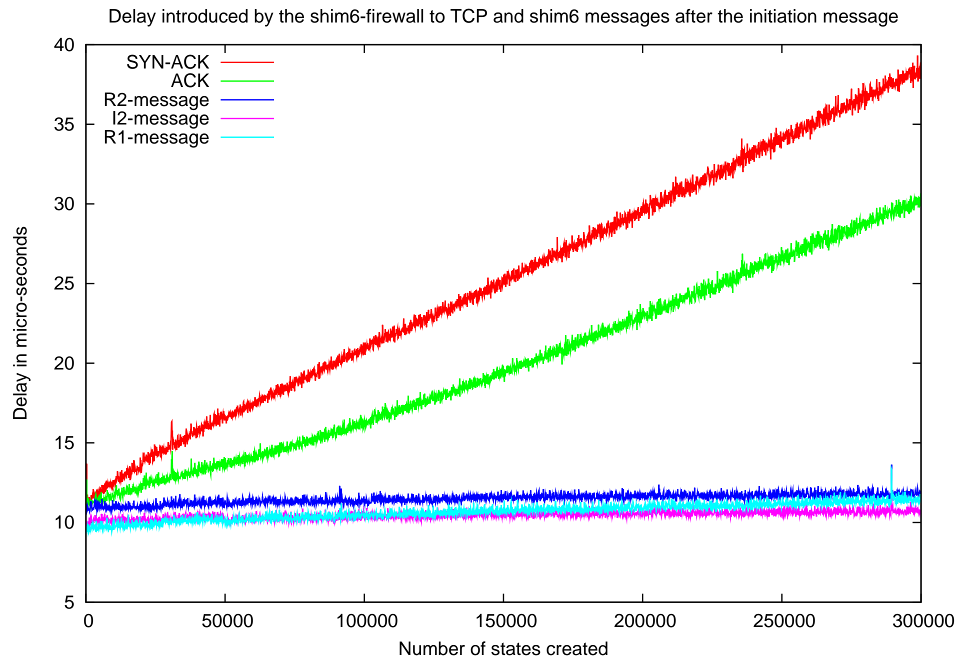


Figure 5.3: Shim6-TCP after the state creation

As one can see in Figure 5.3, the shim6 messages are performing very well. The delay remains constant in comparison to the number of states created in the firewall. This can be explained by the fact that the firewall stores the shim6-state at the top of the list corresponding to the hash of this packet (see also Section 4.3.2). This way, the corresponding shim6-state can be retrieved in a constant time.

The reason why the processing time of the TCP-syn-ack and TCP-ack messages increases with the number of states in the firewall is due to the same reason as the one from the previous test. The shim6-firewall needs to look for each packet that enters the firewall and that is **not** a shim6-control message if there is a corresponding shim6-state that must be updated to avoid the timeout of this state. The corresponding shim6-state will be retrieved based on the IP addresses of the packet and thus, the shim6-firewall will iterate over more and more states (inside the collision table of the netfilter connection tracking state hashtable), as the number of firewall-states increases. This is of course a worst-case scenario because in the case of TCP-state flooding there are no shim6 states present in the firewall. As in the previous test this will perform much better when there are shim6 states present in the firewall.

The delay of the shim6 messages remains constant. This is the expected behaviour, because shim6 messages do not trigger a lookup for related shim6 states.

5.4 Sending shim6 payload messages

One of the most interesting performance evaluation is to see how shim6 payload messages are delayed in comparison to normal packets. It can be expected that these messages will be slowed down considerably, because the shim6 extension header handler needs to retrieve the associated ULIDs of this payload packet. For this, it needs to look in the hashtable maintained by the shim6 module. With a large number of shim6 states present in the firewall this may considerably increase the delay of the packet.
However, payload packets without shim6 payload extension header are not unaffected by the shim6 module, due to the lookup on related shim6 states for maintaining the timeout of these states (see also Section 5.3).

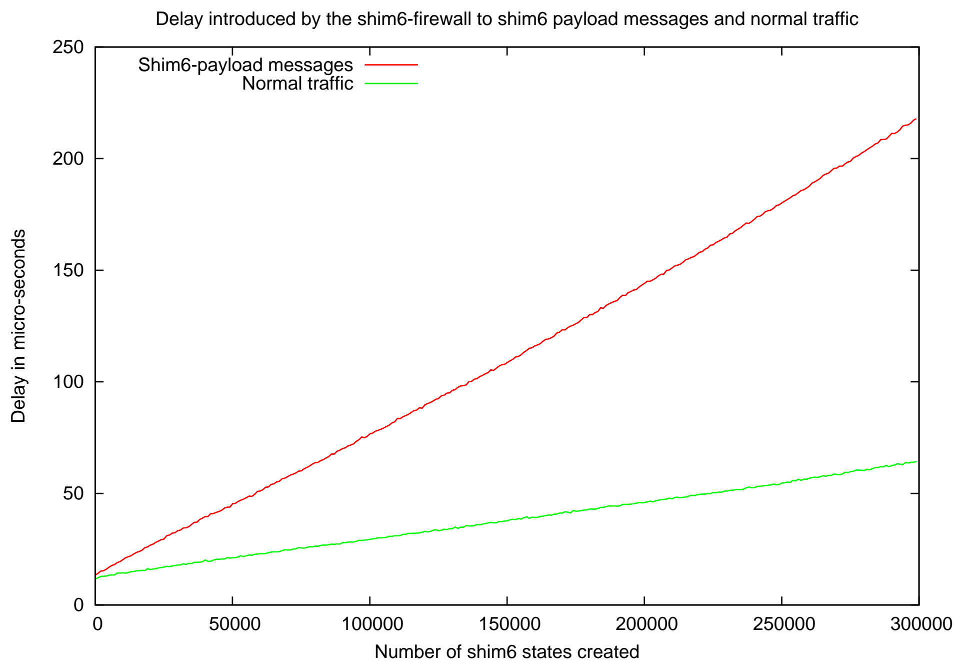


Figure 5.4: Shim6 payload messages vs. normal traffic

The packets sended were simple IP packets that did not contain a TCP-header. This was done due to the lack of the shimulator to generate TCP packets with a shim6 payload header. So the Linux Kernel firewall will create a minimal state for this IP packet. The timer of this state got increased to avoid the timeout of these states. Figure 5.4 shows the results for these performance tests. It shows the comparison between packets containing a shim6 payload header and those that do not contain it. During this test, shim6 states where created and every 1000 states, a burst of 100 payload messages were send. It is important to note that for the shim6 payload messages we chose the 100 first context tags of the shim6 states. That way, the firewall needs to look the furthest in the collision table of the hashtable containing the context tags. So, this performance test can be seen as a worst-case scenario.

We can observe a significant performance breakdown in Figure 5.4. This is the expected behaviour of the shim6-firewall. The firewall needs to lookup in the hashtable containing the context tags for a shim6 state that matches the context tag in the shim6 payload extension header. As the number of shim6 states increases, the collisions in this hashtable increase and the lookup takes more and more time.
The reason why the payload messages without shim6 extension header also rise with the number of states created was explained in the previous tests due to the lookup on matching shim6 states whose timeout needs to be updated. As the payload messages that are sent contain the IP addresses of the first 100 shim6 states it is clear that the firewall needs to lookup the whole collision table of the netfilter state table.

As the performance penalty is of about 150 μs (compared to normal traffic), one could consider to increase the size of the shim6 hashtable to reduce the number of collisions. This needs to be further evaluated for real deployments in function of the number of shim6 states that a usual firewall needs to manage. A total number of 300000 shim6 states might be too high for a real case scenario. In the beginning of the shim6 deployment there may be a much smaller number of shim6 states as there are not so many hosts supporting the shim6 protocol.

5.5 Firewall stressing

Another interesting performance evaluation is to stress the firewall by flooding it with shim6-state initiation messages. The test was performed by flooding the firewall with the shim6 session initiation messages by using the shim6 message generator shimulator. Unfortunately, the shimulator does not allow to send messages at a very high rate. Even after some optimisations to the source code of the shimulator, the sending of a single shim6 messages still takes between 70 and 100 μs. It is clear that with such a delay to send an I1 message of 56 bytes (40 bytes of IPv6 header and 16 bytes for the minimal I1 message)

it is impossible to increase the rate of shim6 messages up to the linespeed of the setup (100Mbps).

Nevertheless the firewall performed very well with the flow that was generated. At about 4-5 Mbps, the firewall managed to parse the packets without huge problems. The firewall managed to treat all the packets without generating a Kernel Panic, which may be possible due to bad semaphore handling inside the shim6 specific code of the firewall. Of 2473914 packets received by tcpdump there were only 6064 packets dropped by the Kernel, which means that only 0.2% are dropped by the Kernel. The packets dropped are mainly due to the forwarding buffer of the kernel which can be overloaded.
Other drops are also possible due to the firewall, if the I1 message of a particular ULID pair is lost, the succeeding messages corresponding to these ULIDs will also be discarded because the shim6-firewall does not allow state creation starting with a message different than the shim6-I1 message.

5.6 Conclusion

The shim6-firewall performs very well in comparision to the standard traffic. However some problems are still present. In particular the lookup over the shim6 states to update the timeout of the shim6 state could be optimized. The best solution would be to lookup in a table in such a way that it is optimized in comparision to the number of shim6 states present in the firewall. In a first usecase, where it is expected that only a small part of the firewall states are in fact shim6 states it would be better to lookup in the context table rather than in the netfilter state table.
The delay introduced for the shim6 payload messages is also to be considered. A simple increase of the size of the hashtable might resolve this issue. But this also needs to be further evaluated as shim6 is being deployed in the real world.

Chapter 6

Configuring a shim6-firewall

This chapter presents recommendations for firewall administrators configuring the shim6-firewall. If the firewall protects a multihomed site with shim6-hosts the configuration needs to be adapted to avoid breaking the security policy of the site. To allow iptables to express these policies it is needed to modify iptables. The new iptables modules are also described in this chapter.

As throughout this whole document, we assume that the firewall sees all the traffic passing. The case of distributed firewalls is not treated in this thesis.

6.1 Firewall configuration for multihomed hosts using shim6

It is a challenge for the firewall administrator to properly configure his firewall so that shim6 is allowed[1]. The configuration needs to be consistent so that no new security holes appear. This section will explain the different problems that need to be considered.

Figure 6.1 illustrates an example of a multihomed site. This site consists of three different parts:

DMZ A Http-server needs to be accessible from the Internet. Incoming connections to port 80 need to be allowed.

Workstations These workstations are allowed to access the Internet. However, no incoming connections from the Internet are allowed to these machines.

Internal machines These machines have only access to workstations and servers inside the site. No outgoing or incoming Internet access from or to these machines is allowed.

As the site is multihomed, it possesses several IPv6 address prefixes (namely P-1 and P-2). We assume that the machines were configured so that every machine being inside one

[1] as is also explained in the case of MIPv6 by S. Krishnan [Kris 08]

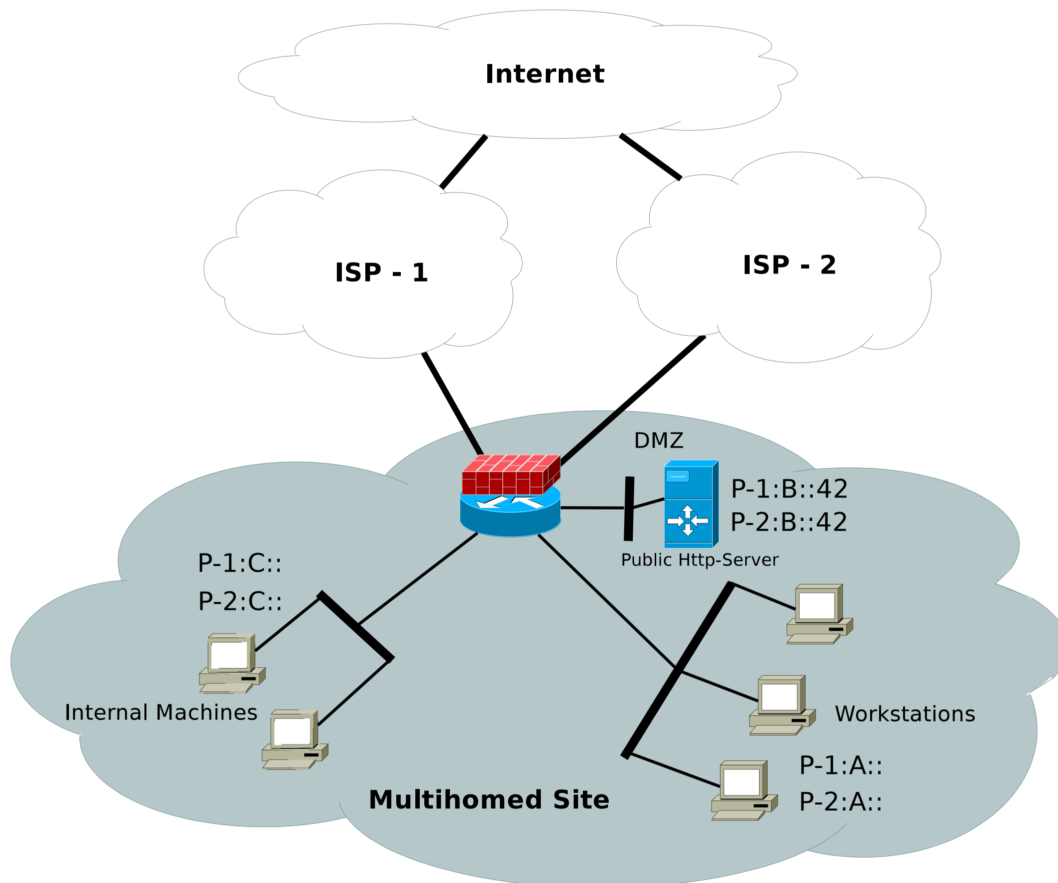


Figure 6.1: Example topology for a multihomed site

of the three previously mentioned parts have a common subnet ID in their IP address. This means that the hosts have the Provider-Dependent prefixes and each part of the site has its own subnet ID (A, B and C in Figure 6.1).
The rules in a configuration without shim6 are as described in Table 6.1. As the site is multihomed, rules need to be expressed for every provider prefix.

6.1.1 How to allow shim6 traffic

In the case where the previous configuration enables shim6 on all the machines, rules have to be added to allow the shim6 messages.

For the Http-Server, incoming connections to port 80 are allowed (see Table 6.1). If this server has shim6 enabled, we want him to be able to initiate a shim6 session. The shim6 session may be initiated by either the server or the client. Thus, we need to allow the shim6 control messages. The shim6 payload packets will be allowed by the rules allowing

Source	Destination	Src-Port	Dst-Port	Protocol	State	Target
Internet	P-1:B::42	any	80	TCP	any	ACCEPT
P-1::B::42	Internet	80	any	TCP	ESTABLISHED	ACCEPT
Internet	P-2:B::42	any	80	TCP	any	ACCEPT
P-2::B::42	Internet	80	any	TCP	ESTABLISHED	ACCEPT
Internet	P-1:A::	any	any	any	ESTABLISHED	ACCEPT
P-1:A::	Internet	any	any	any	any	ACCEPT
Internet	P-2:A::	any	any	any	ESTABLISHED	ACCEPT
P-2:A::	Internet	any	any	any	any	ACCEPT
any	any	any	any	any	any	DROP

Table 6.1: Firewall policy for the multihomed site without shim6

the TCP-traffic. When the firewall sees a shim6 payload message it will filter on the upper layer protocol included in this payload message. If the configuration would allow shim6 payload messages it will create a security hole as any traffic can be included inside a shim6 payload message.

Source	Destination	Src-Port	Dst-Port	Protocol	State	Target
Internet	P-1:B::	any	80	TCP	any	ACCEPT
P-1:B::	Internet	80	any	TCP	ESTABLISHED	ACCEPT
Internet	P-2:B::	any	80	TCP	any	ACCEPT
P-2:B::	Internet	80	any	TCP	ESTABLISHED	ACCEPT
P-1:B::	Internet	—	—	shim6-ctrl	any	ACCEPT
Internet	P-1:B::	—	—	shim6-ctrl	any	ACCEPT
P-2:B::	Internet	—	—	shim6-ctrl	any	ACCEPT
Internet	P-2:B::	—	—	shim6-ctrl	any	ACCEPT

Table 6.2: Firewall configuration of the Http-Server allowing incoming and outgoing shim6 state creations

The workstations have unlimited access to the Internet. However incoming connection requests are not allowed. We want to express the same policy for the shim6 connections. Outgoing shim6 session initiations are allowed. However incoming requests are not allowed to avoid host scanning of the site. The firewall administrator simply needs to allow any outgoing shim6 control messages and block incoming `NEW` shim6 connections. As the rules shown in Table 6.1 already allow outgoing connection requests and block the incoming connections (by only allowing established connections) no rules needs to be added to the ruleset. Incoming shim6 connection requests will be blocked by the firewall.

Filtering in the firewalls is usually done on the IP addresses included inside the packets. However with shim6, these IP addresses are the locators. The ULIDs are included in the state of the connection. The implemented shim6-firewall manages to associate a packet to its corresponding state. It is important to discuss, if the filtering should be done on the locators or on the ULIDs. This depends on the policy that needs to be respected by the firewall. The two next subsections describe scenarios where it should be better to filter on ULIDs rather than on the locators.

6.1.2 Filtering on a shim6 flow

There are several filtering rules that require the firewall to look inside the state representation associated to the packet. As described in Chapter 3, the shim6-firewall does not maintain a locator specific state, but rather groups the state per ULIDs. This may be a problem for the state specific rules such as byte- or packet-limiting rules per connection.

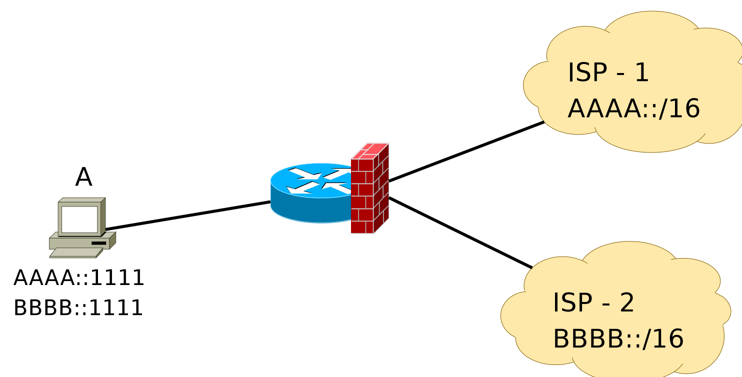


Figure 6.2: A multihomed host connected to two ISPs

On Figure 6.2 we can see the host A, which is connected to two ISPs. The firewall administrator would like to express the rule that no connection over ISP-2 would send more than 5 Mb.

Host A is connected to another host B on the Internet over ISP-1. Some traffic has been carried on this connection (about 20Mb). When shim6 switches the locators to use `BBBB::1111`, the state machine of the firewall will associate this flow of traffic to the original state, identified by the ULIDs. This state has already 20Mb of traffic carried, and thus the firewall would block packets passing by ISP-2, even if no 5Mb of traffic got carried by ISP-2.
The problem is that the same state is maintained for the different locators. The state is

viewed as being always linked to the ULIDs. If the administrator wants to filter on the connection state based on locators, he cannot do that.

This problem is not only specific to shim6, but due to every multihoming protocol which uses different locators to carry traffic for a specific flow (identified by its ULIDs). The information on the bytes or packets exchanged for this connection will not be specific to the locators, but to the ULIDs of this connection. Firewall administrators should avoid using this kind of rules in a multihomed environment. However if these rules are required they should be expressed on ULIDs.

6.1.3 Firewall blocking the new locator pairs

Another scenario that needs to be further discussed is that the firewall might block the shim6 flow of packets, even if the locator switch succeeded.

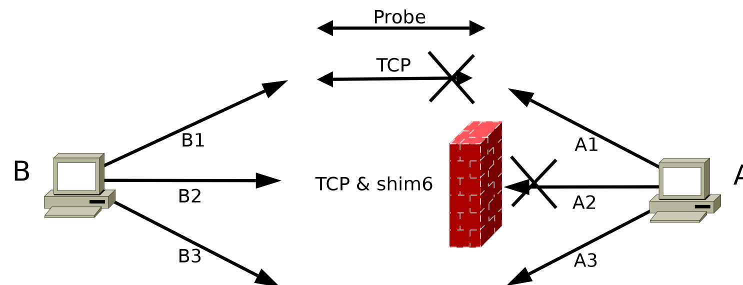


Figure 6.3: Firewall blocking the packet flow after a successfull locator switch

In Figure 6.3 we can see the scenario where the firewall will block the TCP flow after a successful locator switch. Host A and B are multihomed and have both three IP addresses. They have successfully established a TCP and a shim6 connection over the ULIDs A2 and B2. At a certain moment in time, shim6 recognizes the need to switch locators (due to a linkfailure). Shim6 will switch the locators to A1 and B1, using the reachability protocol with its probe messages. The firewall is configured in a way that it will allow the reachability protocol over these locators (allowing shim6 control messages). But, the firewall ruleset does not allow any TCP packet with the locators A1 and B1. Thus, the TCP flow with the new locators will be blocked. Shim6 will recognize that no packets are flowing and will so try to *“ping”* the correspondent host with keepalive messages. These will, once more, correctly pass the firewall.
Thus, shim6 will not switch the locators again, because the keepalive messages are correctly responded by the host. The TCP connection will timeout, even if there is another working locator pair.

One might think that it may be possible for the firewall to block the probe messages when hosts launch the reachability protocol as the firewall is the only entity on the path responsible for the timeout of the TCP connection.
So, it would be necessary for the firewall to check its ruleset for any rule that would block the new payload flow that would pass by these locators. The problem about this is that the firewall cannot correctly check the ruleset because it cannot be sure of the flow that would pass by these locators. If this is the first locator switch, the firewall has not seen any shim6 payload packets and so cannot associate the context tag to a specific upper layer protocol flow.
But even if there were already some shim6 payload packets that passed the firewall, the firewall cannot correctly predict whether the flow will pass the new locators. This is due to the fact that most firewalls allow to filter on packets information that can change "on the fly", e.g. the packetsize.

Thus, it is recommended that the firewall administrators should anticipate the case of a locator switch and thus should set up their ruleset in a way that would allow a flow to pass the firewall even after the locator switch. If a specific traffic class needs to be blocked (e.g., UDP) for an ISP, the rules should also block shim6 control messages. With this REAP will not be able to switch the state to this pair of locators. However when using this, no shim6 at all is able to use these locators.
In general it is no more possible for a firewall to block a specific traffic for an ISP while still allowing shim6 to pass by this ISP. As in the previous section, firewall administrators should express their rules based on ULIDs rather than on locators.

6.1.4 Conclusion

This section described the firewall configuration for a site where shim6-enabled hosts are present. The administrator should be careful when expressing the rules. The rules should be expressed on the ULIDs and not on the locators. With shim6 a flow is identified by its ULIDs an no more by its locators. Thus the firewall should follow this design.

6.2 Adoptions to the iptables implementation

For being able to correctly apply the previous mentioned policies, iptables needed to be adapted. For the filtering in a multihomed environment with shim6, two modules were added to iptables. This section will briefly present their role and usage. It will not go into the

details of their implementations as the implementation is highly based on already existing iptables-modules.

6.2.1 IPv6 extension header match

The default ip6tables implementation already presents a match on the IPv6 extension headers included in the packet. Of course this module does not support the shim6 protocol. Support for the shim6 extension header has been added, so that the module can match on an included shim6 extension header inside the packet.

To use the module, rules need to be expressed like the following:

```
ip6tables -t filter -A FORWARD
          -m ipv6header --soft --header shim6,route
```

The module now supports the shim6 argument. The option `--soft` means that the rule will match on any packet that contains the shim6 header **or** the routing extension header. For a further description of the `ipv6header` module look at the iptables manpage [Kis 09].

6.2.2 shim6-specific module in ip6tables

The previously described adoption only enables iptables to match on shim6 extension header but not on the type of this one (I1, R1, . . .). Thus, a module has been implemented, dedicated to match on any type of shim6 message type. The usage of the shim6 module is the following:

```
ip6tables -t filter -A FORWARD
          -m shim6 [!] --type type[,type]
type = I1, R1, I2, R2, R1bis, I2bis, UPDreq, UPDack, Keepalive,
       Probe, Error, ctrl, pld
```

So the module is also able to match on either shim6 control messages (`type = ctrl`) or shim6 payload message (`type = pld`).
It is possible to specify several types by separating the list by commas. The type match can also be negated by prepending a `!` before the type command.

6.2.3 Matching on the ULIDs

In the default behaviour of iptables it matches on the IP addresses that are contained in the packet. In the default firewall, the IP addresses in the packet correspond to the IP

addresses in the netfilter state associated to this packet. But as the shim6-firewall will associate shim6 payload messages to their corresponding ULIDs, the addresses in the packet and the addresses in the state are not necessarily the same anymore.

So a module was needed to give the firewall administrator the ability to either match on the locators (included in the packet) by using the default way to specify the IP addresses, or to match on the ULIDSs that are associated to this packet flow.
As was decided in Chapter 3, the packets that go through the firewall will be associated to the state containing the correct ULIDs. So, the match for ULIDs just looks at the connection tracking state associated to this packet and matches on the IP addresses contained in that state. These IP addresses may be the same as in the packet or they may be different if a shim6 payload extension header is present.

To match on ULIDs, the administrator needs to use the following syntax:

```
ip6tables -t filter -A FORWARD
          -m ulid [!] --usource address[/mask]
                  [!] --udestination address[/mask]
```

The configuration does not need to specify source **and** destination addresses. If no source-address is specified, the rule will match on all packets matching the specified destination address.

6.3 Conclusion

The implemented modules in ip6tables enable the firewall administrator to express finer grained rules on his firewall regarding the fact that his site is multihomed. The administrator needs to consider the shim6 traffic and allow this one to pass the firewall.
However the admin needs to be aware that he does not has the same freedom to express all kind of rules that he wants. He needs to express consistent rules for all the possible IP addresses (Provider-dependent prefixes). In particular it is not possible to limit the number of exchanged bytes/packets per connection for a specific provider.

The firewall needs to filter on ULIDs and no more on locators. As this is the design of shim6, to identify a flow by its ULIDs and no more on its locators, the firewall rules should do the same.

Chapter 7

Conclusion

The goal of this thesis was to fully support the shim6-protocol in the Linux netfilter firewall. However during this thesis, we discovered that it is much more complicated to support such a rich protocol like shim6 in a firewall. Several problems were discovered for allowing a firewall to support the shim6 protocol, and multihoming in general.

The Master Thesis first introduced the concepts that appeared in this document. Firewalls have been described. Especially the purpose of a stateful firewall has been explained. The host-based multihoming protocol (shim6) has been explained in details.

With these basics, the problems that shim6 poses to a stateful firewall could be analyzed. A solution to track the shim6 context establishment has been designed. However, the design does not support the full shim6 protocol (as does neither the official shim6 implementation, LinShim6). Several changes should be done to the shim6 protocol to allow firewalls to fully support shim6.

The support of shim6 has been implemented in the Linux netfilter firewall. For doing so, several design possibilities have been taken into consideration. Finally, it was chosen to implement an IPv6 extension header handler in the connection tracker. With this one it was possible to implement the shim6-firewall so that it fits nicely in the existing netfilter framework.

As one of the objectives of this thesis was to provide an efficient support for shim6 in the Linux firewalls, this needed to be evaluated. The shim6-firewall performs very well, even when handling a large amount of states. Some performance optimisations are still possible when it becomes known, how many shim6 states a firewall will need to handle.

Firewall administrators need to explicitly allow the shim6 traffic, while watching out that they do not open any security holes for their site. This goal can be achieved by using the newly implemented shim6-module in iptables.
The ulid-module allows the administrator to configure rules that match on the IP addresses

included in the connection state and not in the IP header. So, in the case of a shim6 payload message the rule will match on the ULIDs and not on the locators.

Further work

During this Master Thesis many problems where discoverd with a firewall facing a multihoming protocol. Some of them have been solved, while others could not be completely solved due to time constraints.

The firewall needs to support the ULID- and FII-option, assuming that the shim6 specification will include these options in the reply-messages.
Context recovery for the firewall needs also to be supported by allowing the firewall to trigger a context recovery on the hosts. For this, shim6 needs to allow the firewall to trigger this context recovery by sending a predefined message (as proposed in Section 3.2.2).
The firewall will also need to track the locators to address the issue of context confusion described in Section 3.2.3. For doing so, the shim6 specification needs to bind the number of possible locators to a reasonable upper limit.
With the deployment of the shim6 protocol the firewall will needs to be tweaked to optimize its performance based on the number of shim6 states that are present in the firewall.

It would also be nice to improve the design of the shim6 timeout and checksum correction (Section 4.3.2) so that it fits in a generic design as described in Section 4.3.2. That way multihoming protocols that separate the ULIDs from the locators could dynamically be included into this framework.

The issue of firewalls in a multihomed environment has not been extensively analyzed up to now (besides RFC 4487 on the Mobile IPv6 protocol, [Le 06]). This Master Thesis presents a detailed analysis of a multihoming protocol (shim6) and the problems related to stateful firewalls. There still needs to be done a lot of work in collaboration with the shim6 working group to adapt the firewalls, but also the shim6 protocol, for these two to work correctly together.

Bibliography

[Arkk 06] J. Arkko and I. Beijnum. “Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming”. Internet Draft (work in progress), IETF, 2006. draft-ietf-shim6-failure-detection-13.

[Aura 05] T. Aura. “Cryptographically Generated Addresses (CGA)”. RFC 3972, IETF, Mar. 2005.

[Ayus 06] P. N. Ayuso. “Netfilter’s connection tracking system”. `http://people.netfilter.org/pablo/docs/login.pdf`, June 2006.

[Bagn 08] M. Bagnulo. “Hash Based Addresses”. Internet Draft (work in progress), IETF, Jan. 2008. draft-ietf-shim6-hba-06.

[Barr 06] S. Barré. *Développement d’extensions au Kernel Linux pour supporter le multihoming IPv6*. Master’s thesis, Université Catholique de Louvain (Belgium), June 2006.

[Barr 08a] S. Barré. “LinShim6 - Implementation of the Shim6 protocol”. Tech. Rep., Université catholique de Louvain, Feb 2008.

[Barr 08b] S. Barré and O. Bonaventure. “Shim6 Implementation Report : LinShim6”. Internet Draft (work in progress, IETF, July 2008. draft-barre-shim6-impl-02.

[Benv 06] C. Benvenuti. *Understanding Linux Network Internals*. O’Reilly, 2006.

[Boul 09] N. Bouliane and J. Engelhardt. “Writing your own Netfilter modules”. `http://jengelh.medozas.de/documents/Netfilter_Modules.pdf`, Mar. 2009.

[Ches 03] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Professional, 2nd Ed., 2003.

[Clar 88] D. D. Clark. “The design philosophy of the DARPA Internet protocols”. *Proc. SIGCOMM*, Vol. 18, No. 4, pp. 106–114, Aug. 1988.

[Cont 98] A. Conta, Lucent, and S. Deering. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)". RFC 2463, IETF, Dec. 1998.

[Donz 04] F. Donzé. "IPv6 Autoconfiguration". *The Internet Protocol Journal*, Vol. 7, No. 2, June 2004.

[Kent 98] S. Kent, B. Corp, and R. Atkinson. "IP Authentication Header". RFC 2402, IETF, Nov. 1998.

[Kis 09] A. Kis-Szabo. *iptables - manpage*. Apr. 2009. iptables version 1.4.3.2.

[Koza 07] Y. Kozakai, H. Yoshifuji, H. Esaki, and J. Murai. "IPv6 Specific Issues to Track States of Network Flows". *SIGCOMM*, Aug. 2007.

[Kris 08] S. Krishnan. "Guidelines for firewall administrators regarding MIPv6 traffic". Internet Draft (work in progress), IETF, Apr. 2008. draft-krishnan-mip6-firewall-admin-04.

[Le 06] F. Le, CMU, S. Faccin, B. Patil, Nokia, H. Tschofenig, and Siemens. "Mobile IPv6 and Firewalls: Problem Statement". RFC 4487, IETF, 2006.

[Lero 06] D. Leroy. *Assessment software development for distributed firewalls*. Master's thesis, Université catholique de Louvain (UCL), Belgium, jun 2006.

[Maim 96] U. Maimon. "Port Scanning without the SYN flag". *Phrack Magazine*, 1996.

[McKe 06] P. E. McKenney. "What is RCU, Really?". *Linux Weekly News*, 2006.

[Mekk 07] M. Mekking. *Formalization and Verification of the SHIM6 Protocol*. Master's thesis, Radboud University, 2007.

[Nord 05] E. Nordmark. "Threats Relating to IPv6 Multihoming Solutions". RFC 4218, IETF, Oct. 2005.

[Nord 09] E. Nordmark and M. Bagnulo. "Shim6: Level 3 Multihoming Shim Protocol for IPv6". Internet Draft (work in progress), IETF, 2009. draft-ietf-shim6-proto-12.

[Rooi 00] G. van Rooij. "Real Stateful TCP Packet Filtering in IP Filter". `http://www.usenix.org/events/sec01/invitedtalks/rooij.pdf`, 2000.

[Russ 02a] R. Russell. "Linux 2.4 Packet Filtering HOWTO". `http://netfilter.org/documentation/HOWTO//packet-filtering-HOWTO.a4.ps`, Jan. 2002.

[Russ 02b] R. Russell. "Linux Netfilter Hacking HOWTO". http://netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.a4.ps, Mar. 2002.

[Wehr 04] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler. *The Linux Networking Architecture*. Prentice-Hall, Inc., 2004.

[Zwic 00] E. Zwicky, S. Cooper, and D. Chapman. *Building Internet Firewalls*. Vol. 2, O'Reilly, 2000.

Chapter A

Code of the shim6-firewall

The code of the shim6-firewall can be found on the CD which is provided with this thesis.

A.1 Patches to Linux Kernel 2.6.24

For easy usage, the code is in git-patch format so it can easily be applied to Linux Kernel 2.6.24. However the appliance need to be done in the right order:

1. [PATCH] netfilter: Addition of an IPv6 extension header framework for the connection tracker
2. [PATCH] netfilter: shim6 connection tracker and iptables support for shim6
3. [PATCH] ip6tables: addition of a -m shim6 module
4. [PATCH] ip6tables: ipv6header with shim6-support
5. [PATCH] xtables: ulid-match support

Of course, the iptables modifications need the corresponding patches for the iptables userspace (based on iptables version 1.4.2):

- [PATCH] ip6tables: add support for shim6 in ipv6header
- [PATCH] ip6tables: shim6 module support
- [PATCH] xtables: ulid module support

To use the different modules in the Linux Kernel, during the Kernel configuration the following new options must be enabled:

NF_CONNTRACK_SHIM6 connection tracking support for shim6 (under Networking→Networking support→Network options→Network Packet Filtering Framework→IPv6: Netfilter Configuration→IPv6 connection tracking support)

IP6_NF_MATCH_SHIM6 -m shim6 match for ip6tables (under Networking→Networking support→Network options→Network Packet Filtering Framework→IPv6: Netfilter Configuration→IP6 tables support)

NETFILTER_XT_MATCH_ULID -m ulid match for xtables (under Networking→Networking support→Network options→Network Packet Filtering Framework→Core Netfilter Configuration→Netfilter Xtables support)

A.2 Files modified

Besides numerous minor modifications in several parts of the Linux Kernel, the main features of the shim6 connection tracker has been implemented in the following files:

Extension header framework This one has been implemented in the file `net/ipv6/netfilter/nf_conntrack_exthdr.c`
The handlers for the different protocols (Authentication Header, . . .) are `net/ipv6/netfilter/nf_conntrack_exthdr_*.c`
To register an extension header handler into the framework, the function `nf_conntrack_exthdr_register` must be called and the header file `net/netfilter/ipv6/nf_conntrack_exthdr.h` needs to be included.

shim6 handler The shim6 extension header handler is implemented in `net/ipv6/netfilter/nf_conntrack_exthdr_shim6.c`. The layer-4 shim6 handler (handling the shim6 control messages) is implemented in `net/ipv6/netfilter/nf_conntrack_proto_shim6.c`.

More details about the modified files can be found in the README file and the git patches that are provided with the CD-Rom.

Contributions to related projects

While working on the implementation of the shim6-firewall, several contributions were made to related projects. These contributions are not directly related to the shim6-firewall, but rather bug fixes and design changements to these projects. Following are the git-commits that where included in these projects.

B.1 LinShim6

This patch moves the packet listener of the shim6 implementation from Sébastien Barré ([Barr 08a]) to be **after** the iptables filtering rules.

Listing B.1: Included since LinShim6 0.9

```
commit 3436042a8b2a67793a34b9f13ca60b953b12c6be
Author: Christoph Paasch <christoph.paasch@gmail.com>
Date:   Fri Nov 28 21:01:42 2008 +0100

    move the packet-listener to be after the filter

    This, to avoid context-triggering of packets who will be filtered
    afterwards by netfilter.

diff --git a/net/ipv6/shim6_pkt_listener.c b/net/ipv6/shim6_pkt_listener.
    c
index d04bf55..aed6005 100644
--- a/net/ipv6/shim6_pkt_listener.c
+++ b/net/ipv6/shim6_pkt_listener.c
@@ -349,13 +349,13 @@ static struct nf_hook_ops shim6_hook_ops[] = {
    .owner=THIS_MODULE,
    .pf=PF_INET6,
    .hooknum=NF_IP6_LOCAL_IN,
-   .priority=NF_IP6_PRI_CONNTRACK-1 /*Just before connection tracking*/,
+   .priority=NF_IP6_PRI_FILTER+1 /*Just after the filter*/,
```

```
  },
  {.hook=shim6list_local_out,
   .owner=THIS_MODULE,
   .pf=PF_INET6,
   .hooknum=NF_IP6_LOCAL_OUT,
-  .priority=NF_IP6_PRI_CONNTRACK-1 /*Just before connection tracking*/,
+  .priority=NF_IP6_PRI_FILTER+1 /*Just after the filter*/,
  },
 };
```

B.2 Linux Kernel

Some bug fixes that occupied me while chasing for the bugs in my implementation.

Listing B.2: Included since Linux Kernel 2.6.29

```
commit ec8d540969da9a70790e9028d57b5b577dd7aa77
Author: Christoph Paasch <christoph.paasch@gmail.com>
Date:   Mon Mar 16 15:51:29 2009 +0100

    netfilter: conntrack: fix dropping packet after l4proto->packet()

    We currently use the negative value in the conntrack code to encode
    the packet verdict in the error. As NF_DROP is equal to 0, inverting
    NF_DROP makes no sense and, as a result, no packets are ever dropped.

    Signed-off-by: Christoph Paasch <christoph.paasch@gmail.com>
    Signed-off-by: Pablo Neira Ayuso <pablo@netfilter.org>
    Signed-off-by: Patrick McHardy <kaber@trash.net>

diff --git a/net/netfilter/nf_conntrack_core.c b/net/netfilter/
    nf_conntrack_core.c
index 90ce9dd..f4935e3 100644
--- a/net/netfilter/nf_conntrack_core.c
+++ b/net/netfilter/nf_conntrack_core.c
@@ -726,7 +726,7 @@ nf_conntrack_in(struct net *net, u_int8_t pf,
    unsigned int hooknum,
  NF_CT_ASSERT(skb->nfct);

  ret = l4proto->packet(ct, skb, dataoff, ctinfo, pf, hooknum);
- if (ret < 0) {
+ if (ret <= 0) {
    /* Invalid: inverse of the return code tells
     * the netfilter core what to do */
    pr_debug("nf_conntrack_in: Can't track with proto module\n");
```

```
diff --git a/net/netfilter/nf_conntrack_proto_tcp.c b/net/netfilter/
    nf_conntrack_proto_tcp.c
index a1edb9c..f3fd154 100644
--- a/net/netfilter/nf_conntrack_proto_tcp.c
+++ b/net/netfilter/nf_conntrack_proto_tcp.c
@@ -859,7 +859,7 @@ static int tcp_packet(struct nf_conn *ct,
          */
        if (nf_ct_kill(ct))
          return -NF_REPEAT;
-       return -NF_DROP;
+       return NF_DROP;
     }
     /* Fall through */
   case TCP_CONNTRACK_IGNORE:
@@ -892,7 +892,7 @@ static int tcp_packet(struct nf_conn *ct,
          nf_log_packet(pf, 0, skb, NULL, NULL, NULL,
              "nf_ct_tcp: killing out of sync session ");
        nf_ct_kill(ct);
-       return -NF_DROP;
+       return NF_DROP;
     }
     ct->proto.tcp.last_index = index;
     ct->proto.tcp.last_dir = dir;
```

Listing B.3: Included since Linux Kernel 2.6.29

```
commit d1238d5337e8e53cddea77c2a26d26b6eb5a982f
Author: Christoph Paasch <christoph.paasch@gmail.com>
Date:   Mon Mar 16 15:52:11 2009 +0100

    netfilter: conntrack: check for NEXTHDR_NONE before header sanity
        checking

    NEXTHDR_NONE doesn't has an IPv6 option header, so the first check
    for the length will always fail and results in a confusing message
    "too short" if debugging enabled. With this patch, we check for
    NEXTHDR_NONE before length sanity checkings are done.

    Signed-off-by: Christoph Paasch <christoph.paasch@gmail.com>
    Signed-off-by: Pablo Neira Ayuso <pablo@netfilter.org>
    Signed-off-by: Patrick McHardy <kaber@trash.net>

diff --git a/net/ipv6/netfilter/nf_conntrack_reasm.c b/net/ipv6/netfilter
    /nf_conntrack_reasm.c
index ed4d79a..058a5e4 100644
--- a/net/ipv6/netfilter/nf_conntrack_reasm.c
+++ b/net/ipv6/netfilter/nf_conntrack_reasm.c
```

```
@@ -528,14 +528,14 @@ find_prev_fhdr(struct sk_buff *skb, u8 *prevhdrp,
    int *prevhoff, int *fhoff)
    if (!ipv6_ext_hdr(nexthdr)) {
      return -1;
    }
-   if (len < (int)sizeof(struct ipv6_opt_hdr)) {
-     pr_debug("too short\n");
-     return -1;
-   }
    if (nexthdr == NEXTHDR_NONE) {
      pr_debug("next header is none\n");
      return -1;
    }
+   if (len < (int)sizeof(struct ipv6_opt_hdr)) {
+     pr_debug("too short\n");
+     return -1;
+   }
    if (skb_copy_bits(skb, start, &hdr, sizeof(hdr)))
      BUG();
    if (nexthdr == NEXTHDR_AUTH)
```

Listing B.4: Included since Linux Kernel 2.6.29

```
commit fe2a7ce4de07472ace0cdf460a41f462a4621687
Author: Christoph Paasch <christoph.paasch@student.uclouvain.be>
Date:   Wed Feb 18 16:28:35 2009 +0100

    netfilter: change generic l4 protocol number

    0 is used by Hop-by-hop header and so this may cause confusion.
    255 is stated as 'Reserved' by IANA.

    Signed-off-by: Christoph Paasch <christoph.paasch@student.uclouvain.
        be>
    Signed-off-by: Patrick McHardy <kaber@trash.net>

diff --git a/net/netfilter/nf_conntrack_proto_generic.c b/net/netfilter/
    nf_conntrack_proto_generic.c
index 4be80d7..829374f 100644
--- a/net/netfilter/nf_conntrack_proto_generic.c
+++ b/net/netfilter/nf_conntrack_proto_generic.c
@@ -92,7 +92,7 @@ static struct ctl_table generic_compat_sysctl_table[] =
     {
 struct nf_conntrack_l4proto nf_conntrack_l4proto_generic __read_mostly =
 {
  .l3proto      = PF_UNSPEC,
- .l4proto      = 0,
+ .l4proto      = 255,
```

```
  .name       = "unknown",
  .pkt_to_tuple    = generic_pkt_to_tuple,
  .invert_tuple    = generic_invert_tuple,
```

Listing B.5: Included since Linux Kernel 2.6.29-3

```
commit b98b4947cb79d670fceca0e951c092eea93e9baa
Author: Christoph Paasch <christoph.paasch@gmail.com>
Date:   Tue May 5 15:32:16 2009 +0200

    netfilter: ip6t_ipv6header: fix match on packets ending with
        NEXTHDR_NONE

    As packets ending with NEXTHDR_NONE don't have a last extension
        header,
    the check for the length needs to be after the check for NEXTHDR_NONE
        .

    Signed-off-by: Christoph Paasch <christoph.paasch@gmail.com>
    Signed-off-by: Patrick McHardy <kaber@trash.net>

diff --git a/net/ipv6/netfilter/ip6t_ipv6header.c b/net/ipv6/netfilter/
    ip6t_ipv6header.c
index 14e6724..91490ad 100644
--- a/net/ipv6/netfilter/ip6t_ipv6header.c
+++ b/net/ipv6/netfilter/ip6t_ipv6header.c
@@ -50,14 +50,14 @@ ipv6header_mt6(const struct sk_buff *skb, const
    struct xt_match_param *par)
    struct ipv6_opt_hdr _hdr;
    int hdrlen;

-   /* Is there enough space for the next ext header? */
-   if (len < (int)sizeof(struct ipv6_opt_hdr))
-     return false;
    /* No more exthdr -> evaluate */
    if (nexthdr == NEXTHDR_NONE) {
      temp |= MASK_NONE;
      break;
    }
+   /* Is there enough space for the next ext header? */
+   if (len < (int)sizeof(struct ipv6_opt_hdr))
+     return false;
    /* ESP -> evaluate */
    if (nexthdr == NEXTHDR_ESP) {
      temp |= MASK_ESP;
```

Listing B.6: Will be in the Linux Kernel 2.6.30

```
commit 9d2493f88f846b391a15a736efc7f4b97d6c4046
Author: Christoph Paasch <christoph.paasch@gmail.com>
Date:   Mon Mar 16 15:15:35 2009 +0100

    netfilter: remove IPvX specific parts from nf_conntrack_l4proto.h

    Moving the structure definitions to the corresponding IPvX specific
        header files.

    Signed-off-by: Christoph Paasch <christoph.paasch@gmail.com>
    Signed-off-by: Patrick McHardy <kaber@trash.net>

diff --git a/include/net/netfilter/nf_conntrack_l4proto.h b/include/net/
    netfilter/nf_conntrack_l4proto.h
index debdaf7..16ab604 100644
--- a/include/net/netfilter/nf_conntrack_l4proto.h
+++ b/include/net/netfilter/nf_conntrack_l4proto.h
@@ -90,10 +90,7 @@ struct nf_conntrack_l4proto
   struct module *me;
 };

-/* Existing built-in protocols */
-extern struct nf_conntrack_l4proto nf_conntrack_l4proto_tcp6;
-extern struct nf_conntrack_l4proto nf_conntrack_l4proto_udp4;
-extern struct nf_conntrack_l4proto nf_conntrack_l4proto_udp6;
+/* Existing built-in generic protocol */
 extern struct nf_conntrack_l4proto nf_conntrack_l4proto_generic;

 #define MAX_NF_CT_PROTO 256
diff --git a/net/ipv6/netfilter/nf_conntrack_l3proto_ipv6.c b/net/ipv6/
    netfilter/nf_conntrack_l3proto_ipv6.c
index 727b953..e6852f6 100644
--- a/net/ipv6/netfilter/nf_conntrack_l3proto_ipv6.c
+++ b/net/ipv6/netfilter/nf_conntrack_l3proto_ipv6.c
@@ -26,6 +26,7 @@
 #include <net/netfilter/nf_conntrack_l4proto.h>
 #include <net/netfilter/nf_conntrack_l3proto.h>
 #include <net/netfilter/nf_conntrack_core.h>
+#include <net/netfilter/ipv6/nf_conntrack_ipv6.h>

 static bool ipv6_pkt_to_tuple(const struct sk_buff *skb, unsigned int
     nhoff,
               struct nf_conntrack_tuple *tuple)
diff --git a/net/netfilter/nf_conntrack_proto_tcp.c b/net/netfilter/
    nf_conntrack_proto_tcp.c
index a1edb9c..7d3944f 100644
--- a/net/netfilter/nf_conntrack_proto_tcp.c
```

```
+++ b/net/netfilter/nf_conntrack_proto_tcp.c
@@ -25,6 +25,8 @@
 #include <net/netfilter/nf_conntrack_l4proto.h>
 #include <net/netfilter/nf_conntrack_ecache.h>
 #include <net/netfilter/nf_log.h>
+#include <net/netfilter/ipv4/nf_conntrack_ipv4.h>
+#include <net/netfilter/ipv6/nf_conntrack_ipv6.h>

 /* Protects ct->proto.tcp */
 static DEFINE_RWLOCK(tcp_lock);
diff --git a/net/netfilter/nf_conntrack_proto_udp.c b/net/netfilter/
    nf_conntrack_proto_udp.c
index 2b8b1f5..d402117 100644
--- a/net/netfilter/nf_conntrack_proto_udp.c
+++ b/net/netfilter/nf_conntrack_proto_udp.c
@@ -22,6 +22,8 @@
 #include <net/netfilter/nf_conntrack_l4proto.h>
 #include <net/netfilter/nf_conntrack_ecache.h>
 #include <net/netfilter/nf_log.h>
+#include <net/netfilter/ipv4/nf_conntrack_ipv4.h>
+#include <net/netfilter/ipv6/nf_conntrack_ipv6.h>

 static unsigned int nf_ct_udp_timeout __read_mostly = 30*HZ;
 static unsigned int nf_ct_udp_timeout_stream __read_mostly = 180*HZ;
```

B.3 iptables

A patch for a compiler warning got included in the userspace iptables implementation.

Listing B.7: Included since iptables 1.4.3

```
commit 7cd15e367cc81c839ef2ca061d201c46ca1deb7c
Author: Christoph Paasch <christoph.paasch@gmail.com>
Date:   Mon Mar 23 13:50:11 2009 +0100

    libiptc: avoid compile warnings for iptc_insert_chain

    iptc_insert_chain is too big to get inlined and so it generates
    a warning while compiling.

    Signed-off-by: Christoph Paasch <christoph.paasch@gmail.com>
    Signed-off-by: Patrick McHardy <kaber@trash.net>

diff --git a/libiptc/libiptc.c b/libiptc/libiptc.c
index 544a5b2..c3d9bfc 100644
```

```
--- a/libiptc/libiptc.c
+++ b/libiptc/libiptc.c
@@ -841,7 +841,7 @@ static int __iptcc_p_del_policy(struct xtc_handle *h,
     unsigned int num)
 }

 /* alphabetically insert a chain into the list */
-static inline void iptc_insert_chain(struct xtc_handle *h, struct
    chain_head *c)
+static void iptc_insert_chain(struct xtc_handle *h, struct chain_head *c
    )
 {
  struct chain_head *tmp;
  struct list_head  *list_start_pos;
```