NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

IPv6 Packet Filtering

by

Orla McGann, B.Eng

Masters Thesis

Submitted to the National University of Ireland Maynooth

Department of Electronic Engineering

Hamilton Institute

National University of Ireland Maynooth

Maynooth

Co. Kildare

January 2005

Research Supervisor: Dr. David Malone

Co-Supervisors: Prof. Douglas Leith and Prof. John T. Lewis

Contents

List of Figures

List of Tables

Abstract

The Firewall, and packet filters in particular, have become an essential part of the Internet. They provide protection to individual hosts and networks from the nasty elements of the Internet: attackers, viruses and worms. The creation of a new version of the Internet Protocol, IPv6, has been hailed as the solution to many of the ills of the current protocol. It expands the address space sufficiently for future use, has a well thought out address assignment policy to make the filtering of routes simpler, has improved network management facilities and last but not least, it has in-built security via IPsec.

Even with all this, the packet filter is still expected to be a part of the IPv6 Internet at least until the problems associated with the automatic exchange of keys for IPsec are resolved. This thesis will discuss the challenges posed in filtering IPv6 packets by both stateless and dynamic means. It will also gather together all the current packet filtering techniques available for IPv4. An introduction to the new version of the IP protocol, IPv6, is presented in Chapter 1.

Chapters 2 and 3 deal with the current methods of stateless and dynamic filtering for IPv4. Chapter 4 discusses the stateless filtering of IPv6 packets, including the issues that arise in filtering the Neighbour Discovery protocol, ICMPv6 packets and Extension Headers. This chapter includes a gathering together of some of the operational experiences reported by those currently filtering IPv6 packets.

Chapter 5 discusses the dynamic filtering of IPv6. This chapter contains the bulk of my independent study. Here I propose using the Flow Label field of the IPv6 header as an additional piece of state to be tracked by the dynamic filter. In order to see if this was a valid option, I first had to establish whether operating systems set this Flow Label value consistently throughout the life time of a connection. I describe tests used to determine this and investigate some of the anomalous results that were obtained. Finally, I discuss possible future work in this area.

Ruleset Notation

As most of the commercial and open source firewalls available use a distinct syntax for their rule sets, I use a generic syntax to illustrate the packet filtering techniques described in this thesis. All rules, except for those that are explicitly mentioned as being of a different syntax, given as examples in this thesis will be described using the following generic syntax notation. The Backus Naur Form (BNF)[24] is described as:

```
   rule ::= action policy proto addr-list [direction] [via-iface] [options]

action ::= ‘‘add’’ | ‘‘remove’’
policy ::= ‘‘allow’’ |‘‘deny’’
proto ::= ‘‘all’’ | ‘‘tcp’’ | ‘‘udp’’ | ‘‘ip’’ | ‘‘ipv6’’ | ‘‘icmp’’ | ‘‘ipv6-icmp’’ | number
addr-list ::= ‘‘from’’ addr-mask [ports] ‘‘to’’ addr-mask [ports]
direction ::= ‘‘in’’ | ‘‘out’’
via-iface ::= ‘‘via’’ interface-name
options ::= [fake-state] [state-ops] [tcpflags] [icmptype]

addr-mask ::= ip-addr/[netmask] | ipv6-addr/[netmask] | host
ports ::= port-num | port-range | service
port-range ::= portnum - portnum
port-num ::= number

fake-state ::= ‘‘setup’’ | ‘‘established’’ | ‘‘related’’ | ‘‘reply’’
state-ops ::= ‘‘keep-state’’ | ‘‘check-state’’
tcp-flags ::= [syn] [ack] [fin] [rst] [psh] [urg] [ecn]
icmptype ::= [time-exceeded] [port-unreachable] [host-unreachable] [echo-reply] [echo-request]
[don’t-fragment] [dest-unreachable] [parameter-problem] [packet-too-big]
ip-addr ::= any ipv4 address
ipv6-addr ::= any ipv6 address
netmask ::= number <1 - 32> | number <1 - 128>
service ::= [ssh] | [dns] | [ftp] | [telnet] | [smtp] | [http] | [https] | number
host ::= hostname
```

Acknowledgements

There are many people who contributed to the effort that is my masters thesis. This acknowledgements section will work on a "last match" basis, so I will start with a default thank you statement to cover everyone who helped me along the way lest I inadvertently leave them out later on: Thank you!

Firstly, I would like to thank my supervisor Dr. Dave Malone for taking a chance on an unknown kid and being mostly convinced I'd be able to do this. It was enough to make me turn down two good network engineering jobs and break my mother's heart. Again. Oh and most especially, for imparting his brand spanking new Powerbook for me to work on at home. I miss it terribly!

I would like to thank Prof. Douglas Leith and Dr. Robert Shorten for plucking us from the roadside in Rathmines and giving us a new home in the lovely Hamilton Institute, after Professor John Lewis's untimely death (R.I.P).

I'd like to thank Ken Duffy for all his LaTeX help and for pointing out that I'd be paid so much more in Maynooth than in DIT :). I'd like to thank Wayne G. Sullivan for providing LaTeX help on the rare occasions that Dave, Ken and Peter couldn't. I would also like to thank Ian Dowse and Niall Murphy for proof reading my thesis and providing useful corrections and criticisms.

I'd like to thank the following people for being a welcome source of distraction, office mates and in some cases providing the strangest lunch time conversations a gal could ever ask for: Peter Clifford, Mark Lee, Finn McLeod (of the clan McLeod), Karol Hennessy, Will Lee, Rade Stanojevic, Steve (Michella) Strachan, Andy ("squirrel") Crossan, Paul O'Grady, Carlos Villegas, Brian Keegan, Tony Metcalfe and Simon McCann. A special mention must also go to the Hamilton chicks: Rosemary, Kate, Parisa and Sandrine. May there be more of us soon.

I'd like to thank my (pent)house-mates Tony and Guy; firstly for letting me move in with them, then for letting me usurp the dining table for so long and finally for tip-toeing around me when I was working. I'd also like to thank Barak Pearlmutter, who may as well be a housemate, for feeding me and introducing me to ginger tea — the perfect study aid.

Finally, I would like to extend my thanks to my family and friends for all their help and moral support; in particular Mark McLaughlin, Etain O'Leary, Ann and Peter Kennedy, James Raftery, Eamonn Hoban-Shelley and Renee Krautter.

Chapter 1

Introducing version Six of the Internet Protocol

The Internet Protocol (IP) is part of the TCP/IP network suite that is widely deployed on the Internet. It is used to provide addresses to distinctly identify hosts, so that they can communicate with each other. Its current form, IPv4, has remained relatively unchanged since it was adopted by ARPAnet in the late 1970s. Unfortunately, the unprecedented need for IP addresses has meant that there has been a rapid decline in the number of IP addresses available for allocation. The manner in which the addresses were disseminated originally, classed based networks, also contributed to this shortage because large numbers of addresses were allocated but remained unused.

The Internet Engineering Task Force (IETF) formally approved the new version of the protocol, version Six (IPv6), on November 17th 1994. The main objectives of this new version were to expand the address space sufficiently for future use and also to make improvements to IPv4 where necessary; particularly in the areas of security, network scalability and quality of service.

The workgroup developing this new version of the protocol decided that the address size should be expanded from the 32-bit addresses of IPv4 to 128-bit addresses. It is hoped that this will provide enough addresses for all manner of Internet connected devices for the foreseeable future: mobile phones, pocket PCs, printers, scanners, routers, fridges, toasters etc. These new addresses are discussed further in section 1.3.

IPv4 and IPv6 are not interoperable; there are separate networking stacks for each protocol and a host (or router) must have both in order to recognise and process the header information in a packet where both exist on a network[1].

1.1 The IPv6 Header

The IPv6 header is based on the old IPv4 header (see figure 1.1). There are still fields for the source and destination addresses, but they have been expanded to accommodate the larger addresses and the version field is set to six instead of four. Fields that were obsolete or deemed unnecessary were removed completely: the Header Length field (the IPv6 header is a fixed length of 40 bytes), the Checksum field and the fields used for Fragmentation (ID, Offset and Don't Fragment bit). All the optional fields were removed from the main IPv6 header and made into "Extension Headers".

The other changes that were made were more of a restructuring nature. The functionality of the Type-of-Service (ToS) field in the IPv4 header was replaced by the Traffic Class field and the Flow Label field. These are used to provide support for real-time delivery of data, so-called Quality of Service (QoS). The Payload Length field has replaced the Total Length field, as the length of the IPv6 header is fixed there is no longer any need for this to be taken into account when calculating the length of the packet (the optional headers in IPv4 were of variable length, thus so was the overall header). The Payload Length field can register a data payload of up to 64kB, or more if the Jumbogram option is specified (see section 1.2). The Time-To-live field was replaced by the Hop Limit field. It operates in the same way as in IPv4, with each router the packet passes through decrementing the field by one, indicating each "hop" in an end-to-end route.

The old Protocol Type field has been replaced by the Next Header field; this contains the code that indicates whether an extension header follows the main header. The Next Header field is also used to specify the transport layer protocol of the packet payload, just as in IPv4. These codes are the same as they are in IPv4: TCP (6), UDP (17) and ICMPv6 (58). A Next Header value of "59" means there is no next header in the packet.

This restructuring has resulted in a much simpler header compared to its IPv4 counterpart; the IPv4 header has thirteen fields in it whereas the IPv6 header has only eight. Even though the new

[1]This is no different to networks running IP and Appletalk, or IP and Novell's IPX.

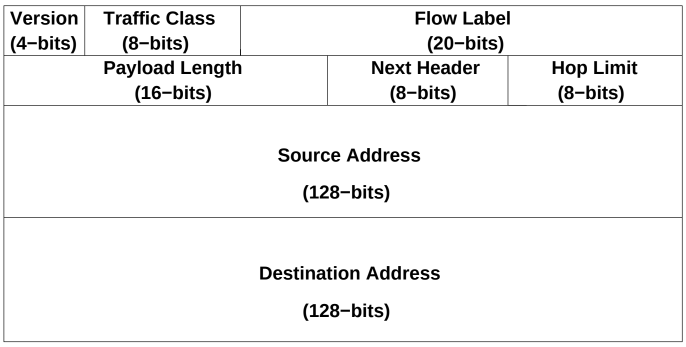

Figure 1.1: The IPv6 Header

addresses are four times the size of the IPv4 addresses, the restructuring has resulted in a new header that is only twice the size of its predecessor. This is intended to offset the bandwidth cost of using these larger addresses and it should help to make routing these packets more efficient.

1.2 Extension Headers

As previously mentioned, all the options fields of the IPv4 header are not present in the main IPv6 header and flexible extension headers that are placed in between the IPv6 header and the transport layer header were created instead. These extension headers provide support in IPv6 for features, such as security (in the form of IPsec), source routing, network management and fragmentation.

There are six Extension Headers: Hop-by-Hop option, Destination option, Routing, Fragment, Authentication and Encapsulation Security Payload. Different extension headers can be chained together in a packet. Each extension header also has a Next Header field, which is used to identify the header following it. Figure 1.2 shows this chaining process and table 1.1 contains the next header codes.

Extension headers should always be chained together in the order they are listed above. This is to facilitate the processing of these headers at the destination. The Hop-by-Hop options header must *always* follow the main IPv6 header, as it is the only extension header that must be examined by intermediate nodes.

The Jumbogram and Router Alert options are part of the Hop-by-hop extension header. Jumbo-

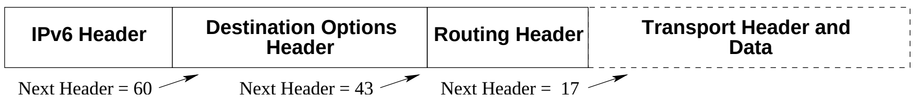

Figure 1.2: The Extension Headers

grams are used to send packets with a larger data payload than the 64kB specified by the Payload Length field in the IPv6 header. To implement this, the value of the Payload length field should be set to zero and the Jumbogram extension header attached; then a much larger payload of up to 4GB can be sent in the one IPv6 datagram (on links with a high enough MTU). The Router Alert Hop-by-Hop option is used to notify transit routers that they should examine the contents of the packet more thoroughly before forwarding it on. This option is used to specify that the datagram requires special processing by the nodes en-route.

Each extension header only occurs once per packet at most, except the Destination options header. The first instance of the Destination option is used to carry information to the destinations listed in the destination address field in the IPv6 header and the addresses in the Routing header, and the second is for optional information that is only to be read by the final destination[17].

The Routing header is used to specify intermediate nodes the packet must pass through on the way to the destination. Different "types" of routing headers may be used. The "type 0" routing header is similar to IPv4's loose source routing option. Its header comprises of a next header field, which identifies the header following it (as before); the Hdr Extn Length, which gives the length of the routing header (in the type 0 case, the Hdr Extn Length is twice the number of addresses given in the header); the routing header type; the Segments Left field, which identifies how many nodes must still be visited by the packet; a 32-bit reserved field that is to be ignored; and finally the addresses that must be visited en-route by the packet.

In IPv6, fragmentation of a packet is only permitted when it is performed by the source node. Routers are not allowed to fragment a packet. If a packet is received by a router that is too big for the link, it must be discarded and an Internet Control Message (ICMPv6) must be sent back to the source of the packet to inform them the packet was dropped (see section 1.5). Removing the option to fragment packets en-route should result in fewer problems for hosts and routers, such as crack attempts using overlapping fragments and broken Path MTU (this is described further in section 2.5).

Extension Header	**Subtype**	**Next Header Code**
Hop-by-hop		0
	Padding	
	Router Alert	
	Jumbo Payload	
Destination Options		60
	Padding	
	Binding Update	
	Binding Acknowledgement	
	Binding Request	
	Home Address	
Routing		43
	Type 0	
Fragment		44
Authentication		51
Encapsulation Security Payload		50

Table 1.1: Table of IPv6 Extension Headers

If a source wishes to fragment a packet it uses the Fragment extension header. The original packet that is too large is divided up into two sections: the “unfragmentable part”, which contains the IPv6 header and all extension headers which must be processed by nodes en-route to the final destination (i.e. the Hop-by-Hop option, and the Destination option and Routing header where specified); and the “fragmentable part”, which consists of the rest of the extension headers (if there are any) and the payload data. The fragmentable part is divided into fragments of multiples of 8-octets long, except possibly the final fragment, and each fragment is prefixed by the “unfragmentable part” and the fragment header.

Security was also a major concern of the IETF when it was designing IPv6. They aimed to establish three important security services: packet authentication, packet integrity, and packet confidentiality. These security features are provided by IPsec[45] via the Authentication (AH) and Encapsulation Security Payload (ESP) extension headers.

The AH provides integrity validation, which guarantees that a packet comes where it claims to have come from. This is achieved by the exchange of cryptographic keys, either manually or automatically (using Internet Key Exchange (IKE)). Before each packet is sent, the header creates a checksum based on the key agreed by both hosts (typically a MD5 hash). This hash is then re-run on the receiving end and is compared to the original checksum.

The AH is used to prevent host spoofing attacks and packet modification attempts, but it does not provide any protection against packet sniffing. The ESP extension header is used to provide packet confidentiality as well as the same security services that AH provides. This high level of privacy and integrity for packets was unheard of in IPv4, except in the case of Secure Socket Layer (SSL) applications or where the IPv4 implementation of IPsec was deployed; IPsec was back-ported to IPv4, but was not made an integral part of the protocol as in IPv6.

ESP can be deployed in two ways: transport mode or tunnel mode. In transport mode the encryption is applied to the transport layer and other upper layer protocols but not the IPv6 header or any other extension headers that come before the AH and ESP headers. With Tunnel mode ESP, the entire packet is encrypted including all of the IPv6 header, and a new header prefixes the encapsulated encrypted packet. RFC 2402[46] describes the difference between AH and ESP as:

“The primary difference between the authentication provided by ESP and AH is the extent of the coverage; ESP does not protect the IP header fields unless those fields are encapsu-

lated by ESP (tunnel mode).”

1.3 New Addressing Syntax

The addressing syntax for IPv6 is completely different to the current 32-bit “dotted quad” format that is used for IPv4. In IPv6, the 128-bit addresses are broken up on 16-bit boundaries, with each boundary separated by a colon (as opposed to the dot used in IPv4 addresses). Then each 16-bit block is converted from binary numbers to a 4 digit hexadecimal notation, instead of decimal as in IPv4. For example: The IPv6 address `2001:0770:011d:0000:020b:dbff:fe63:545d` is derived from the binary sequence:

```
0010000000000001:0000011101110000:0000000100011101:0000000000000000:
0000001000001011:1101101111111111:1111111001100011:0101010001011101
```

This binary sequence is simplified by removing the leading zeroes in each 16-bit block. Each block must have at least one digit in it, except where there is a continuous set of one or more zeroed blocks; in which case this can be compressed to “::”. It is only possible to remove one section of zeroed blocks and replace them with the “::” notation, as the address becomes ambiguous otherwise. Thus, the IPv6 address given above can be written in the simplified “colon hexadecimal” notation as: `2001:770:11d::20b:dbff:fe63:545d`.

Obviously, remembering these addresses is out of the question and Domain Name System (DNS) has been modified for IPv6 to provide the same host to address mapping as exists in IPv4; the details of which are outside the scope of this thesis[1].

IPv6 networks are written as `ipv6_address/prefix_length` in Classless Inter-Domain Routing (CIDR) format. The prefix length denotes, in decimal notation, the size of the network; i.e. a `/48` prefix length means the first 48-bits of the address are fixed and the remaining 80 bits are available for use within the network itself.

1.3.1 Interface Identifiers

The interface identifier, also known as the unique identifier, distinguishes an interface on a network and makes sure that there is no address duplication on the subnet. The interface ID is usually 64-bits long if autoconfiguration is used; it is comprised of the 48-bit Ethernet MAC address, with the remaining 16-bits padded out with the modified Extended Unique Identifier (EUI-64)[31] constant: `fffe`. The 7th bit (the universal bit) of the MAC address must also be set to 1 from 0. This bit indicates that this is a globally scoped address, as opposed to a local one. However, most implementations set the universal bit on both local and global scoped addresses, so instead it is used to avoid clashes between manually configured addresses and autoconfigured ones.

For example: to create a unique identifier on a network, take the 48-bit MAC address and set the 7th bit to 1. Split it down the middle into two 24-bit sections, and insert the 16-bit HEX digits of "`fffe`" in between the two sections. So for a given MAC address of `00.0b.db.63.54.5d`; first we must set the 7th bit to '1'[2], which gives `02.0b.db.63.54.5d`. This is split up into `02.0b.db` and `63.54.5d`. Then we add in `'fffe'` in between them. Finally we gather the HEX digits into groups of 4 and separate each group with a semicolon to give the IPv6 interface identifier: `020b:dbff:fe63:545d`.

1.4 Address types in IPv6

IPv6 addresses are slightly more complicated than IPv4; in IPv4 most addresses could be categorised into public, private, broadcast, multicast and network addresses. This section describes the different types of addresses available in IPv6 and the context in which they can be used. So far, only about 15% of the vast IPv6 address space has been allocated for use[38]. These allocations have been divided up into different types: Unicast, Multicast and Reserved addresses. The type of IPv6 address is determined by the high-order bits of the address. The remaining 85% is reserved for future use. Table 1.2 contains all the current IPv6 allocations.

IPv6 enabled hosts can have multiple addresses per interface. IPv4 hosts typically have only one IP address which identifies them, but in IPv6 hosts can be reached via different addresses depending on the context they are being contacted in (i.e. their scope). There are a number of different scopes

[2]The HEX digit 00 is 00000000 in binary bits. If the 7th bit is set from 0 to 1 this gives us the binary number 00000010, which converts to 02 in HEX digits

Address Type	**Allocation**
Unspecified Address	::
Loopback Address	::1
Compatible IPv4 Addresses	::0.0.0.0/96
Mapped IPv4 Addresses	::ffff:0.0.0.0/96
Global Unicast	2000::/3
Link-local Addresses	FE80::/10
IETF Reserved (Formerly Site-Local)	FEC0::/10
Multicast Addresses	FF00::/8

Table 1.2: Table of Current IPv6 Address Assignments

defined in the IPv6 Addressing Architecture RFC[31], but the main scopes available for use are Global, Link-Local and Site-Local.

Globally scoped addresses are intended for general use on the Internet. These addresses are assigned by the Regional Internet Registries (RIRs): RIPE NCC, APNIC, ARIN, LACNIC etc. The RIRs make assignments to Internet Service Providers (ISPs), who in turn make allocations to everyone else. This aggregateable allocation structure was chosen to help minimise the number of routes in the global routing table (a major problem in IPv4 due to the way those addresses were allocated), by having all European prefixes allocated by RIPE, all American prefixes allocated by ARIN etc. where feasible. Obviously, there are corporations that span several continents, which may only want one large allocation for the whole organisation.

Link-Local addresses are used for local communication between hosts connected on the same physical link. Each interface is assigned a (unicast) link-local address, which is used in the autoconfiguration process (the details of which are discussed in sections 1.6 and 1.7).

Site-Local addresses are for use within a "site", which is loosely defined as a collection of related subnets. They are intended for use only within a site and packets with these addresses should not leave the boundaries of the site network. Unicast Site-Local addresses are prone to the same problems as private addresses in IPv4: they cannot be used for communication with other networks and if a company merges with another that is also using site-locally scoped addresses there is the potential for address collisions. The other problem associated with site-local addresses is that it is uncertain as

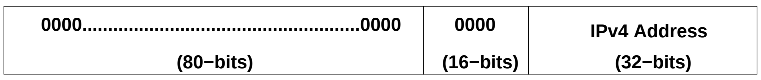

Figure 1.3: Compatible IPv4 Addresses

to how these addresses should be routed; a "site" is a very ambiguous definition and could be a very large network spread out over a number of locations. Site-local addressing is currently being reviewed by the IETF.

1.4.1 Reserved Addresses

There are a number of specially assigned addresses defined in IPv6: the unspecified address, the loopback address, compatible IPv4 addresses and mapped IPv4 addresses. The unspecified address is "0:0:0:0:0:0:0:0", which is usually abbreviated to "::" (as per the notation described earlier). It is used as the source address in packets from an interface that has yet to be assigned an address of its own. It should never be used as a destination address and therefore it doesn't have any scope associated with it.

The loopback address is "0:0:0:0:0:0:0:1" (again, this is abbreviated to "::1") and is similar to the IPv4 address 127.0.0.1. It is used to contact the loopback interface on a host. The IPv6 Scoped Addressing Architecture Internet Draft[14] describes it as:

> *"The IPv6 unicast loopback address, ::1, is treated as having link-local scope within an imaginary link to which a virtual "loopback interface" is attached."*

Compatible IPv4 addresses were defined as part of a transition mechanism from IPv4 to IPv6. The addresses are of the form "::x.x.x.x/96" (see figure 1.3), with the bottom 32-bits defined by the IPv4 address of the host using the Compatible address and the top 96-bits are zeros. The IPv4 address used in compatible IPv4 addresses *must* be a public IPv4 address. The transition mechanism that utilises these addresses is discussed in section 3.4.2.

Mapped IPv4 addresses are used to represent the IPv4 addresses of a host as IPv6 addresses. They are syntactically the same as the compatible IPv4 addresses, but the 16-bits in between the IPv4 address and the leading zeros is "`FFFF`" instead of "`0000`". Figure 1.4 shows the mapped IPv4

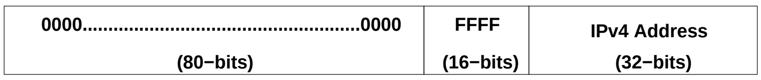

Figure 1.4: Mapped IPv4 Addresses

addresses.

These special addresses are all unicast scoped addresses (except the unspecified address), but they are assigned from the `0000 0000` binary prefix of IPv6 addresses, instead of the `0010 0000` global unicast prefix block.

1.4.2 Unicast Addresses

Unicast addresses are assigned for each interface on a host and are used for "one-to-one" communication; where a packet is sent from one interface and is received by only one interface. There are two main scope of unicast addresses currently in use: Global and Link-Local.

Global unicast addresses are the common form of addressing in IPv6 and are comparable to public IPv4 addresses. Table 1.3 contains the allocations from the global unicast address space. All IPv6 addresses that begin with the "`2001::/16`" prefix are production level addresses, intended for general usage on the Internet. Allocations from this prefix have been made to the Regional Internet Registries[39] and they have allocated prefixes to Local Internet Registries (LIRs)[61].

The "`3FFE::/16`" prefix was allocated for use in the 6-Bone[21], the IPv6 testbed, but this project has been wound down and the use of these addresses is discouraged. They are due to be made obsolete on the 6th of June 2006. Anyone with a 6Bone allocation is free to use these addresses until that date, but they have been formally deprecated[21]. The 6to4 address prefix (`2002::/16`) was allocated to facilitate the transition from IPv4 to IPv6. These addresses are used to automatically encapsulate IPv6 packets in IPv4 packets; this is described further in section 3.4.2.

As mentioned previously, Link-Local addresses are used for local communication. The host uses this address to talk to its neighbours; these are the other hosts that are directly connected on the same link via a hub or switch. All IPv6 enabled hosts must have a Link-Local address assigned to their interfaces. The Link-Local address is created by prefixing the interface identifier with "fe80::",

Address Prefix	**Allocation**
Reserved	2000::/16
Production Addresses (allocated to the RIRs)	2001::/16
6to4 Addresses	2002::/16
6-Bone Addresses (IPv6 Testbed)	3FFE::/16
Reserved	3FFF::/16

Table 1.3: Table of Global Unicast IPv6 Address Assignments

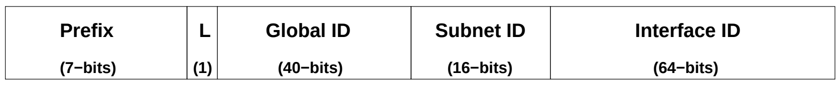

Figure 1.5: Unique Local IPv6 Unicast Addresses

which is the reserved network block for Link-Local addresses. So, the interface described previously has the full link-local IPv6 address assigned to it of: `fe80::020b:dbff:fe63:545d`.

An allocation for another scope of Unicast addresses, Site-Local unicast addresses, had previously been made by IANA. After extensive discussion by the IETF IPv6 working group it was decided that these addresses should be deprecated[35] and replaced with another, less ambiguous, option.

The current suggestion for replacing Site-Local addresses is with "Unique Local IPv6 Unicast Addresses"[30]. These addresses will be globally unique, but not globally *routable*; they are only intended for use within a "site" for local communication.

Figure 1.5 shows the format of Unique Local addresses. The "L" bit is toggled to denote whether the address is locally or centrally assigned its Global ID, which means that the prefix will either be "1111 1100" (FB) or "1111 1101" (FC). The 40-bit Global ID is to be pseudo-randomly chosen to create a globally unique prefix. The subnet ID allows the prefix to be subnetted for administrative purposes (just like a normal global unicast address prefix), and the Interface ID is created the same way as link-local and global interface IDs (as described in section 1.3.1) .

> *"Centrally assigned global IDs are uniquely assigned (by an allocation authority, probably IANA). Local assignments are self generated and do not need any central co-ordination or assignment, but have a lower (but still adequate) probability of being unique."*

Unique Local IPv6 addresses aim to remedy the problems that were inherent in the Site-Local unicast address specification. They should have a globally unique prefix that is well-known, as this facilitates the filtering of these addresses at site boundaries. The addresses are ISP independent and do not require a constant connection, or even any connection, to the Internet. Sites can be combined or privately interconnected without causing addressing conflicts or requiring a renumbering of these services.

1.4.3 Anycast Addresses

Anycast addresses are used for "one-to-one-of-many" connections. They identify multiple interfaces belonging to a number of different nodes, usually on discrete links. A packet sent to an anycast address is delivered to *one* of the interfaces identified by that address; the "nearest" one, as defined by the routing protocol. They do not have their own separate IPv6 block like Unicast and Multicast addresses. Instead, they are allocated from the Unicast address space[31]:

> *"Anycast Addresses are taken from the unicast address space, and are syntactically indistinguishable from unicast addresses."*

Until the use of these addresses is better defined, the IETF has placed some restrictions on them: anycast addresses cannot be used as a source address in an IPv6 packet and they should only be assigned to routers, not hosts.

The Subnet-Router anycast address is a specially predefined anycast address. It is formed on each subnet by padding the interface ID with zeros, i.e. it is the first address in a subnet block. This address is used in networks that have multiple routers. Packets sent to the subnet-router anycast address will be delivered to the "nearest" router on the subnet. All routers are required to have this address configured for each interface it has that is connected to a subnet. RFC 2526[41] defines the other reserved anycast addresses on a subnet, such as the Mobile IPv6 Home Agents address.

1.4.4 Multicast Addresses

The last type of IPv6 address defined by the IETF are multicast addresses. Packets sent to a multicast address are delivered to each interface that is identified by that address; i.e. "one-to-many"

Address Scope	**Permanent**	**Transient**
Reserved	FF00::/16	FF10::/16
Interface-Local	FF01::/16	FF11::/16
Link-Local	FF02::/16	FF12::/16
Site-Local	FF05::/16	FF15::/16
Organisation-Local	FF08::/16	FF18::/16
Global	FF0E::/16	FF1E::/16
Reserved	FF0F::/16	FF1F::/16

Table 1.4: Table of Multicast IPv6 Address Assignments

communication. IPv6 multicast operates in the same way as it does for IPv4, but it is a compulsory part of IPv6. Packets must not use a multicast address for its source address, nor can it be used in the Routing Header.

All IPv6 multicast addresses start with the "`FF00::/8`" prefix. The next four bits of the address are flags, which specify the group of the multicast address. The first three bits of these flags are reserved and must be set to zero. The remaining bit is toggled to differentiate between the basic groups: permanently assigned ("`0`") and transient addresses ("`1`"). The next four bits of the address define its scope. Table 1.4 contains the current multicast assignments.

The concept of scope is the same as that already defined for Unicast: link-local must be confined to use within a link; site-local, for use within a site; and global, for use on any network. Other scopes defined for multicast addresses are: interface-local (i.e. local loopback), admin-local (administratively defined and configured for use), and organisation-local (for use within a collection of subnets belonging to a single organisation).

There are a number of multicast groups defined for finding resources on the local node or link: the "all-nodes" group[3] ("`FF01::1`" and "`FF02::1`") and the "all-routers" group[4] ("`FF01::2`", "`FF02::2`" and "`FF05::2`"). Some of these permanently assigned addresses, such as the NTP servers group have variable scope associated with it, but others such as the DHCPv6 servers group are assigned with only one scope (site-local scope in this case).

[3]which sends a message to all the interfaces on a link.

[4]which which sends a message to all routers on the link.

Another permanently assigned multicast group is the "solicited-node" group. Addresses in this group are formed by taking the last 24-bits of the interface ID and inserting it after the prefix "`FF02::1:FFxx:xxxx`". A solicited-node address must be constructed for each unique interface ID, of unicast or anycast type, on a node.

1.4.5 Address Requirements

Each IPv6 host must have the following addresses configured for each of its interfaces. These addresses, except for the loopback address, are used extensively in the autoconfiguration of addresses by the Neighbour Discovery protocol:

- A link local address for each interface,
- The loopback address (`::1`),
- The link-local all-nodes address (`FF02::1`),
- The solicited node multicast address for each distinct interface ID,
- An address for each other multicast group it is supposed to join (e.g the DHCPv6 server group).

1.5 Internet Control Message Protocol version 6

There have been a number of changes made to the Internet Control Message Protocol (ICMP) so that it works with IPv6[11]. Firstly, the functionality of Internet Group Management Protocol (IGMP) and Address Resolution Protocol (ARP) have been incorporated into the new version of ICMP, known as ICMPv6; these functions are part of the Neighbour Discovery protocol that is described in section 1.7.

Changes have also been made to notify hosts of problems specific to the IPv6 protocol, such as unrecognised extension header types and header fields. The rest of the codes from ICMP have been retained: Echo Request, Echo Reply, Packet Too Big, Time Exceeded etc. The messages have been divided up into two types — "error messages" and "informative messages"— with the most significant bit of the code set to "`0`" and "`1`" respectively. Table 1.5 contains some of the more frequently used ICMPv6 messages.

ICMPv6 Type	**Subtype**	**Code**
Destination Unreachable		1
	No route (0)	
	Administratively Prohibited (1)	
	Address Unreachable (3)	
	Port Unreachable (4)	
Packet Too Big		2
Time Exceeded		3
	Hop Limit Exceeded (0)	
	Fragment Reassembly Time Exceeded (1)	
Parameter Problem		4
	Erroneous Header Field (0)	
	Unrecognised Next Header type (1)	
	Unrecognised IPv6 Option (2)	
Echo Request		128
Echo Reply		129
Router Solicitation		133
Router Advertisement		134
Neighbour Solicitation		135
Neighbour Advertisement		136
Redirect		137

Table 1.5: Table of ICMPv6 Messages

The use of ICMPv6 is essential to the correct operation of IPv6 and *must* be implemented by each node. In particular, as there is no fragmentation allowed by routers, “Packet Too Big” messages must be allowed in order to determine the correct size of the link so that packets are not silently discarded en-route.

1.6 Address Autoconfiguration

IPv6 addresses can either be assigned manually by an administrator or automatically by the machine itself. Probably the most useful feature of IPv6 is its autoconfiguration of addresses; both stateless[77] and stateful autoconfiguration. The latter occurs when there is a Dynamic Host Configuration Protocol (DHCPv6) server available that assigns an address to each newly connected machine on a network, and the former is when a new host “learns” its address from the local router.

As previously discussed, when a machine is configured with an IPv6 stack it creates an interface ID for itself from its Ethernet address and the EUI-64 constant. Using this unique identifier, the host automatically configures a link-local address for itself by adding the “`fe80::/10`” prefix. This link-local address is then used to contact other hosts and routers on the same link.

Once it has its link-local address, the interface uses the Neighbour Discovery protocol to check that this address is unique and also to configure a global unicast address if a network prefix is advertised by the router (described in 1.7.3).

1.7 Neighbour Discovery

In IPv4, address resolution between the Ethernet and IP addresses was achieved using the Address Resolution Protocol and ethernet-layer broadcast addresses, and IGMP was used to manage IPv4 multicasting. In IPv6, the functionality of IGMP has been incorporated into ICMPv6 and broadcast addresses have been replaced by multicast addresses.

Neighbour Discovery (ND)[60] protocol is used to perform address resolution between Ethernet and IPv6 addresses. It is comprised of a set of ICMPv6 messages, some of which are sent as multicast packets: Neighbour Solicitation and Advertisements, Router Advertisements and Solicitations, and

Redirect messages. ND is also used to configure link-local and global addresses on an interface, to obtain and advertise network parameters (such as the network prefix(es)), and to maintain router and host reachability information.

1.7.1 Duplicate Address Detection

ND is also used to establish the uniqueness of each interface on a subnet, so that address conflicts do not arise. When a host creates a link-local address for itself, it is not considered a permanent usable address until the Duplicate Address Detection (DAD) process has taken place.

DAD is instigated by sending an ICMPv6 Neighbour Solicitation (NS) message from the unspecified address "::" to the solicited-node multicast address corresponding to the interface ID; the autoconfigured address can not be used while it is still "tentative".

If, after a certain time period has elapsed and no response to the solicitation message has been received the host assumes that no one else is using the address and adopts it. If another node is configured with that address, it should reply with a Neighbour Advertisement (NA) message to the "all-nodes" multicast address (`ff02::1`). Then, the tentative address is marked as unusable and the administrator must intervene and generate another random ID, which can then be used as the interface identifier.

1.7.2 Neighbour Unreachability Detection

Another feature of the ND protocol is Neighbour Unreachability Detection (NUD). All IPv6 nodes keep a table of known neighbours, known as a "neighbour cache"; this replaces the IPv4 ARP table functionality. The main difference between the two is that neighbour advertisement and solicitation messages are only sent to the hosts that wish to receive such information and not every host on a link (as with messages sent to the broadcast address).

From time to time, hosts and routers will disappear from the local link for any number of reasons. NUD is used to check that a neighbour is still reachable via the address it has stored in its neighbour cache, by sending a NS message to the cached address. Neighbour unreachability may also be due to a change in the hardware of a host; i.e. the Ethernet card has changed and hence the MAC address for the interface has changed. ND is able to maintain host and router reachability in the event of a impromptu hardware change, by updating the neighbour cache with the new MAC address while maintaining the same link-local address. A Neighbour Advertisement message can also be sent by a host to all its neighbours on the link to inform them of a change in its address so that they can update their cache accordingly.

The ICMPv6 Redirect message is used to inform a host that there is a better way of routing a packet; i.e. there is a better first hop. This message is the same as its IPv4 counterpart, but unlike IPv4, IPv6 hosts know that the next hop is always "on-link" because of the link-local scope of the address delivering the redirect. The security implications of this are discussed further in section 3.2.

1.7.3 Network Prefix Assignment

The last part of ND is Network Prefix Assignment. This is achieved using the Router Solicitation (RS) and Router Advertisement (RA) messages. When an interface has successfully configured its unique link-local address, it can send out a RS message to the "all-routers" link-local multicast group ("`FF02::2`"). The router(s) on the link then respond to this message with a (unicast) RA message, which contains the global prefix(es) available for use by the interface.

Generally, it is not necessary for a host to send a RS query to determine this information as the router(s) on a link periodically sends out a Router Advertisement message announcing this to all nodes on the link. As well as providing the network prefix(es) available, these messages may also include additional information such as: the Maximum Transmission Unit (MTU) of the link, the default hop limit and gateway for the routing of packets (where there is more than one router on a link), and the length of time a host can be considered reachable (i.e. the length of time a cached address is considered valid). The RAs can also be configured to tell hosts to use stateless or stateful autoconfiguration (DHCPv6) and to statelessly renumber a network when there is a change in the network prefix.

Once a host has received this prefix announcement it can configure a global unicast address for itself, by appending its interface ID to the prefix. If there are multiple routers on the link advertising these parameters, the host chooses the most suitable one that will be used to forward packets for it. NUD and Redirect messages facilitate this decision.

1.8 Network Management

Although the drive for more IP addresses was the impetus for creating IPv6, it is in the area of network management that the full benefit of the new protocol can be seen. Features such as the automatic configuration of addresses on a subnet and the ease and speed of renumbering a large number of nodes automatically is a huge benefit to network administrators.

The hierarchal address allocation procedure means that networks have a contiguous block of addresses, which reduces the complications associated with routing packets and creating subnets. Also, the volume of subnets available in even the smallest allocation (a /48) provides 65536 possible subnets, which means that it is easier to plan networks with security as well as ease of operation in mind. For example, all important servers can be put on their own subnet with only limited access available to it. Hosts can be multihomed with multiple global unicast addresses per interface, so that if connectivity is lost via one provider, access is still possible via a secondary prefix and route.

This practice can also be extended to individual hosts, as they can have multiple addresses per interface that can be of any scope. A proposed security benefit that takes advantage of multiple addresses per interface is detailed in the Transient Addressing for Related Process section 3.6. It proposes a method of firewalling IPv6 services by giving each service its own IPv6 address. Access could also be restricted to services based on the scope of the address that was used to contact it.

Chapter 2

Stateless IPv4 Firewalls

2.1 Packet Filtering

Packet filtering is the process of controlling traffic on a machine or network by matching each packet, whether incoming or outgoing, against a predefined security policy (ruleset). Filtering is generally done on the network level by the router (or another dedicated device), but it can also be done in the kernel of most versions of operating systems via programs like IPFW, IP Filter and Netfilter. Packets can be filtered on a number of different criteria: source and destination addresses; source and destination port numbers; the network interface the packet is received on; the direction the packet is travelling (incoming or outgoing) and the protocol type (TCP, UDP or ICMP). It is called stateless packet filtering because the processing of each packet does not depend on the previous packets sent and received in the connection. In general, packet filtering firewalls inspect only the IP, and in some instances the TCP, header information.

There are different actions that can be taken by the firewall when it receives a packet, such as “drop”, “permit”, “reject” (and send an ICMP notification back) or “redirect” (forward on the packet somewhere else). Most packet filters also come with accounting capabilities, which allow you to log when a packet matches a rule. This allows the administrator to determine weak points in the network and improve them, or to see if the network is under attack.

Stateless packet filtering is a fast and effective method of dealing with individual packets, but only for simple operations[82]. Requests that require a much more detailed protocol knowledge or the ability to track information beyond headers are better suited to stateful packet filtering (which will be discussed further in Chapter 4) or proxying.

The greatest strength of packet filtering is the ability to provide a choke point which allows for detailed control over the network from a single location. This means that an administrator can concentrate all their efforts in securing the network in one location, instead of trying to tie-down several areas and running the risk of leaving large holes in the network which may then be exploited. To completely protect internal hosts, packet filtering should be used in conjunction with other techniques, such as proxying, bastion hosts, virus scanning and active patching of end hosts[8].

2.2 Filtering on Ports

There are a number of packet filtering techniques that are usually employed as an initial security policy. The most basic packet filtering technique is filtering based on the port number. TCP and UDP use port numbers from a 16-bit number range to identify the two ends of a connection. These port numbers are divided into different blocks for different purposes.

The ports from 1–1023 are known as reserved ("system") ports, because traditionally only privileged processes could assign them in a program. This was done to prevent normal users from setting up a service on the reserved port and acting as the legitimate service; e.g. a Simple Mail Transfer Protocol (SMTP) server collecting all the e-mail for a network instead of the real SMTP server. This theft of the source port can also occur if someone sets up a service on a port that is normally reserved for another service, in the hope of thwarting the packet filter. For example, SSH is sometimes run on the HTTP or HTTPS port, as these ports are almost always "open" to allow web traffic through, but the SSH port might be blocked.

Some operating systems such as OpenBSD and Solaris treat port 2049 (Network File System) as privileged too. This is to prevent users stealing this port by setting up an NFS server in place of the legitimate one. On Solaris, it is possible to add and remove privileged ports as they are required:

```
# ndd -set /dev/udp udp_extra_priv_ports_add 2049
```

Most common services have been assigned port numbers in the reserved range by the Internet Assigned Numbers Authority (IANA) to avoid this problem[40].

As previously mentioned, there are port numbers on both ends of a connection. The next range of port numbers are called ephemeral ports. These are the ports used by clients to make (short-term) connections to a server; they range from port 1024–4999 typically. Every time a client is run, a different port number from that range is assigned unless the client specifically requests a particular port number.

The last range of port numbers are from 5000–65535 and are used for other services, generally services that are not in wide use on the Internet, "dynamically" assigned or are unassigned. The "well-known" ports are usually found in the `/etc/services` file on Unix operating systems.

The ephemeral port range can be controlled on some operating systems[1]. The range of ephemeral ports set aside by IANA means that there is a limit of 3975 simultaneous connections to a particular port on a server.

Modern demands on systems meant that this amount proved to be insufficient, as these port numbers were being exhausted too quickly on busy servers. Mike Gleeson[26] suggested that it would be better to move the ephemeral port range to the high end of the ports range to 49152–65535. This meant that there would be a much larger block of ports available (16383) and because they are at the high end of the port range, it is less likely that there would be a system service listening on a port there.

This makes it easier to define a block of ephemeral ports for use in packet filtering rules to allow connections for services and programs such as File Transfer Protocol (FTP) and programs like `traceroute`.

On FreeBSD there are sysctl variables for the reserved ports and two blocks of ephemeral ports; the normal IANA default range and an alternative high range from 49152–65535. The high port range is specifically for client outbound connections which do not want to be filtered by the firewall. OpenBSD has two ephemeral port ranges which can be altered by sysctl variables; a low range from 1023–49151 and a high range from 49152–65535. Linux operating systems have one sysctl variable for

[1]On FreeBSD and OpenBSD this is via the `sysctl` commands: `net.inet.ip.portrange.first`, `net.inet.ip.portrange.last`, `net.inet.ip.portrange.hifirst` and `net.inet.ip.portrange.hilast`. On Linux it is the `net.ipv4.ip_local_port_range`

Services	**Port Number**	**TCP and UDP**
FTP	21	yes
SSH	22	yes
Telnet	23	yes
SMTP (mail)	25	yes
Domain	53	yes
HTTP	80	yes
POP3	110	yes
NTP	123	UDP only
HTTPS	443	yes
Microsoft SQL Server	1433	yes
Oracle	1525	yes
X11	6000	yes
Amanda	10080	UDP only

Table 2.1: Table of Services and corresponding Port Numbers

the ephemeral ports, which ranges from 32768–61000 by default.

To make firewalling a heterogenous network easier, it is better if the same ephemeral port range is specified on each of the different operating systems, so that they all coincide with the open range on the network firewall. Otherwise, a larger range of ports covering all client machines' ephemeral port ranges must be opened on the packet filter.

Table 2.1 contains examples of port numbers that are most frequently in use on Unix systems. Although these services usually run on the port numbers described above, this need not always be the case. It is common for people to run services on other ports to obfuscate the fact that they are running these services; so called "security through obscurity".

Typically, rules are defined to allow or disallow the various network services such as Domain Name Service (DNS), Secure Shell (SSH), Simple Mail Transfer Protocol (SMTP), Hyper Text Transfer Protocol (HTTP) etc. Whether you allow or disallow these various services will depend on the security policy that has been decided upon for the network. This defines how your network needs to be accessed, and how users on the network are allowed access the Internet (this is discussed further in section 2.10).

For example, the following rules allow connections from my machine to any other machine on the Internet for SSH, SMTP, HTTP and Secure HTTP (HTTPS). Specifying that connections from "myip" must be from a port in the ephemeral port range will block incoming connections that appear to be outgoing ssh connections.

```
add allow tcp from myip 49152-65535 to anyip ssh
add allow tcp from myip 49152-65535 to anyip smtp
add allow tcp from myip 49152-65535 to anyip http, https
```

Some services, such as DNS, need to have rules specified for both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), as it has different functions depending on which transport protocol it is using.

```
add allow tcp from myip 49152-65535 to anyip dns
add allow udp from myip 49152-65535 to anyip dns
```

Generally, filtering needs to be performed on both incoming and outgoing packets as most connections are bidirectional. That is, there is no point in allowing SSH packets out of your network, if the returning packets are not allowed back in.

```
add allow tcp from myip 49152-65535 to anyip ssh out
add allow tcp from anyip ssh to myip 49152-65535 in
```

A decision must also be made on whether to allow various service connections (SSH for example) to remote machines from local machines, as well as from remote machines in to a local machine running the service behind the packet filter.

```
add allow tcp from anyip 49152-65535 to myip ssh in
add allow tcp from myip ssh to anyip 49152-65535 out
add allow tcp from myip 49152-65535 to anyip ssh out
add allow tcp from anyip ssh to myip 49152-65535 in
```

It is useful to divide the ruleset up into rules for incoming connections to an interface and outgoing connections from an interface. The specific interface name (i.e. eth0, for the external interface) can usually be used as a filtering parameter to this end. Depending on the size and configuration of the network, there may also be packets arriving on multiple interfaces that need to be filtered.

```
add allow tcp from myip to anyip ssh out via eth0
add allow tcp from anyip ssh to myip in via eth0
```

2.2.1 Filtering FTP

Other services, such as File Transfer Protocol (FTP), need special attention when being filtered because of the way the protocol works. FTP uses two separate TCP connections to transfer a file: the command channel (for the FTP commands to be sent over on port 21) and a data channel (for the actual file, usually port 20 on the server side). Rules need to be defined for both channels so that all related packets are allowed through the firewall, but it is more complicated than that.

FTP has two methods for setting up the data channel, depending on whether the connection is being established from the server to the client, or vice-versa. For server-to-client connections (active mode FTP), the client listens on an arbitrary port and tells the server this via the PORT command. The server then opens a connection to this port on the client from its port 20.

For client-to-server (passive mode) connections, the client sends the PASV command to the server at the start of the connection to indicate that a different data channel should be used. A data channel is opened in the same direction as the original command channel connection. The client sends the port number it wants the server to connect to to send the data. The server then informs the client of the (ephemeral) port it is listening on, and the client connects to this port to establish the data channel.

If FTP is used in passive mode only outbound connections from the client need to be allowed through the firewall, which is generally the case anyway. With active FTP, incoming TCP connections need to be catered for in the filtering rules as well so that the data channel can be set up. This can lead to a number of problems because a large range of ports must be left open on the packet filter to allow these return packets. There is also the chance of someone pretending to be an FTP client, and then issuing the PORT command to request a data channel on a reserved port, in the hopes of

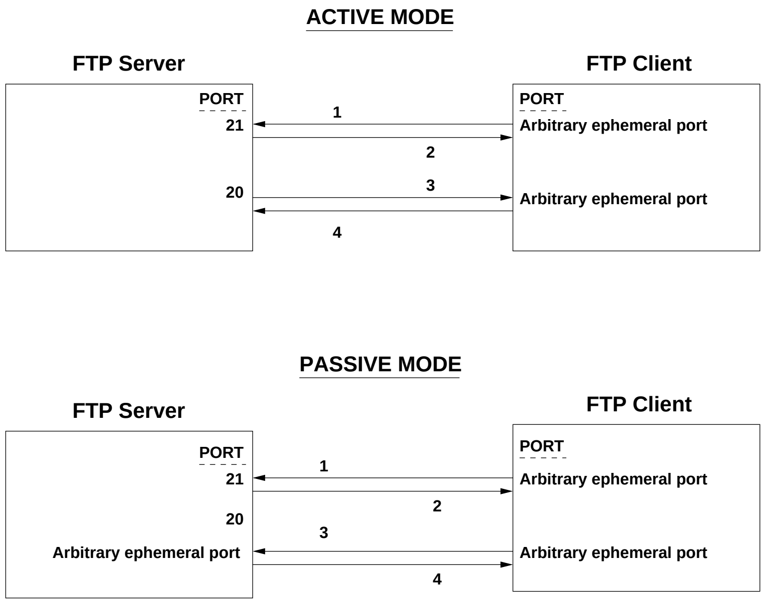

Figure 2.1: Active and Passive FTP

hijacking the connection. Newer versions of FTP have resolved this issue by not allowing requests to open data channels on reserved ports.

Administrators still generally only allow passive FTP through the firewall, if at all. There are more secure methods of file transfer available via programs such as Secure Copy (SCP) which uses a normal SSH connection to send the file to a remote system. This means that only port 22 needs to be opened on the firewall.

Rules to allow Passive FTP for incoming and outgoing packets are as follows:

```
/*Passive FTP from me to other places*/
add allow tcp from myip to anyip ftp out via eth0
add allow tcp from myip 49152-65534 to anyip 49152-65534 out via eth0
/*Passive FTP to me*/
add allow tcp from anyip to myip ftp in via eth0
add allow tcp from anyip 49152-65534 to myip 49152-65534 in via eth0
```

2.2.2 CERT Packet Filtering Recommendations

The Computer Emergency Response Team (CERT) identifies typical packet filtering techniques that they recommend are always implemented by default[78]. These are services that are frequently abused by intruders/crackers to gain access to hosts, as the protocols themselves are flawed.

CERT say that it is prudent to block all UDP traffic (except that which you specifically allow, such as Network Time Protocol (NTP) on port 123), as there is little useful everyday UDP traffic and it is frequently used as a way of attacking networks. Likewise, the Network File System (NFS) and Remote Procedure Call (RPC) on ports 111, 135 and 2049, both UDP and TCP, should be blocked. Certainly, these protocols are not secure by default, and there is usually no reason for their traffic to traverse network boundaries. Also, these ports have recently been abused by viruses and worms such as Blaster[48] and Sobig[73]. Networks that filtered out this traffic were not affected by these worms, and also helped to prevent their spread to other networks.

Other services which they recommend should be filtered as a matter of course are: Trivial File Transfer (TFTP) and Private Terminal Link (ttylink) (ports 69 and 87 on TCP and UDP respectively) which, again, are often used as a means of attacking and entering a network. Services such as direct remote X sessions (X11, TCP and UDP ports 6000+) and line printing (lpr, TCP and UDP port 515) and the ports that internal servers use that run on unprivileged ports (such as Oracle and MySQL databases) should also be filtered. These services should not need to be accessed remotely, but if it is necessary to remotely use Graphical User Interface (GUI) programs, X forwarding over SSH is a much better solution as only port 22 needs to be left opened.

Finally, they recommend that access to DNS on port 53 (TCP) should be filtered to block everyone but the secondary nameserver. As mentioned previously, DNS uses both TCP and UDP to function; UDP is used to perform domain look-ups, and TCP is used to do zone transfers. There are two types of domain servers: primary servers, which load their data locally from zone data files and secondary servers, which load their data remotely over the Internet via these zone transfers[1].

The premise for filtering TCP DNS requests is that it prevents zone transfers from happening, which helps to prevent crackers from gaining valuable knowledge about the other hosts on the network. In reality, it provides a false sense of security and makes it difficult for legitimate scanning (such as the RIPE hostcount[62], which compiles statistics on the number of registered domains in the RIPE IP address space) to take place. And with sufficient computational power, it is possible to do reverse

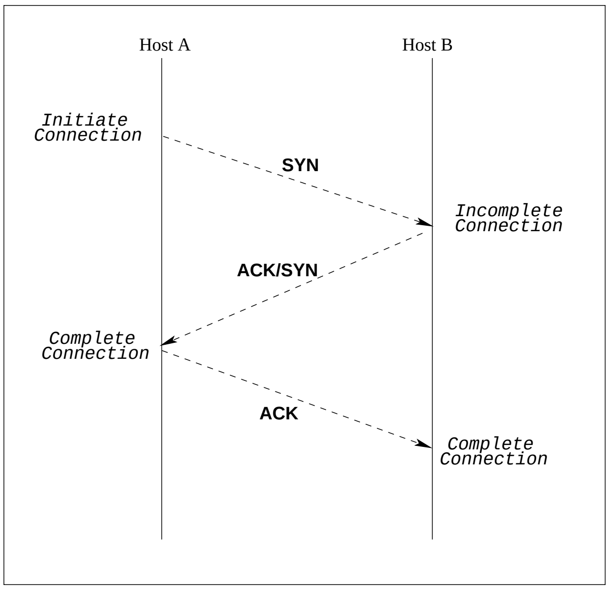

Figure 2.2: The TCP 3 way Handshake

lookups for all the addresses in a block to gather the information that blocking zone transfers is supposed to protect.

2.3 Filtering on TCP flags

TCP uses a "3-way handshake" (figure 2.2) to establish connections between two hosts. The 3-way handshake consists of three packets: SYN (the TCP flag set in a packet to a host, to request the opening of a connection), SYN/ACK, and ACK (which is an acknowledgement that the SYN packet was received by the other end of the connection).

These packets aim to establish a TCP "sequence number" at both ends of a connection. Sequence numbers are assigned to the data transmitted between the two hosts from a 32-bit range that starts with an arbitrary number called an Initial Sequence Number (ISN). ISNs are exchanged at the beginning of a connection so that conflicts do not arise between packets that get delayed, lost or duplicated in the network[74].

It is possible to create rules to filter packets based on these TCP flags, because the SYN bits are only set when a host is trying to "setup" a connection with another host, and so by controlling the SYN packets you can control the connection. Similarly, any packet sent that has the ACK bits set but not the SYN bits, is part of an "established" connection; i.e. it has already completed the 3-way handshake and sockets are listening for traffic at either end of the connection. Most stateless firewalls have "established" and "setup" keywords that can be used to create rules to filter traffic based on the presence of these flags. It is also possible to filter on the other TCP flags, such as Finish (FIN), Reset (RST) etc.

Filtering on the ACK flag means that rules that are specified to allow return packets back through the firewall do not get hijacked by new connections. For example:

```
add allow tcp from myip ssh to anyip 49152-65535 out
add allow tcp from anyip 49152-65535 to myip ssh tcpflags ack in
```

When rules are defined that use the established keyword, we allow all established packets through the filter and hope that they belong to connections that have already completed the setup stage. If established packets arrive that are not associated with an existing connection they should be rejected by the end host to which they are destined.

To control which connections are allowed "setup", these packets are filtered more carefully than established packets by only allowing through the firewall connections to services that the administrator wants (and thus has rules specified for). Filtering the packets that only have the SYN bit set is useful in blocking attempts to make incoming connections via rules that are specified for outgoing connections; that is, where an external host tries to connect to a high ephemeral port from port 80 (or some other privileged port) on its machine, instead of the normal connection from your local machine to port 80 on the remote machine. If the SYN flag is not explicitly specified in the rule, it is perfectly legitimate for connections to be initiated from a privileged port on a remote machine to an ephemeral port on your local machine, however it is probably not what was intended when the rule was defined[2].

```
add allow tcp from myip 49152-65535 to anyip 80
add allow tcp from myip 49152-65535 to anyip 80 tcpflags syn
```

A ruleset might also block illegal combinations of TCP flags, for example SYN/RST/ACK. This

[2]Using the `in/out` keywords described earlier will also achieve this

could be used to protect end-hosts with poor TCP stacks from confusion, or to foil attempts at OS fingerprinting (see section 2.8 for more detail).

Many rulesets work on a “first match” basis — the rules are processed sequentially for each packet until a matching rule is found. A rule allowing established traffic through the firewall is usually put at the top of the ruleset. Most of the traffic going through the packet filter will be established already and this saves valuable computation time searching through the ruleset looking for a match.

2.4 Filtering on Addresses

So far we have discussed filtering on ports and flags in the TCP and UDP header. The next main method of packet filtering is done by filtering based on the source and/or the destination address. Packet filtering on addresses is mostly used to detect and reject spoofed packets on the Internet from entering the network.

Filtering packets based on the source and/or destination address is also done to allow unrestricted external access to a particular machine on an internal network, regardless of the protocols they use to do this. Obviously, if done carelessly, this leads to a very large hole in the firewall and is ill-advised, as it is not possible to tell that the external host is who it is claiming to be; either because of a source address forgery or a “man-in-the-middle” attack. Both attacks occur when a (external) host pretends to be another host that you trust. In the case of the source address forgery attack the host does not care about the return packets.

Another very common type of address based filtering is to permit connections only from certain trusted remote hosts. This can be just as problematic, because packets might be spoofed claiming to come from the remote host to circumvent the packet filter. It is also a bad idea to implicitly trust any remote host that is not under your own administration, because there is no way of knowing if the host has been cracked or not; thus, giving the attacker free reign on your network too.

When the destination port is also specified in the rule, address filtering can be useful for blocking general access to specific servers such as the domain name server and the mail server. In the case of the mail server: if there is a rule specifying that all mail on the network *must* be delivered to the mail server, this eliminates the problem of open-relaying of mail.

```
add allow tcp from anyip to mailserverip 25 in via eth0[3]
add deny tcp from anyip to anyip 25 in via eth0
```

Rules should not usually be specified with hostnames in them. There is a potential risk of leaving the host open to attacks during the booting process, as the machine will have to wait for the DNS and network interface to come up before it will be able to resolve these hostnames to their IP addresses. In the meantime, there is an incomplete packet filtering ruleset protecting the host or network. It is also important to only have IP addresses and netmasks in the ruleset because the firewall may be circumvented if the DNS cache becomes poisoned[72]. However, creating variable names that correspond to certain IP addresses makes it much easier to configure the packet filter and only one instance of this address will have to be updated if for some reason the IP address of the server needs to be changed.

Finally, all local loopback network traffic should always be allowed when it comes via the loopback interface, lo0, and denied otherwise.

```
add allow ip from anyip to anyip via lo0
add deny ip from 127.0.0.1 to any
```

2.4.1 Ingress and Egress Filtering

Many Denial-of-Service (DoS) attacks rely on packets whose source addresses have been spoofed. Spoofed IP addresses have also been used in attacks against services that use IP addresses for authentication, e.g. UNIX rsh/rhosts or services restricted to intranets. Ingress filtering[20] is specifically used to target these spoofed packets. When packets arrive from the outside but appear to have originated from an address local to your network, they are detected by the filter and dropped at the interface. Ingress filtering should always be performed on the "edge" of a network, or by the ISP; preferably at the edge as we do not want to have to rely on the ISP for our security.

Ingress filtering is performed to deny packets with an invalid address access to your network. This can include any of the following types of addresses:

[3]eth0 is the external interface

- Inappropriate Private Addresses[4] (as source or destination addresses)
- Broadcast Addresses (as source addresses)
- Inappropriate Broadcast Addresses[5] (as destination addresses)
- Local Loopback Addresses (on non-loopback interfaces)
- Multicast Addresses (as source addresses)
- Inappropriate Multicast Addresses (as destination addresses)
- Un-Allocated Addresses[6]
- Source Addresses that belong to the subnet the packet is destined for (arriving on an outside interface).

There is no genuine reason why packets with a source address that is a private address, unallocated address, broadcast address or multicast address should be arriving at the edge of the network; or leaving your network for that matter whether accidently or not. Even if multicast is in use on the network, there is no valid reason why a packet would have a multicast address as the source address, and so it is necessary to filter these addresses to reduce the possibility of DoS attacks. This is specified in RFC 1112[15]:

> *"A host group address (multicast) must never be placed in the source address field or anywhere in a source route or record route option of an outgoing IP datagram.*
>
> *An incoming datagram with an IP host group address in its source address field is quietly discarded."*

[4]RFC1918: Address Allocation for Private Internets

[5]Such as Directed Broadcast address, which was used in Smurf DoS attacks (`http://www.cert.org/advisories/CA-1998-01.html`). These attacks were instigated by a host sending ICMP echo requests from a spoofed source address to a directed broadcast address that caused all the hosts on the link to respond to the ping, thus filling up the network with unwanted traffic.

[6]Un-Allocated Addresses are the addresses reserved by IANA that have yet to be allocated to the various Regional Internet Registries (RIRs) and so should not be in general use. A complete list of the assigned IPv4 address space can be found on IANAs webpage[37]. If you block these addresses the list must be kept up to date, because they may be allocated for use in the future.

Another type of filtering that is usually performed by network administrators is Egress filtering. It is used to make sure that only packets with a valid IP address leave the network; that is, packets whose address belongs to the local subnet of addresses. This does not protect the network from attack per say, but these packets can be indicative of network misconfiguration or a malicious internal user. This is extremely important for ISPs as it stops their subscribers from launching hard to trace DoS attacks using spoofed addresses thus helping to prevent the further propagation of these attacks on the Internet.

These Ingress and Egress "anti-spoofing" rules are usually implemented as:

```
/*Perform Ingress filtering for anti-spoofing*/
add deny all from mysubnet/prefix to anyip in via eth0
/*Perform Egress filtering for anti-spoofing*/
add deny all from anyip to mysubnet/prefix out via eth0
add allow all from mysubnet/prefix to anyip out via eth0
add deny all from anyip to anyip out via eth0
```

Sample rules for filtering out packets with the invalid addresses described above are generally implemented as:

```
/*Local Loopback, v4 Multicast, Broadcast*/
add deny all from 127.0.0.0/8 to anyip out via eth0
add deny all from 224.0.0.0/3 to anyip out via eth0[36]
add deny all from 255.0.0.0/8 to anyip out via eth0
add deny all from 127.0.0.0/8 to mynet in via eth0
add deny all from 224.0.0.0/3 to mynet in via eth0
add deny all from 255.0.0.0/8 to mynet in via eth0

/*Private Addresses should not enter or leave the network*/
add deny all from 10.0.0.0/8 to anyip out via eth0
add deny all from 192.168.0.0/16 to anyip out via eth0
add deny all from 172.16.0.0/12 to anyip out via eth0
add deny all from 10.0.0.0/8 to mynet in via eth0
add deny all from 192.168.0.0/16 to mynet in via eth0
add deny all from 172.16.0.0/12 to mynet in via eth0
```

2.4.2 Unicast Reverse Path Forwarding

Unicast Reverse Path Forwarding (URPF) is a feature of routing software (such as Cisco's IOS[2]and Juniper's JunOS[43]), which has been designed to alleviate the many problems that can be caused when spoofed IP addresses are passed through a router; as is common with denial-of-service (DoS) attacks.

Certain source addresses can be identified as "spoofed" by Unicast RPF and when this happens the packets from these addresses are dropped at the edge of the network; URPF only forwards packets to the network if they have an address that is verifiably consistent with the router's view of the network. Many of the common DoS attacks rely on a constantly changing source IP address, to hold out longer against efforts to filter the attack from the network. Static Ingress filtering, as described in the previous section, is unable to deal with attacks from hosts that randomise or specially choose their spoofed source address.

Unicast RPF uses the routing table to check to see that the interface that the packet is received on matches the interface that would be used if a packet was to be sent from the local network to the packet's source address. If these interfaces do not match, or if there is no corresponding route to the source address in the routing table, the packet is discarded and it never reaches its destination.

Thus, when Unicast RPF is configured on a router, the *routing table* is consulted for every incoming packet to the network. The routing table contains the set of routes, known to the router (see figure 2.3), that allows it to forward on a packet to its destination. These routes can either be static or dynamic; depending on whether the administrator manually enters them into the routing table, or if the router "learns" the routes from another router via a routing protocol such as Border Gateway Protocol (BGP) or Open Shortest Path First (OSPF). This makes URPF act like a dynamic ingress filter.

The routing protocols may know of multiple routes to a given address. In this case, the route that is considered "best" by the routing protocols will be used by URPF. If there are multiple "best" routes, then URPF considers all of these valid. That is, where your network has multiple paths back

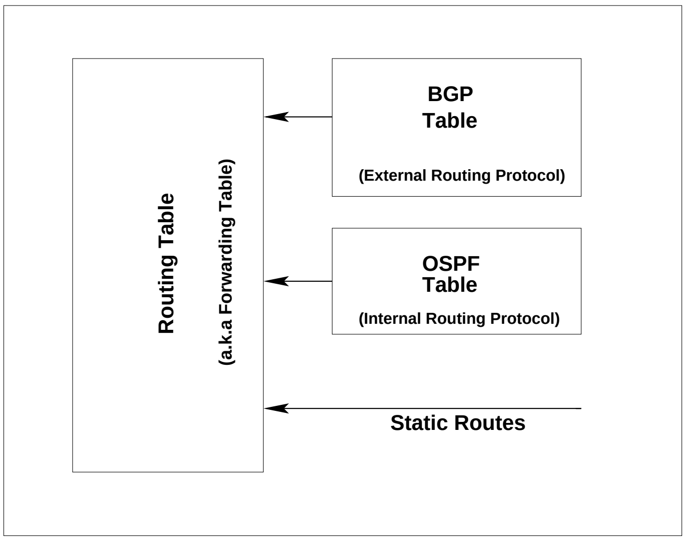

Figure 2.3: The Routing Table

to the source of the packet, if each of them is of equal-cost (in routing terms: same number of hops, weighting etc.) the packet will be considered valid and will be forwarded on to its final destination.

Unicast RPF and Ingress/Egress Filtering

Unicast RPF is more effective if it is combined with Ingress and Egress filtering at the interface; this is configured using packet filtering, either Access Control Lists (Cisco) or Fail Filters (Juniper). Unicast RPF only examines the legitimacy of the source address of a packet, whereas Ingress filtering can examine the legitimacy of both addresses of a packet. When combined it is more effective at rejecting all "spoofed" packets that try to enter the network.

"Fail-Filters" can also be used in conjunction with Unicast RPF to allow for a post-URPF check on packets. This prevents the dropping of legitimate packets bound for the network, when Unicast RPF is implemented in a situation where there is the possibility of multiple best paths that are not of equal cost[10] (i.e. asymmetric routes: where packets come in one way to a network, but leave via another). In this case, where a filter is implemented on an interface, Unicast RPF checks the validity of the packet as normal. If and only if the packet fails the check, it is then passed through the packet filter and checked to see if a rule exists that states whether the packet should be dropped or forwarded on.

This is to allow, primarily, for exceptions and as a by-pass method to allow certain blocks of network addresses to pass the Unicast RPF check. It also allows for logging of the packets, and thus is an aid in detecting the actual source of the "spoofed" packets (the counters will be incremented when the packets start dropping, which can alert the administrator to the problem), or if there is a misconfiguration. This ability to log when URPF rejects a packet, allows the administrator to gather information on a router to determine the source of a potential attack; or in the event of an attack taking place, information about the attack such as the time it happened, the duration and the possible source of the attack. The logging feature can also be used to debug the Access Control List to make sure that there are no missing rules or misconfigured routes on the router.

Strict Mode and Loose Mode

On Cisco and Juniper routers, Unicast RPF is setup in "strict" mode by default; the incoming packets at an interface are checked for a valid source address that has a corresponding route in the routing table *and* it checks that the interface expects to receive packets from this route at this interface. If the packet fails the Unicast RPF check it is rejected from the interface, the RPF counter is incremented, and the packet is passed on to an optional Fail Filter or Access Control List.

In loose mode, the only check that is performed on a packet is whether the packet's source address comes from a verifiable route; that is, it has a corresponding prefix in the routing table. If the source of the packet can not be verified, the packet is dropped from the interface. Once again there is the option of passing it to a Fail Filter/ACL to either accept, deny, or log the packet.

In certain circumstances it is advantageous to use Unicast RPF in loose mode: an ISP that has multiple interfaces with connections to the Internet would not want to have to resort to Fail Filtering the majority of their network traffic in order to avoid losing any legitimate packets, as this would put undue pressure on the running of the router. Loose mode still allows for partial DoS protection, as packets that do not have a matching prefix in the routing table will still be dropped.

Advantages of using Unicast RPF

There are a number of advantages to using Unicast RPF over just using Access Control Lists/Fail Filters on a router. Routers are typically quick at looking up routes in the routing table, but slow at firewalling; the route lookups are done using customised routing hardware, while packet filtering (performed by software) is limited by the sluggish CPUs often found in routers.

URPF operates at the router's packet forwarding speed, so it is much faster at filtering the packets. And, as URPF is done using the customised forwarding hardware, it has a minimal CPU overhead so it can be used effectively on less "beefy" routers.

URPF also dynamically adapts to changes in the routing table, including static routes, so there is less administration required for configuration when aspects of the network change, as opposed to Access Control Lists which must be updated manually.

As with all filtering and logging, it is very intensive in the use of the CPU and memory of the router, and will eventually degrade the performance of the router. It is therefore sometimes recommended to only use Unicast RPF with filtering during an attack; or at scheduled intervals where the router is not being heavily taxed; or if the router has a decent CPU.

Problems with Unicast RPF

The main problem that occurs when using Unicast RPF is when it is used on multihomed networks[7]: where multiple paths exist as a return path for a source address, but they are not necessarily of equal cost. It is frequently touted that this problem is caused by asymmetrical routing (many connections to the Internet are asymmetric), but the real problem for current implementations appears to be due to the way that the "best path" is selected[67]; it is only able to select *one* best path. This "best path" selection characteristic of the routing tables is the cause of these asymmetric routes, and the problems generated when legitimate traffic gets dropped by Unicast RPF.

As we have seen, when using Unicast RPF in a situation where a host is multihomed (see figure 2.4) — either to separate routers in one ISP, or to multiple ISPs — you must ensure that packets travelling up the link to the Internet match the route that is advertised out that link. Otherwise Unicast RPF will filter these packets as spoofed packets and they will be dropped at the interface.

[7]Multihoming is the method by which a network can use two discrete Internet connections provided by separate Internet Service Providers (ISPs). This means that the network has two or more direct connections to the Internet Backbone and so if one of the providers is experiencing difficulty, there is no loss of connection of the network to the Internet. Multihoming is also used improve the performance of a network, as it allows hosts to send packets via the most direct route through the Internet to deliver the packets.

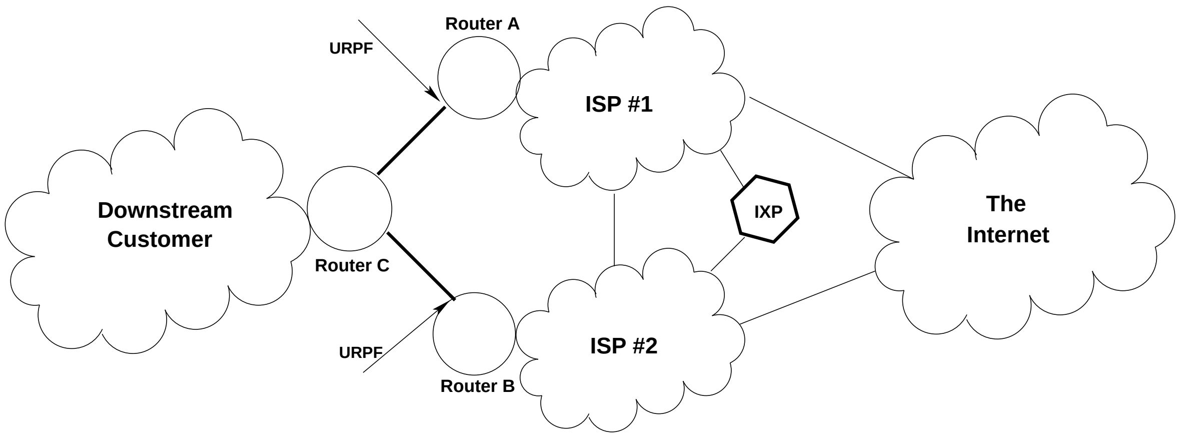

Figure 2.4: Multihomed Networks and Unicast Reverse Path Forwarding

The best way to prevent this is to advertise all of the public route prefixes for the network out each of the links.

Unicast RPF acts at the routing stage and inspects the source/destination address used for routing. Thus, it cannot inspect packets that are encapsulated in tunnels (Generic Route Encapsulated Tunnels (GRE), Layer Two Tunnel Protocol (L2TP) etc.). In this instance, Unicast RPF must be configured at the encapsulating/decapsulating router so that the encapsulation and tunnelling layers of the packet header are removed. Unicast RPF can then proceed to process the packet as normal.

Certain implementations of URPF behave differently if they are enabled on a router that has a default route. The expected behaviour in "loose mode" is to accept all packets when there is also a default route (because the interface is not checked in loose mode). Some implementations have modified Unicast RPF's behaviour in order to make it more useful in this case. However, the main effect seems to have been to cause confusion among network operators.

Unicast Reverse Path Forwarding was originally developed by Cisco and Juniper for their routers. This functionality has since been incorporated into the packet filters available on Unix Operating Systems such as IPFW (the `verrevpath` option), Packet Filter (can specify "loose mode" URPF with the `no-route` option[8]) and "Reverse Path Filtering"[55] can be specified in the Linux kernel by doing the following:

[8]`# block in from no-route to any`

```
# for i in /proc/sys/net/ipv4/conf/*/rp_filter ; do
> echo 2 > $i
> done
```

As URPF is implemented at the kernel level on Unix systems it is possible to do more flexible lookups to filter packets, than those currently done on proprietary routers. These implementations of URPF are not limited to just using the routing table to do the reverse lookups; they could also use the BGP or other routing protocol tables instead; thus potentially avoiding the problem of only being able to check the "best" route.

2.5 Packet Fragmentation

If a packet is too large to be sent on a link in one unit the router should split it up into smaller parts, each of which contain enough information for the receiving host to piece the fragments back together again. Simple packet filters will drop fragments because the headers do not contain enough higher level information (such as TCP headers) for them to implement filtering on.

In order for these fragmented packets to be filtered correctly, the filtering is usually done on the first packet which contains all the header information. An "allow all" rule is then specified to pass the rest of the fragments of the packet through the firewall, like:

```
add allow tcp from anyip to anyip fragmented
```

In some cases the firewall may piece the fragments back together before sending on the whole packet to the destination host; this is to prevent attacks on machines which may react badly to overlapping fragments, or fragments that are missing the first fragment of the packet[33].

If a fragmented packet arrives at an interface that is the first fragment of a packet and it does not contain enough bytes for a complete header (20-bytes for TCP, 8-bytes for UDP and 4-bytes for ICMP), it may be silently dropped by the packet filter. Another special circumstance which may arise where packets are unconditionally dropped, is when a packet arrives with a fragment offset of "1". Although these are legitimate packets, their only purpose is to try to circumvent firewalls so they are dropped.

In general, it is considered bad practice to fragment packets because if any one fragment gets dropped or lost en-route then the entire packet must be retransmitted[44]. It also imposes an extra load on the routers that are performing the actual fragmenting of the packet. Path MTU Discovery[16] was devised as a method for establishing the largest packet that can be sent on a link, without the need to fragment it.

2.5.1 Path MTU Discovery

The Maximum Transmission Unit (MTU) is the largest number of bytes of data that can be sent on the link layer in a single transmission, i.e. in a packet. The Path MTU is the lowest MTU of any of the hops in the path between two hosts. Path MTU Discovery works by setting the "Don't Fragment"(DF) bit of the IP header on a large packet and sending it between two hosts. If any MTU between the hosts is smaller than the packet sent, the router then sees that the DF bit is set and instead of fragmenting the packet it sends back an ICMP "Can't Fragment" error message. The sending host can then reduce the size of the packet to a suitable MTU and resend it.

Network administrators have taken to filtering all ICMP messages because they were abused in DoS attacks in the late 1990s. If the ICMP "Can't Fragment" messages (type 3) cannot make it back to the sending host, because these messages are filtered at some point en-route, then the host will never know that the packets it is sending are too big. It will continue to try to retransmit the packet and it will keep being silently dropped by the router whose MTU is smaller than the packet being sent.

The practice of incorrectly filtering these critical ICMP messages has broken Path MTU discovery on the IPv4 Internet. As a result of this, there now exist routers which fragment packets even though the DF bit is set, as a means of working around ICMP filtering.

Another method that is sometimes used to prevent the fragmentation of large packets is TCP Maximum Segment Size (MSS) rewriting. The MSS option of TCP is used to identify the largest unit of data that TCP will send to the other end of a connection. MSS rewriting is used to manipulate the transfer by reducing the amount of data sent in a packet, so that the entire packet is less that the MTU. Obviously, this is of little use for large UDP packets.

2.6 Filtering ICMP Messages

Internet Control Message Protocol (ICMP) is used to determine the status of IP networks. ICMP is a fundamental part of the workings of the Internet, and so it cannot be filtered without careful consideration of the effects of filtering these messages. As previously mentioned, ICMP "Don't Fragment" messages should always be allowed through the packet filter so that Path MTU discovery can function correctly.

```
add allow icmp from any to any icmptype don't_fragment
```

Other ICMP messages that need special attention when filtering are: Echo Request (message type 8), Echo Replies (message type 0) and Time Exceeded (type 11) notifications. These messages are used in programs such as `ping` and `traceroute`.

The `ping` program uses timed ICMP "Echo Request" and "Echo Reply" packets to probe a remote machine to test if the machine is reachable or not. To allow pings through the packet filter, the following is needed:

```
add allow icmp from any to any icmptype echo_request, echo_reply
```

The `traceroute` program is used to determine the path a packet takes through the Internet to reach its destination; i.e. the number of "hops" it takes. UDP packets are sent as probes to a high ephemeral port (usually in the range 33434–33525) with the Time-to-Live (TTL) field in the IP header increasing by one until the end host is reached. The originating host listens for "Time Exceeded" ICMP responses from each of the routers/hosts en-route. It knows that the packet's destination has been reached when it receives a "Port Unreachable" ICMP message back. We expect a "Port Unreachable" message from the destination because no service should be listening for connections in this high port range. Occasionally, the maximum number of hops (specified by the TTL field) is exceeded before the "Port Unreachable" message is received. The maximum TTL defaults to 64 hops, but this can be changed using the sysctl variable `net.inet.ip.ttl` on FreeBSD and `net/ipv4/ip_default_ttl` on Linux.

Thus for the `traceroute` program to work correctly through a packet filter, rules must be specified for UDP connections in both directions and ICMP "Time Exceeded" messages. Traceroute can also be modified to use the TCP or ICMP protocol to send the probes. Rules which will permit traceroutes are:

```
/*Traceroute from me*/
add allow tcp from myip to anyip 33434-33525 out via eth0
add allow udp from myip to anyip 33434-33525 out via eth0
/*Permitted replies to my traceroutes*/
add allow tcp from myip 33434-33525 to anyip out via eth0
/*Traceroute to me*/
add allow tcp from anyip to myip 33434-33525 in via eth0
add allow udp from anyip to myip 33434-33525 in via eth0
/*Replies to traceroute to me */
add allow tcp from anyip 33434-33525 to myip in via eth0
add allow udp from anyip 33434-33525 to myip in via eth0
```

and also:

```
add allow icmp from any to any icmptype time_exceeded, port_unreachable[9]
```

ICMP Redirect messages are generated by a router (gateway) when a shorter route to a destination exists. This serves to inform the originating host that there is a shorter route to that destination and any additional traffic should be sent there directly[57].

The IP protocol specifies little sanity checking of redirect messages so an attacker can easily spoof these redirect messages and poison the routing table on a host. This host would then be open to a number of attacks: DoS attacks — because the new "router" may not be a router at all; packet sniffing; and "man-in-the-middle" attacks. ICMP Redirect messages should be filtered on networks where they have no legitimate use; e.g. on networks that only have one gateway as there is only one legitimate route out of the network. Also, these redirect messages should only be obeyed by hosts on the network, never the gateways themselves, and only when the messages come from a router on a directly attached network.

[9]or host_unreachable

2.7 IP Options

The Options field is an extra part of the IP header, which varies in its length and contains additional information about the packet that usually is not necessary in normal communications[63]. This optional header information can be used for security (such as is used by the United States Department of Defence to define security clearance levels), time-stamping the packet and special routing capabilities.

In general, these options are rarely used and not all hosts and routers support them. Source Routing is one option that is still somewhat in use, though rarely for legitimate reasons. IP packets usually specify a source and a destination address in each packet, but make no decision on *how* the packet should get from the source to the destination. This decision is made by the routers that the packet passes through en-route by their routing software. There are two forms of Source Routing: Loose and Strict mode. Strict Source routing is almost never used. It specifies the *exact* route a packet must take to a destination. The more common form, Loose Source Routing, specifies one or more hops that a packet must pass through to get to the destination.

Loose source routing can be used to send packets to a host from a spoofed IP address that pass through the attackers machine. RFC 1122[5] states that the destination host *must* respond to the packet using the inverse of the route specified in the routing header, so the return packets will go to the attackers machine instead of directly to the spoofed source of the packet [[8], pg 29].

Loose source routing has also been used in “sequence-number guessing” attacks, where an attacking machine sends spoofed packets with the source address of a trusted host to the machine under attack. The return packets are sent back via the attacker’s machine so they can see the sequence numbers of a legitimate connections and use this to impersonate the trusted host. This attack is not as prevalent as it used to be, as most operating systems implement randomised sequence numbers and the filtering of Loose Source Routing has become more common place.

One legitimate use of Source Routing is in connection with `ping` and `traceroute`, which administrators use to diagnose network problems. Even where this functionality is required on a network, and generally only ISPs tend to use it, it should not need to pass the network boundary. If loose source routing is not required on the network, packets that arrive with this option set should be blocked at the edge routers of the network to prevent source address spoofing and connection hijacking attempts.

2.8 Operating System Fingerprinting

OS fingerprinting is the practice of probing the TCP/IP stack (and applications) on a remote system to determine the operating system it is running. Knowing the OS and its precise version is a valuable asset to a cracker, as many security exploits are dependent on this information[23]. Of course OS fingerprinting is also a valuable asset to system administrators as they can remotely scan their network for vulnerabilities, so that they can target the systems that need patching or updating before these vulnerabilities can be exploited by someone else.

With some systems, it is easy to find out this information via programs such as `telnet` and `ftp` as they print a "banner". Certain Unix implementations of these programs display a banner when a user logs in remotely by default that contains the operating system's name and version number. Most system administrators neglect to turn off this banner when they install the machine. The following example shows that the remote system Carbon is running Debian Linux version 3, the "Woody" release. We also know the version of OpenSSH the server is running.

```
orly@oscar $ telnet carbon.redbrick.dcu.ie 22
Trying 136.206.15.1
Connected to carbon.redbrick.dcu.ie.
Escape character is '^]'
SSH-2.0-OpenSSH_3.4p1 Debian 1:3.4p1-1.woody.3
```

If the host is remotely accessible via FTP, then it may also be possible to find this information by issuing the `SYST` command. In the following example we see that the host is running a BSD kernel based system.

```
orly@oscar $ telnet ftp.kame.net 21
Trying 203.178.141.194...
Connected to ftp.kame.net.
Escape character is '^]'.
220 orange.kame.net FTP server (Version 6.00LS) ready.
USER anonymous
331 Guest login ok, send your email address as password.
PASS orly@dmz.ie
```

```
230- note that all activities are logged.
230- number of user is limited to %M at a time.  (you are user #%N)
230 Guest login ok, access restrictions apply.
SYST
215 UNIX Type:  L8 Version:  BSD-199506
Quit 221 Goodbye.
```

The `nmap` program provides another technique for remote OS fingerprinting. It works by probing the remote host with "unusual" TCP (and sometimes UDP) packets, monitoring their response to these packets and matching the output against its extensive database of OS "fingerprints". Differences in the implementation of the TCP and IP protocols on operating systems allow for this fingerprinting. Combining enough of these probes and their responses makes it possible to identify the exact version of the OS in most cases. Appendix A is the output of the nmap program which was run over IPv4, to determine the exact OS of `ftp.kame.net`.

2.8.1 Port Zero OS fingerprinting

The port number "0" is reserved for special use by IANA. When a program specifies a source port of zero, the operating system automatically assigns an arbitrary ephemeral port for use instead.

No legitimate traffic should ever be received from a remote host with a source or destination port of 0. But, because the specifics of how a host should deal with these connections is unclear different OSes have different responses and thus this can be used to fingerprint. If these fingerprinting attempts are not wanted, a rule can be specified in the packet filter to block incoming connections on the external interface (eth0) to port 0:

```
add deny all tcp from any 0 to any via eth0
add deny all udp from any 0 to any via eth0
add deny all tcp from any to any 0 via eth0
add deny all udp from any to any 0 via eth0
```

2.8.2 Protocol Scrubbing

Protocol Scrubbing is the process of packet normalisation for the TCP and IP protocols. It removes the inconsistencies that occur in the different implementations of these protocols, which allow for effective fingerprinting of remote systems and also normalises TCP traffic to aid Network Intrusion Detection (NID) systems to process traffic without any ambiguity. A TCP and IP scrubber was designed by David Watson, Matthew Smart, G. Robert Malan et al. to implement this[80].

> *"Protocol Scrubbers are transparent interposed mechanisms for explicitly removing network scans and attacks at various protocol layers"*

Their TCP scrubber modifies traffic, in real-time, at the edge of a network to combat insertion and evasion attacks, which deteriorate the effectiveness of the NID system. Insertion attacks occur when the NID accepts a packet that the end host refuses and evasion attacks occur when the NID rejects a packet that the end host accepts. This confusion can be caused by the differences in the way the protocol stack reassembles TCP byte sequences that are out of order and the way illegal TCP flag combinations are handled. Modifying the packets at the edge of the router so that they behave consistently for the NID and the end host prevents these attacks.

Their IP scrubber modifies traffic flows in real-time to block OS fingerprinting scans. The IP level ambiguities in protocol implementation, such as IP fragment reassembly and IP Type-of-Service flags can be used to determine the operating system of a remote host (via programs like `nmap`).

As these scrubbers are used at the edge of the network they are most useful when combined with a packet filter to help prevent these attacks. OpenBSD's Packet Filter also has the option to implement packet scrubbing on all packets. This is specified via the "scrub" option:

```
scrub in all
```

Other features of Packet Filter are described in section 2.11.4

2.9 Tunnelling

Tunnelling is the process of sending a packet of one protocol type encapsulated in another, using the network of the second protocol to forward the packet to its destination. When it arrives at the other end of the connection, the encapsulation header is removed and the original packet is re-injected into the network.

Tunnelling is used in many different scenarios on the Internet. It has been used as a transition mechanism for the migration to IPv6 (discussed further in section 3.4.2), where an IPv4 header prepends the IPv6 packet and then the packet is sent over the current IPv4 Internet to its destination. When it reaches its destination, the encapsulated IPv4 header is removed and the packet becomes solely an IPv6 packet again.

IPv4 can also be encapsulated in itself, directly and indirectly, and it is in this form that it is often used to by-pass firewalls. SSH and HTTP tunnels are frequently used to pass traffic that is otherwise blocked by the firewall, such as peer-to-peer traffic, Real Audio feeds etc. If a firewall permits user packets to be sent, then a tunnel can be set up to another host which is not behind a firewall, in order to circumvent the firewall local to the network.

On the other hand, tunnelling is a useful method for sending sensitive data across a public network like the Internet. IPsec provides this functionality via its network-layer encryption mechanism: Encapsulated Security Payload (ESP). IPsec can work by tunnelling; taking the packet that needs to be protected and encrypting it. Then a new IP header is attached, which may differ from the original header. Typically, a gateway source and destination address will be used instead of the original source and destination address as this provides protection from traffic analysis as well as packet sniffing.

It is difficult to filter traffic correctly that is tunnelled, either using a tunnel protocol (like L2TP) or through another protocol such as SSH, as the real packet headers are hidden from the firewall. Traffic that is tunnelled using a tunnelling protocol should either be de-encapsulated at the firewall and the packet processed as normal, or it should be dropped by the filter on the assumption that it is probably being used to obfuscate traffic that would normally be blocked by the packet filter.

Filtering traffic that is tunnelled through SSH or HTTP or some other widely used protocol is much more difficult and cannot be performed by a simple stateless packet filter, as the content of the data needs to be examined in order to determine if it really is SSH or HTTP traffic. Again, this job is better suited to proxying. Obviously, if the packet is encrypted (using SSH, Secure Socket Layer (SSL) or ESP) then filtering it is hopeless unless the keys are known to the firewall, or some clever control is exerted that automatically negotiates the key exchange with the firewall as well.

2.10 Default deny vs. Default allow

A security policy is a set of decisions that define an organisation's stance towards security. It should specify acceptable behaviour when using the organisation's resources, and what repercussions there are if these boundaries are crossed.

Although hosts should have all unnecessary ports blocked and have bug fixes applied as soon as they are available, a well implemented packet filter will block all connection attempts except to those services that are wanted/necessary. For example, mail for the organisation should only be delivered to one machine to be relayed on to its destination. If mistakes are made and an attempt is made to send mail to somewhere other than the mail server, or someone tries to relay mail to another unauthorised server, it will be blocked by the packet filter.

The last and most important rule in a packet filter ruleset is the default rule; either to "default deny" or "default allow" all packets that do not match one of the earlier rules in the ruleset. Whether you default deny or allow all other packets is determined by the security policy for the network; there is a trade-off between security and usability, depending on which stance is taken.

The default deny stance states that everything that is not expressly permitted in the ruleset is denied, and the default allow policy states that everything that is not denied by the ruleset is permitted through the firewall. Clearly, from a security point of view the default deny stance is the obvious stance to take, as you are only allowing the protocols and applications that are needed/wanted on the network. With the speed that new protocols are developed and utilised on the Internet this means that you only need to be extra vigilant of security bugs in the protocols and applications that are permitted through the packet filter: if you do not run a program it does not matter if it has security holes in it.

On the other hand, security is a trade-off with convenience. If users find the firewall too restrictive to do what they want to do (within the remit of the security policy, obviously) they will find more and more obscure, and probably insecure, ways of circumventing it. Ideally, the administrator needs to find secure solutions that allow users to complete the tasks they require, so that they do not find other ways of getting around the security systems in place; such as running peer-to-peer programs through SSH tunnels, or running SSH programs through HTTP tunnels. At least it is possible to monitor the usage of services accurately when everything isn't being tunnelled through HTTP.

2.11 Comparison of IPFW, IPTables, PF, IP Filter and Cisco ACLs

So far the rules have been described in a general notation. This is because each of the packet filters that are available for use (whether freely available or proprietary software) all use a different syntax. This section describes the differences between some of the freely available packet filters for Unix systems (such as FreeBSD, Linux and Solaris): IPFW, IP Tables, PF and IP Filter, and Cisco's Access Control Lists (ACLs).

A rule which permits TCP traffic from the 10.0.1.0 network to any address on port 22 is described at the end of each of the sections on the different packet filters to demonstrate the differences in their syntax.

On Unix systems, packets are filtered in the kernel. The packet filters listed above are implemented either as loadable kernel modules or they are built directly into the kernel when it is being configured. These kernel modules also have a corresponding "userland" program (usually of the same name as the kernel module) that allows the administrator to create rulesets and configure the packet filter more easily.

The logging option on packet filters can be used to constantly test and review rulesets to see if there are holes in the ruleset that need to be fixed and that there are no "back-doors" into the network. Programs like `hping` can be used to create arbitrary packets to test filter rules.

2.11.1 IPFW

IP Firewall (IPFW) is the default firewall for the FreeBSD operating system. It works by defining an IPFW configuration, or rule chain, which is used to match packets within the kernel. When a match is made, the action specified by the rule determines the fate of the packet. The rules are numbered from 1 to 65535 and the ruleset is processed in ascending numerical order. A default rule (numbered 65535), which cannot be deleted, catches all packets that do not get matched for processing earlier in the ruleset. The default rule can be set to either deny or allow, depending on what was specified when the kernel was compiled.

Rules can be put in "sets" to make configuration more manageable, as well as making it easier to swap between rulesets and "flush"(purge) a ruleset. If no set is specified when the rules are being defined, they are added to the default set, zero (0). The "skipto" option[10] can be used to skip all the rules up to a particular number and processing of the packet continues at the rule number given to the skipto option, or the next highest rule number. Skipto allows more complex rules to be defined than is possible with a simple first match system.

IPFW rules are usually of the form:

```
# ipfw add 100 allow tcp from 10.0.1.0/24 to any 22 out eth0
```

2.11.2 IP Tables/Netfilter

IP Tables/Netfilter is the default packet filter for Linux kernel based Operating Systems. It is the next generation of the original IP Chains program; Netfilter is the kernel part of the program and IP Tables is the userland part. Again, the rules can either be specified on the command line or in a configuration file which is loaded into the kernel during the boot up.

There are three standard rulesets or "chains": incoming, outgoing and forwarding. Each ruleset is applied as a packet enters, leaves or is forwarded by the Linux IP stack. The rules in a chain are used sequentially. If a packet matches the rule, the specified action is taken and the check terminates there and then. The three standard chains have a default policy (either deny or allow) associated with them, which is applied when the packet does not match any other rule.

It is also possible to establish "user defined" chains. These chains may be called from one of the standard chains via the jump flag (-j). If a packet does not match a rule in the user defined chain, it returns to the point where the user chain was called and continues checking until a match is made[11],

[10]The `skipto` option is similar to the `goto` function in BASIC.
[11]Thus, acting like `gosub` in BASIC.

or the default policy is reached in one of the standard chains (user defined chains do not have default rules). User defined chains allow more complicated rulesets to be built up with greater ease than IPFW's "skipto" option.

```
# iptables -A INPUT -o eth0 -p tcp -s 10.0.1.0/24 -d any -dport 22
```

2.11.3 IP Filter

IP Filter can be used on a number of Unix systems including the BSD kernel based OSes, the Linux kernel based OSes, HPUX and Solaris. It also works by matching packets against a list of rules that are defined in a configuration file.

The rules are checked in sequence for each packet and the *last* rule that successfully matches the packet determines its fate. It is possible to define rules which contain the "quick" keyword; which means that if a packet matches this rule, the rule checking terminates at that point (similar to the "first match" system that IPFW and IP Tables uses).

Rules can also be arranged into "groups" to help simplify complicated configurations. A group contains a head rule, which the packet is matched against initially to determine whether the rest of the rules in the group will be executed. The rules are then executed as normal. As with the chains in IP Tables: at the end of each group, processing of the packet continues at the next line in the ruleset.

The default policy rules must go at the beginning of the ruleset in IP Filter because it works on a last match basis. So, everything is denied/accepted initially (depending on the security policy) and then individual rules are created to deny/permit the desired connections. If the quick keyword is used to make IP Filter work as a first match filter, then the default rules will need to go at the end of the ruleset, as described previously.

```
pass out on eth0 proto tcp from 10.0.1.0/24 to any port eq 22
```

2.11.4 Packet Filter (PF)

Packet Filter is the default packet filtering program for OpenBSD[32]. The rules are implemented by the `pfctl` command, organised into multiple linked lists and is loaded into the kernel part of PF.

Packets are matched against rules on a "last match" basis, as with IP Filter, and can match more than one rule. Again, the general default deny/allow policy rules (`block all / pass all`) are followed by more specific rules to allow certain packets.

When a rule matches that is flagged as "final", the search ends and the action specified by the rule is performed on the packet; similar to the "quick" option in IP Filter.

The ruleset is automatically optimised by PF. If a group of consecutive rules contain a particular parameter and the packet does not automatically match on this parameter, the rest of the "group" is skipped and processing continues on. The PF syntax is mostly compatible with the IP Filter syntax.

```
pass out on eth0 inet proto tcp from 10.0.1.0/24 to any port 22
```

2.11.5 Cisco ACLs

Cisco provides Access Control Lists to perform packet filtering on its routers. ACLs are also used for selecting packets to be analysed, forwarded on, or logged. Each ACL is divided into different types, depending on the network protocol being filtered: IP (Standard and Extended), IPv6, IPX, Appletalk etc.

Each type of ACL is assigned an arbitrary number from a particular range to identify it. The ranges are used to specify the type of traffic to be filtered; the standard IP range is from 1–99 and the extended IP range is from 100–199. Standard IP ACLs only allow you to filter packets based on the source address of the packet. Extended IP ACLs allow for a much more detailed deny/permit policy, as you can filter packets based on the source and destination ports and addresses and by different transport types (TCP, UDP, ICMP etc.) as well. So, to perform proper packet filtering on cisco routers, the ACLs have to be from the extended IP range.

There is an implicit "deny all" rule appended to the end of each ACL, similar to IPFW and IP Tables. The syntax used to create filtering rules in the ACL is closest to that used in IPFW (or rather IPFW uses similar syntax to ACLs). The main difference is that subnets are defined using an inverted subnet mask instead of CIDR notation; i.e. to netmask off a /8 subnet you would normally specify a netmask of 255.0.0.0, but in an ACL it would be 0.255.255.255.

To specify a particular port to filter on, you have to use one of the following keywords: "eq" for equals, "gt" for greater than, and "lt" for less than. In IOS 10.3 and later releases, the "any" keyword is used to replace the cumbersome 0.0.0.0 255.255.255.255 syntax.

Separate lists are created for outgoing traffic from an interface and incoming traffic to an interface, and both must be specified to have a complete filter in place. Once the ACLs have been created, they must be applied to the interface that they are to be used on. Only one ACL can be specified for each direction per interface. The same ACL can be applied to multiple interfaces on a router. This is different to the packet filters described previously, where the rules were applied to *all* packets by default.

Cisco ACLs lack any kind of `goto` or `gosub` functionality, so all of the rules must be checked for each packet.

The following commands would be used to do this:

```
configure terminal
interface ser0/0 (or whatever interface you wish to use)
access-list 100 out
access-list 101 in
write
```

```
access-list 100 permit tcp 10.0.1.0 0.0.0.255 any eq 22
```

2.11.6 Features of Note

There is little difference in functionality between the various packet filters available. Each of the filters have additional functionality that distinguishes it from the others, and usually determines whether this will be the filter you use or not. IP Filter (and PF to a lesser extent) has a distinct advantage because it has cross-platform capabilities. IPFW is only available for FreeBSD, IP Tables is only available for Linux and IOS ACLs are only available on Cisco Routers.

IPFW contains the traffic shaping software "Dummynet", which is used to rate limit traffic (manage bandwidth) on the network. It can also be used to simulate networks and links, using queues and pipes, for testing purposes.

IP Tables provides a very strong Network Address Translation (NAT) system through its IP Masquerading software. It is also a modular firewall system, so you can write your own filters and load them directly into the kernel.

IP Filter has advanced filtering options, such as the ability to filter on the options information in the IP header, and it provides more flexibility in the handling of TCP options. It also provides a packet duplication feature that copies the packet and can then feed it into a network sniffer/intrusion detection system. This packet duplication feature can also be used to log the packet on a remote machine and makes for easier consolidation of log files for analysis.

PF has the ability to randomise the TCP sequence numbers (see section 2.3) by adding a randomly generated number to each sequence number in a connection. This protects hosts with weak sequence number generations from the possibility of connection hijacking attacks.

PF provides functionality to protect hosts behind its packet filter from fragmentation attacks. If a packet arrives to the firewall that is fragmented, PF caches it and when it has retrieved all the other fragments it reassembles the packet. This means that there can be no conflict between overlapping fragments (which again can be exploited by crackers, because different OSes deal differently with these overlapping fragments, as discussed previously in section 2.5); the packet is fully formed when it reaches its destination.

PF also provides functionality for TCP and IP normalisation (see section 2.8.2). Each of the various implementations of the TCP/IP stack that are used by the different Operating Systems have subtle differences in the way they interpret the protocol specification (RFC). Programs such as `nmap` have been written to analyse these differences and can provide information about the type of operating system, down to the version of the kernel, that a host is using. This information can be exploited by crackers to target a host for certain vulnerabilities. PF can be used to normalise traffic to and from hosts behind the packet filter in order to block OS fingerprinting attempts.

Other features of these packet filters, such as their stateful filtering will be discussed in Chapter 4.

Chapter 3

Stateless IPv6 Packet Filtering

The basic packet filtering techniques described for IPv4 in Chapter 2 also apply for IPv6; in particular, there will be no change to the way packets are filtered on their port numbers and flags. However, due to some major changes that have been made to the protocol as part of the new design, there are some IPv6 specific issues which need to be taken into consideration when creating a firewall: autoconfiguration of addresses; multiple addresses of different type (unicast, multicast or anycast) and scope (local, global, site, etc.) per interface; extension headers; and built in IPsec.

3.1 Packet Filtering on Addresses

Just as in IPv4, filtering packets based on their source and destination address is performed to determine and reject spoofed packets. An important consideration when filtering on address is to permit packets belonging to the stateless autoconfiguration process and the ND protocol in general.

It is trivial to write IPv6 equivalent rules for IPv4 rules that only involve addresses and interfaces. For example, all traffic with the local loopback address (`::1`) on the loopback interface (lo0) should be permitted:

```
add allow ipv6 from ::1 to any via lo0
add allow ipv6 from any to ::1 via lo0
```

3.1.1 Multiple Addresses per Interface

In Chapter 1, the concept of IPv6 hosts having multiple addresses per interface was discussed. This clearly has implications for packet filtering IPv6 enabled hosts; rules will have to be specified for each address they have configured. Otherwise, packets may not be filtered correctly or the filter might be circumvented by sending packets to another address on the interface that does not have rules associated with it.

IPFW provides a useful feature in its firewall via the "`me`" and (proposed) "`me6`" keywords. They are used to match any configured addresses on the hosts interface, which means that the firewall rules automatically adapt to local address changes. Obviously, this is only useful where the host is the one performing the filtering, as it does not take into account changes in the addresses for hosts that are being protected behind the firewall.

As global IPv6 prefixes are assigned by the ISP, if a network changes service provider its IPv6 prefix will also change. This means that hosts can have addresses with the new prefix and also "deprecated" addresses with the old prefix. The use of these deprecated addresses is discouraged for new connections but old connections can continue to use them for some (administratively specified) time. This means that the packet filter will need to accept packets to and from these deprecated addresses for the duration that they are being phased out.

3.1.2 Stateless Autoconfiguration

The autoconfiguration of addresses described in section 1.6 details the specific multicast ICMPv6 messages that need to be permitted for autoconfiguration to take place, such as the link-local "all-nodes" and "all-routers" addresses and solicited-node multicast addresses all from the `ff02::/16` block of addresses.

The following rules are required in the packet filter to allow the Neighbour Discovery protocol and autoconfiguration to function correctly behind a packet filter:

```
# Duplicate Address Detection
add allow ipv6-icmp from ::  to FF02::/16
# RS, RA, NS, NA, redirect...
```

```
add allow ipv6-icmp from FE80::/10 to FE80::/10
add allow ipv6-icmp from FE80::/10 to FF02::/16
```

The security problems posed by ICMP Redirect messages in IPv4 have not been transferred in IPv6 because the Neighbour Discovery protocol specification states that these messages should only ever be received “on-link”; that is, all ND packets have a hop limit of 255 so they cannot be used to influence the routing decision of packets that are not on the same link as the router is.

All link-local IPv6 traffic, both unicast and multicast, should also be permitted for the correct functioning of autoconfiguration; link-local addresses are also used for DHCPv6. It is also possible to run services over link-local addresses that you only want accessed by nodes on the local link; e.g. remote access to the DHCP server, but only via link-local hosts.

As in IPv4, IPv6 multicast addresses (both link-local and global) must not be used as source addresses, or appear in any router header. Also, routers must not forward on multicast packets beyond the scope indicated by the scope field in the destination address; i.e. site-local multicast must not leave the “site” and link-local multicast must not leave the local link.

```
# Allow any link-local multicast traffic
add allow all from FE80::/10 to FF02::/16
add allow all from mynetwork to FF02::/16
```

Although the autoconfiguration of addresses generally makes the life of the network administrator much easier, consideration must also be given to making firewall rules with IPv6 autoconfigured addresses. If the ethernet card on a machine needs to be replaced for some reason, this will mean that the host will have a new network identifier generated for it and these changes will have to be reflected in the firewall rules as well.

Autoconfiguring an address is the first thing a host does when it boots up, so these rules are usually put at the top of the ruleset on packet filters that work on a “first-match” basis, so the hosts can actually configure an addresses to be filtered on.

3.1.3 Filtering Privacy Addresses

RFC 3041[58] defines privacy extensions for stateless address autoconfiguration in IPv6. It addresses concerns that an attacker may be able to correlate the use of mobile devices and/or users based on the fact that the interface identifier, created from the MAC address and the `''fffe''` padding bits for all global unicast addresses, remains fixed even when the network prefix changes.

Francis Dupont and Pekka Savola[18] wrote an internet-draft that expressed concerns about the use of these privacy extensions. They noted that privacy extensions that changed with a high frequency made it difficult for Ingress filters to function correctly. It becomes much harder to distinguish between spoofed addresses formed from a topologically correct address within the same prefix as legitimate hosts that varies a few of the lower order bits and these privacy addresses.

Therefore, the use of privacy addresses should be curtailed so that packet filtering remains effective on the network. Where the use of these addresses is required, a separate subnet should be created for hosts using these addresses that is outside of the firewalled network.

3.1.4 Server Subnets

It may be useful to take advantage of the large number of subnets available in IPv6 to create a subnet that has all the servers that require access through the firewall on it; then create rules for the entire subnet instead of for each host.

3.2 Filtering ICMPv6

As well as permitting the multicast ICMPv6 messages that are part of the Neighbour Discovery protocol, there are other ICMPv6 message types which must be permitted for the correct functioning of IPv6. Firstly, "Destination Unreachable" messages should always be permitted just as in IPv4. Secondly, "Parameter Problem" messages such as Unrecognised Next Header field and Unrecognised Extension Header (IPv6 Option) should also be permitted to allow the correct functioning of the protocol through the packet filter.

```
# Allow ICMPv6 destination unreachable and parameter problem
add allow ipv6-icmp from any to any icmptypes dest-unreachable, parameter-problem
```

The `ping` and `traceroute` programs have also been ported to use IPv6 addresses, so where these programs are desired on the network rules will need to be defined to permit their traffic as described in section 2.6.

3.2.1 PMTU and IPv6

As discussed in section 1.2, the fragmentation field is not present in the main IPv6 header. It is no longer possible for routers to fragment a packet as it passes through; only the source node is allowed to do this. If a packet is received at a router that is too big for the link, it is discarded and an ICMPv6 packet should be sent to the originator of the packet to inform them that the packet was dropped. As the fragmentation of packets will no longer be common use, attacks such as those described in section 2.5 should not be a problem in IPv6.

This also means that Path MTU Discovery can not be tampered with, as it has been in IPv4, and will have to work correctly on the IPv6 Internet. Thus, it is important that these ICMPv6 "Packet Too Big" messages are always permitted. Otherwise, packets that are too big for the link will just be discarded en-route. The originating host will never know that the packets it is sending are too big and will continue trying to send them. It is not compulsory to use PMTU discovery in IPv6; where it is not in use, a host must not send IPv6 datagrams with an MTU larger than 1280. A rule to permit these ICMPv6 messages is implemented as:

```
# Allow Packet Too Big (do not filter it out)
add allow ipv6-icmp from any to any icmptypes packet-too-big
```

3.3 Filtering IPv6 Extension Headers

The most complicated part of filtering IPv6 packets comes when we try to filter the optional extension headers that come in between the main header and the upper-layer headers. Most IPv4 packet filters are based on filtering these upper-layer headers. Pekka Savola has noted a problem with the current IPv6 specification[71]:

> *"It is not possible to parse the header or proceed to the next extension header unless the processing of the previous header has been implemented."* but he notes *"It seems obvious that firewalls will always want to inspect the headers, and in whichever order they want."*

Indeed, all packet filters will need to be able to deal with Extension Headers in some way. At the very least, they must be able to skip past the extension headers by examining the next header value and length until they come to the transport layer header. This should be possible because all of the extension headers defined so far are encoded in Type-Length-Value (TLV) format, except the Fragmentation header. Packets that use this header can be reformed into the whole packet by the filter before forwarding it on to its final destination, as is common in IPv4.

Another option available is to just drop packets with particular extension headers, as is often done with IPv4 options, though this is probably not good practice as the full use of extension headers has yet to be realised; they may be used for multihoming or mobility in the future. The filter should reject packets with extension headers that are unknown or malformed, or that come in the wrong order in the header chain. There is currently no recommended behaviour for dealing with informing the source node that the packet was discarded, as the IPv6 specification only deals with how the destination node should respond to unknown or badly formed extension headers.

There is still the problem of how new extension headers will be processed. There is no requirement as yet that they should conform to the TLV format, so it will be difficult for packet filters to skip over them without making wild assumptions of the content. There is no immediate solution to this problem, other than Savola's solution that all new extension headers are standardised so that skipping over them is possible.

3.3.1 Hop-by-Hop Options

The Hop-by-Hop option header is the only extension header that should be processed by the nodes en-route as well as the destination node. Thus, packet filters must be able to process these headers too.

Suresh Krishnan has identified the possibility of a DoS attack occurring via the hop-by-hop extension header because there is no limit on the number of options that may be present in the hop-by-hop header, even though each of the extension headers may only appear once per packet[49]. Currently there are only a handful of hop-by-hop options defined by the IEFF such as Router Alert, Jumbogram and Tunnel Encapsulation limit which leaves 97 (0x61) undefined options which may occur any number of times in the packet; thus, when the router/host tries to process a packet overloaded with options, all of its CPU will be occupied causing the denial-of-service.

He proposes a solution to this security issue by limiting each option type to occurring only once per packet and specifying that the options must be arranged in reverse numerical order, with the exception of the "PAD1" and "PADN" padding bits which can occur anywhere in the hop-by-hop header. The final restriction he proposes is that these padding options must not occur side by side in the header. This proposal is currently an internet-draft and has yet to be adopted by the Internet community but it does raise issues regarding the best method of filtering and processing these headers.

3.3.2 Destination Options

It is also possible for there to be unknown options in the Destination options header, but they must be created with the TLV format[17]. So, the packet filter should at the very least be able to skip over these options until it comes to the transport layer header; it is probably a bad idea to pass packets with options which may significantly change the way the end host processes the packet and may compromise the security policy of the network.

The Destination Options header also uses the same "PAD1" and "PADN" bits that the hop-by-hop options use and so may also be used to cause a denial-of-service attack. Thus, the use of these padding bits should be restricted so that they do not occur side-by-side in the same packet.

The main use of the destination options header to date is for IPv6 mobility, so there will need to be attention paid to filtering packets with this extension header in the future. At the very least, it should be possible to match packets based on the Home Address, Binding Request and Binding Acknowledgement options.

3.3.3 Routing Header

Presently, there is only one type of routing header defined for IPv6, type 0, which is similar to the Loose Source Routing option of IPv4. The security problems experienced with source routing in IPv4 have not transferred to IPv6 because there is no requirement that the destination host use the inverse route in the source routing header to send the return packets. It can just send the packets directly to the source of the packet, so if the original packet was from a spoofed address the legitimate host will just reset the connection. Thus, it is not necessary to have a blanket ban on packets with this extension header.

However, care must still be taken when filtering packets with a routing header as it may still be used maliciously and to circumvent packet filters[70]. The first of these attacks may occur where a packet is sent to a host behind the filter that the attacker knows is permitted, say to port 80 on the web server. Another host is then specified in the routing header that is also behind the packet filter but remote access to this host is normally denied. That is, if the packet was sent directly to this host it would be correctly blocked by the packet filter. The packet filter should inspect the routing header's "Segments Left" field to make sure it is set to zero before passing it, so that can not be used to circumvent the filter.

The second attack possible is where a host may be used as a "reflector" in a DoS attack to hide the real source of the attack, though this problem can be mitigated by performing Ingress Filtering as in IPv4.

3.3.4 IPsec

As previously mentioned, IPv6 has provisions for implementing end-to-end encryption of packets using the Encapsulation Security Payload extension header. This obviously poses a number of problems with respect to packet filtering. The most obvious is that the real IPv6 and TCP/UDP headers used for filtering will be encrypted as well. The other problem is deciding whether to pass or deny this traffic through the filter; how do we decide whether the traffic is genuine or hiding something?

One proposed solution is to have the firewall, or some host that the firewall trusts, that is able to negotiate the encryption keys used for the connection in order to decrypt the packet to examine its headers for filtering. Another solution is to just encrypt the payload of the packet and to leave the real headers visible for inspection by the packet filter. Unfortunately, either solution would negate total security provided by the end-to-end encryption. There has yet to be any consensus on a solution to filtering encrypted packets, but most sites err on passing IPsec packets established by internal hosts to external hosts.

3.4 IPv4 to IPv6 Transition Mechanisms

A number of methods were devised by the Next Generation Transition working group (NGTRANS)[1] to provide for an easy transition from IPv4 to IPv6. The first of these solutions was to implement "dual-stacked" networks; that is, hosts on the network have both IPv4 and IPv6 networking stacks and their network interfaces are configured with both types of addresses.

As mentioned previously, these protocols are treated distinctly on the network so solutions such as Mapped Addressing and IPv6 encapsulated in IPv4 were devised to send packets between the two networks. The other transition mechanism solutions proposed are based around "tunnelling" IPv6 packets through the IPv4 Internet, such as 6over4, 6to4, ISATAP and Teredo.

3.4.1 Mapped Addresses

Mapped addresses were designed for use in dual-stacked networks to simplify address selection for applications running on these dual stacked hosts; all addresses on the host appear as IPv6 addresses at higher layers. This is achieved by taking the IPv4 address of the host and appending it to the IPv4 mapped address prefix "`::ffff:0000/96`".

These addresses are internal to the host running the protocols and so they should never appear on the wire[28]. Mapped Addresses are IPv6 representations of IPv4 addresses and so they pose a threat on dual stacked hosts; if an attacker is able to send a IPv6 packet with a Mapped address as the destination this could be used to by-pass the IPv4 packet filter. The following rules implement this:

```
    add deny all ipv6 from ::ffff:0.0.0.0/96 to any via eth0
```
[2]

[1]Now known as the IPv6 Operations working group (v6ops).

[2]The notation 0.0.0.0 is used to indicate any IPv4 address

```
add deny all ipv6 from any to ::ffff:0.0.0.0/96 via eth0
```

Colm Mac Carthaigh proposed using v4 mapped addresses and configuring IPv4 rules via mapped addresses instead of having a separate IPv4 and an IPv6 ruleset for dual-stacked networks[53].

> *"My approach for an ACL implementation (which whilst a layer removed, shares some of the problems) was to compile all subnet representations into 128bits, any v4 rulesets are converted into mapped format during compilation.*
> *During testing, the macros which handle v4 (mapped or otherwise) would only care about the 4th set of 32-bits. If a 128-bit comparison is somehow triggered, then we'll match that too."*

This should help to avoid firewall misconfigurations and the possibility of using mapped addresses to circumvent the IPv4 packet filter.

3.4.2 IPv6 Tunnels

As we saw in Chapter 2, packets that are tunnelled through another protocol are difficult to filter because the packet filter is unable to view the full headers of the packet. Thus, these packets will need special consideration by the IPv6 packet filter depending on which of the transition mechanisms is deployed on the network.

IPv6 packets encapsulated in IPv4, so-called manual tunnels, is one of the most widely available transition mechanism. It works by taking the original IPv6 packet and prepending it with an IPv4 header whose source and destination addresses are predefined at each end of the tunnel. These encapsulated packets are identifiable by the value in the Protocol field of the IPv4 header which is set to 41. So, if no IPv6 service is desired on the network, packets whose Protocol value is set to 41 can be discarded by the packet filter. If IPv6 is required, then packets will need to be de-encapsulated before they reach the packet filter so they can be filtered as normal IPv6 packets.

The 6over4 mechanism is used to automatically tunnel IPv6 over IPv4 networks[6]. It does this by taking a globally unique IPv4 address and appending it to the IPv4 Compatible prefix "`::/96`".

Packets that use these addresses are really IPv4 packets that want to interact with IPv6 enabled applications and as such they can be used to achieve objectives that were blocked in IPv4. Thus, packets with the compatible prefix should first be filtered to ensure that they do not contain malicious or unsuitable IPv4 prefixes (as described in section 3.5) and secondly, if the local network is dual-stacked packets from the local IPv4 address range should be blocked from entering or leaving the network altogether as they are probably trying to circumvent the IPv4 packet filter.

Packets using the 6to4 transition mechanism[7] can be identified by their "`2002::/16`" prefix. IPv4 networks can deploy 6to4 addresses on their network by taking one of their globally unique IPv4 addresses and converting it to hexadecimal to create a unique /48 IPv6 network. Only the edge router needs to be configured with 6to4; all the hosts behind this edge router will "speak" IPv6 natively and all the conversion between IPv4 and IPv6 will be done by the 6to4 edge router. Again if no IPv6 service is required, all packets with this prefix can just be discarded by the packet filter. If IPv6 is wanted on the local network, then ingress and egress filtering of packets with malicious IPv4 addresses must be implemented, and normal packet filtering can be applied to the IPv6 packets as though they had global unicast addresses.

There are two other tunnelling mechanisms that are currently under consideration by the IETF: ISATAP[76] and Teredo[34]. Both of these mechanisms have been designed for use in environments that use Network Address Translation (NAT) in IPv4. The filtering of packets that use these tunnels is implemented in IPv4 not IPv6. ISATAP packets can be filtered by checking the Protocol field is set to "41". Teredo tunnels work over UDP and do not use any particular port number, which makes them difficult to filter. These packets can only be filtered in IPv6 if they are de-encapsulated outside the packet filter before processing.

3.5 Ingress and Egress Filtering

It is just as important to perform Ingress and Egress filtering to block packets with spoofed source and destination addresses in IPv6 as it is in IPv4. In particular, it is hoped that Egress filtering will be more common in the IPv6 Internet. Anti-spoofing Ingress rules should be specified to block packets with source addresses that have the address prefix of the internal network arriving at the outside interface. Egress filtering rules should be specified to stop packets leaving the network with destination addresses that contain the global prefix of the local network. Rules should also be specified to keep multicast site-local addresses from leaking out to the Internet.

```
#Deny packets with a src addr from local network prefix on outside interface
add deny ipv6 from mynetwork to anyip in via eth0
add deny ipv6 from anyip to mynetwork out via eth0

#Deny Site-Local packets from entering and leaving the network
add deny all from ff05::/16 to anyip via eth0
add deny all from anyip to ff05::/16 via eth0
```

As well as filtering out these spoofed packets, packets with malicious Compatible IPv4 addresses should also be discarded at the edge of the network. These malicious packets are packets with private, broadcast, multicast and local loopback IPv4 addresses that are appended to the "::/96" prefix.

```
# Disallow packets to malicious IPv4 compatible prefix.
add deny all from ::224.0.0.0/100 to anyip via eth0
add deny all from anyip to ::224.0.0.0/100 via eth0
add deny all from ::127.0.0.0/104 to anyip via eth0
add deny all from anyip to ::127.0.0.0/104 via eth0
add deny all from ::0.0.0.0/104 to anyip via eth0
add deny all from anyip to ::0.0.0.0/104 via eth0
add deny all from ::255.0.0.0/104 to anyip via eth0
add deny all from anyip to ::255.0.0.0/104 via eth0
```

Packets that arrive or are trying to leave via the outside interface that have a 6to4 prefix (2002::/16) should also be filtered to ensure they do not contain IPv4 private, broadcast, multicast or loopback addresses as source or destination addresses. Rules to implement this are as follows:

```
# Disallow packets to malicious 6to4 prefix.
add deny all from 2002:e000::/20 to anyip via eth0
add deny all from anyip to 2002:e000::/20 via eth0
add deny all from 2002:7f00::/24 to anyip via eth0
add deny all from anyip to 2002:7f00::/24 via eth0
add deny all from 2002:0000::/24 to anyip via eth0
add deny all from anyip to 2002:0000::/24 via eth0
add deny all from 2002:ff00::/24 to anyip via eth0
add deny all from anyip to 2002:ff00::/24 via eth0

add deny all from 2002:0a00::/24 to anyip via eth0
add deny all from anyip to 2002:0a00::/24 via eth0
add deny all from 2002:ac10::/28 to anyip via eth0
add deny all from anyip to 2002:ac10::/28 via eth0
add deny all from 2002:c0a8::/32 to anyip via eth0
add deny all from anyip to 2002:c0a8::/32 via eth0
```

3.5.1 Unicast RPF and IPv6

Unicast Reverse Path Forwarding has also been adopted for IPv6. Again, it functions as a dynamic Ingress filter blocking spoofed packets that do not have a verifiable source address from gaining access to the local network. It is probably more important to implement Unicast RPF for IPv6 than IPv4 as there is an even larger block of un-allocated address space and un-routable address space, such as link-local and site-local addresses, in IPv6[9]. The network administrator will need to constantly check to make sure that no one is using this address space as globally routable address space. It also means that if someone is randomly choosing addresses, Unicast RPF will block all packets using an un-allocated or un-routable source address; which will save valuable computational time and effort.

Unfortunately, the problems caused by using Unicast RPF in a multihomed network (as described in section 2.4.2) are further complicated in IPv6 because of the method of address assignment. The consensus for single-homed cases is that global unicast prefixes are assigned by your ISP, which was chosen to curtail the growth of the global routing table and to allow for proper filtering of traffic in and out of the network.

There has yet to be a consensus for the multihomed case. One possible solution is to take an address prefix from each ISP. The questions then are: how do you decide which address prefix to use, and how should they be routed? There are a couple of options available such as using source routing *within* the multihomed site so that the packets are always routed to the exit router for the site that corresponds to the source address the packet came in on. Or, by dynamically issuing the correct source address using ICMPv6 messages (Neighbour Discovery) in the event of one of the links failing.

As there is little certainty yet of how Unicast RPF will affect IPv6, it is probably best to not use Unicast RPF on multihomed sites if you are using IPv6 for the moment.

Lindqvist and Abley note that a collection of anycast nodes on the Internet is similar to multihomed hosts from the point of view of the routing protocols, and as such may pose problems where Unicast RPF is implemented, even if none of the anycast nodes are multihomed[52]. Thus, they recommend that strict-mode Unicast RPF is not implemented by networks that are connecting to anycast nodes.

3.6 TARP Addressing

Transient Addressing for Related Processes (TARP) has been suggest by Bellovin and Gleitz[27] as an alternative to overly complicated stateful firewalling. It is based on the premise that separating network services based on their process groups, and using multiple addresses per host, can simplify the firewalling rules. Generally, a host (either a client or a server) just uses a single interface and a single network address to identify itself. TARP proposes changing this to multiple addresses per host: one for each process group identified on the host.

The notion of process groups and process management stems from a multi-tasking environment. Each task or execution of a program is called a process. Each process is then assigned an identifier in the form of a “process identifier” (pid). Processes are then organised into process groups as a means of collectively distributing control signals to the processes. These processes groups also need to have an identifier, called a “process group identifier” (pgid); this ensures that only processes from the same group can interact with each other.

Also, in the same way that related processes are grouped together to form a process group, related process groups can be gathered together into a session. What is being proposed then, is each group of related process will use the same TARP address to identify it. This will facilitate the creating of firewall rules to govern the network traffic that is being sent to these processes.

This proposed method of addressing would help overcome some well known firewall problems that arise when dynamic port negotiation is used by application layer protocols. Instead of trying to follow the needs of the application protocol as it develops, the firewall can make a single decision based on the transport layer protocol (be that TCP or UDP) and the source and destination addresses. There is no need to configure firewall rules as the traffic develops, or worse, keep a range of ports open unnecessarily. This means that the firewall can permit all traffic between the source and the destination, regardless of the port the traffic goes through.

There is no loss of control over the traffic with this method, because by segregating and controlling which addresses are allowed to offer network services outside of the firewall, it is possible to allow internal protected hosts to use the network services they require and deny requests from potentially dangerous sources at the same time.

3.6.1 Why IPv6 makes TARP easier to implement

In order to distinguish between the various client/server requests that arise on a network, the address used by a client will be tied to its process group identifier. That is, all related processes will use the same address, and this address will come and go as is required by the natural life-cycle of the processes that use it, such as occurs in a multitasked environment like UNIX.

IPv6 makes TARP easier to implement for a number of reasons. Firstly, IPv6 addresses are four times the size of their IPv4 counterpart so there is no address space constraints due to depleted address space, as is prevalent in IPv4. The IPv6 protocol specifically provides the ability to assign a multiple addresses per interface. Finally, the hierarchal allocation of addresses in IPv6 provides for the manipulation of the lower 64 interface bits, thus we can assign an address for each process group.

3.6.2 TARP Addressing Syntax

Typically, the 128-bit IPv6 address is broken up into its Network Identifier (the first 64-bits of the address) and a unique interface identifier (the bottom 64-bits of the address) on that network, and it is assigned either manually by the network administrator or automatically (using the autoconfiguration function of IPv6 described in section 1.6).

The addressing scheme used for TARP takes advantage of the 16 bits (usually the EUI-64 identifier that is described in section 1.6) that are used to pad out the address to the full 64 bits. This allows for 2^{16} IPv6 addresses (65,536 addresses) to work with. These addresses are then linked to a process group, and are used as the method for identifying related processes.

TARP addresses are created by actively modifying the padding bits to vary the address on the interface for each separate process. This is still be a unique identifier, because the other part of the address will be the MAC address (ethernet address) of the interface. The construction of TARP addresses varies slightly from the way it is done using the EUI-64, in that instead of splitting the MAC address and inserting in the extra 16 bits in the middle, the 48 bit MAC address is left as it is and the varying 16 bits are added at the end to make up the 64 bit Identifier. There is also no need to set the 7th bit as in the EUI-64 case. So, for a given MAC address of `00.0b.db.63.54.5d` gather the bytes into groups of 4 and separate with a colon, giving `000b:db63:545d`. The last four bytes of the address will then vary between `0000` and `ffff`, giving an address between: `000b:db63:545d:0000` and `000b:db63:545d:ffff`.

Each TARP address is defined as being of the form *Subnet Prefix:Ethernet Address:xxxx* with the last 2 bytes of the address corresponding to the process group identifier (pgid) that is associated with the address; *Subnet Prefix:Ethernet Address:pgid.*

If a port needs to be identified, it will be added to the address with the separation of a dot as normal. That is, *Subnet Prefix:Ethernet Address:pgid.21* describes an address using port 21.

3.6.3 Clients and Servers

It is important to note whether the host process is acting as a client or a server; client applications will select their address dynamically from the pool of addresses but the server applications must use a designated static address. This will usually be *Subnet Prefix:Ethernet Address:0.*

Similar processes groups must use the same address in order to conserve the number of available addresses. For example, different outgoing Telnet sessions will have a different TARP address associated with each one, whereas a web browser with media software such as a sound player running in it, which is structured as multiple processes, will only have the one address to identify it. Any subsequent bound sockets that arise from the same process group will be assigned the same TARP address as the rest of the processes in that group have.

Locking the server to a reserved address serves a number of useful purposes. Firstly it allows the client processes to locate the server: it is hard to reach a connection that keeps moving. Secondly, it assists the server/firewall interaction as the firewall does not need to be constantly updated with the server's new address, and it provides control for firewalled hosts running servers using wild-card listening sockets.

3.6.4 The Problems with TARP

The main problem that can occur when using the basic TARP scheme to assist in firewalling, is the problem of "coat-tail riding". This occurs when a host uses one service on a server, which it has been cleared to use through the firewall, to begin using another service which it does not have permission to interact with.[3]

Consider the case of a Firewall whose policy allows for outgoing ftp, but denies incoming telnet sessions: An internal host opens an FTP connection from *Protected Area:Ethernet Address:pgid.5040* to a compromised site *Compromised FTP server:0.21.* The firewall sees that this is compatible with its security policy and allows all traffic between the two servers. A telnet server listening on a wildcard socket on the internal host machine could then send and receive connection requests from the compromised server in violation of the security policy.

This problem can be remedied by forcing the server to operate on the `::0` address, and by preventing wildcard listening sockets from connecting to TARP addresses which they are not allowed to connect to. This is discussed further in the next section.

The other problem identified when using TARP occurs when the transport layer protocol that is being used is the User Datagram Protocol (UDP). It is assumed that servers will create sockets by listening on a set of reserved addresses for contact from a client. It must be ensured that servers will listen for connections, and respond to those requests from the same reserved address that the client originally contacted, in order to prevent coat-tail riding. UDP packets are somewhat problematic because certain kernels (BSD derived) do not force a multihomed server to respond to client requests from the same address as they were contacted at in the first place.

[3]We assume here that the only method of Firewalling employed is that of TARP and address based filtering. Obviously we can still use other Firewalling methods such as blocking incoming Telnet connections to ensure that coat-tail riding to these services does not happen.

Again, modifying the kernel so that a process group is classified as a server process — and therefore set to the *Ethernet Address:0* address — will force a response to the request on the server address.

3.6.5 Refining the Kernel

The kernel works by listening for incoming data and classifying that data according to the transport layer protocol and port the data used to establish a connection. It then matches the connection to a list of available sockets. As described in the section above, some problems can occur when using TARP addressing. By modifying the kernel's assignment rules as follows, Bellovin and Gleitz overcome these problems.

1. Control must be provided to prevent wildcard listening sockets from providing network services on an address assigned to another process group
2. It must allow for connections to servers on the address `Ethernet Address:0`[4]
3. It must connect the appropriate incoming packets to sockets listening on a specific TARP address
4. It must allow the server to match sockets listening to a valid address that is outside the reserve of TARP addresses

Using our coat-tail riding example from before, we see that the Compromised Server, which is attempting to connect to Telnet services from the outside, will fail to have its SYN packet matched because the rules require a *specific* destination address even though the telnet server is listening on a wildcard socket[5]. The SYN match will also fail because the Telnet servers listening socket will belong to a different process group than the FTP server, and will therefore have a different TARP address assigned to it. So, the packet that is trying to ride the coat tails of the open firewall rule is never delivered to the telnet socket on the machine, and so the connection is never actually opened.

[4]Normally connections to port 0 on hosts are not allowed, as this is the port that is used in socket programming to specify that an arbitrary ephemeral port should be used; described in section 2.8.2

[5]We check for SYN packets because these are the packets that are sent when a TCP connection is trying to be established, and programs such as Telnet use TCP to send data.

3.6.6 TARP and Firewalling

TARP has been designed to simplify dynamic (stateful) firewalling. Instead of creating more complicated filtering rules, TARP simplifies this operation by filtering on address only. For a connection based protocol like TCP, this means that the data only needs to be verified at the time when the connection is established.

The firewall rules governing this are defined so that for outbound packets the firewall needs to check: the outbound destination port; that the address belongs to the TARP family and whether the protocol involved uses secondary data connections using other ports (i.e. FTP in active mode).

For inbound data, the firewall should reject all requests for services which are not destined for the designated server address, *Ethernet Address:0*. For services which have been authorised, the data should flow freely between the server and the client.

When a connection is closed or a process group ends, there are several methods to revoke authorisation. Again this is most easily done with a TCP connection as the firewall just disallows packets between the source and destination address, once all the related processes/connections have been terminated. For UDP and ICMPv6 packets, the firewall can use a timer to revoke authorisation within a certain time period after the last use of an address. Alternatively, the host can explicitly release an address from a service once the process group terminates. Even if there is a delay in removing the authorisation from the firewall, the client is still protected because once the process group has terminated, its corresponding address is removed from that interface so the connection request or data packets never reach the host.

In summary, TARP incorporates interesting ideas that are not possible with IPv4. While not yet widely accepted, TARP shows how IPv6 makes new firewalling schemes possible.

3.7 Comparison of IPFW, IP Tables, IP Filter, PF and Cisco ACLs

All of the firewall programs described in Chapter 2 have been upgraded to perform stateless filtering of IPv6 packets. IPFW, IP Tables and Cisco ACLs filter IPv6 and IPv4 packets separately using

IP6FW, IP6Tables and “ipv6 access-list” respectively. IP Filter and PF implement IPv6 filtering with the same interface they use for IPv4. IP6FW was ported from the original IPFW so it only performs very basic filtering of IPv6 packets. Luigi Rizzo, who wrote the new version of IPFW — IPFW2 — is currently working on integrating support for the filtering of IPv6 packets with the same granularity available for IPv4.

IP Filter supports a full set of packet filtering techniques for IPv6 and ICMPv6 packets. Rules must be specified with the “-6” option so that they are parsed and loaded into the kernel. It even supports matching packets based on the contents of the extension headers. It is currently the most complete open source IPv6 packet filter available.

PF also provides for the filtering of IPv6 packets as part of its normal IPv4 filter. Here IPv6 packets are matched via the “inet6” and “icmpv6” keywords. PF has yet to implement complete filtering for Extension Headers; the only header it matches on is the fragment header and it blocks packets with this header by default.

Cisco has implemented the filtering of IPv6 via its Access Control Lists since version 12.0(21)ST of IOS. Initially, filtering could only be performed based on the source and destination addresses of the packet. In later versions of IOS support was extended to include filtering on IPv6 header fields (such as the Next Header, Traffic Class, Payload Length etc.), Extension Headers and also TCP ports and flags similar to the Extended ACLs of IPv4.

An interesting feature of Cisco IPv6 ACLs is that they contain an implicit “permit” rule to allow neighbour discovery packets, as well as the usual default deny all rule, so that the ND protocol will always work correctly.

```
permit icmp any any nd-na
permit icmp any any nd-ns
deny ipv6 any any log
```

The “stateful” features of these packet filters, where available, are discussed in Chapter 5.

Chapter 4

Dynamic Firewalling of IPv4

In the previous chapters, techniques were discussed for the filtering of traffic based on certain criteria: the source and destination ports and addresses, the direction of the traffic and the interface used. If TCP was used as the transport layer protocol, its flags (SYN/ACK/FIN/RST etc.) were also used to filter packets. Stateless packet filtering provides a thorough method of preventing malicious packets from entering a network, but it can be quite complicated to generate a complete ruleset that achieves all of this without running into mistakes.

Furthermore, rules must to be specified to allow the return packets of outgoing connections back in through the firewall so that protocols could function correctly. Large ranges of ephemeral ports needed to be opened up in the firewall by these rules, as there was no way of telling which ephemeral port would be used by the remote host to reply to the connection.

Dynamic filtering, so-called "stateful" filtering, creates state on the firewall for each connection that leaves the local network through the firewall and remembers the source and destination ports and addresses of the packet. This state creates a dynamic rule in the packet filter ruleset that allows the return packets from the outgoing connection back in through the packet filter. This means that it is not necessary to specify reverse rules in the ruleset for each service that is allowed through the filter, to allow the relevant return packets back in.

Thus, dynamic filtering is used to simplify packet filtering rulesets in order to avoid the problems of hijacking connections and insertion attacks that may happen with stateless filtering. It is also used to block port scanning and remote fingerprinting attempts, ensuring that packets follow the protocol from SYN to SYN/ACK through the data to the final FIN and hence belong to the established connection.

4.1 State Tracking

Dynamic packet filtering is achieved by the filter keeping track of each packet that passes through it, in both directions. It does this by creating an entry in its state table which corresponds to the packet, usually comprised of the quin-tuple of source and destination addresses and ports.

4.1.1 TCP

For TCP connections, when the first part of the "three-way hand shake" (a packet with the SYN flag set) is sent or received, a record of the source and destination ports and addresses is entered into the state table. Then when the three-way handshake is completed, a rule is created in the firewall to allow all subsequent packets related to this connection through.

Obviously there is a limit to the number of simultaneous states that can be stored at any one time[1], which leads to the question: How do you decide when states are removed?

TCP also uses a three-way handshake to close connections: a FIN packet is sent from the host wishing to close the connection; it waits for an FIN/ACK from the other end and then the connection is terminated when the final ACK is sent. When this final ACK is sent the state should be removed from the table. States should also be removed from the table when the host receives or sends a RST to a connection. However, if a machine crashes at either end of the connection the closing FIN or RST packets may never be transmitted so another mechanism for terminating states is needed.

Each packet that goes through the firewall refreshes the state and therefore keeps the connection open. We must also consider how long the state should remain in the table if there is no traffic being sent over the connection. This is more difficult to ascertain; there is a trade off between leaving a state too long in the table that may allow packets that are not related to the connection through, and leaving a state for too short a time in the table that results in legitimate connections (which may

[1]Otherwise the firewall would be opened up to a connection flooding DoS attack[50].

be idle) functioning correctly. There is also a trade off on the amount of memory used to store these states.

Currently, there are two methods for establishing the lifetime of a state in the state table. The first is to use a timer, which closes the state after N idle seconds (as determined by the administrator). The other option is to use "keep-alives". This is achieved by the firewall periodically sending an ACK acknowledging the last amount of data received. If the host at the remote end of the connection is still alive, it will send back an ACK and this will refresh the states lifetime. If the remote host has closed its end of the connection it will send back a RST packet and the firewall will know to remove the state.

The latter is a better solution for TCP, because it is difficult to set a timeout value that is long enough to allow for idle connections (such as SSH connections) that also removes stale connections to free up the state table for new connections. Short timeout values are irritating for remote SSH users because their connections will "hang" if they are not in constant use. Using "keep-alives" means that the state stays active on the firewall and the remote connection is always available.

4.1.2 UDP

Technically, it is not possible to track the state of a UDP connection because it is stateless by definition; i.e. it does not have a distinct start, middle and end. Nonetheless, dynamic packet filters still "keep-state" for UDP packets. Packets with the same source and destination ports and addresses are considered to be part of the same UDP connection.

Again, we must consider the problem of how to determine the end of the connection and when to remove the state from the table. The timeout method described above is more suitable for connectionless protocols such as UDP, as there is no way of knowing when the connection is finished. Many packet filters, such as IP Filter and PF, use an "adaptive-expiration" scheme[29] because the first UDP packet could either be a once-off or part of a "connection".

> *"The first [UDP] entry will create a state entry with a low timeout. If the endpoint responds, PF will consider it a pseudo-connection with bi-directional communication and allow more flexibility in the duration of the state [table] entry"*

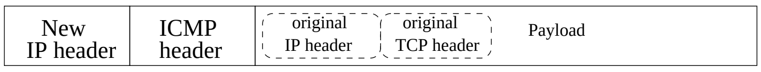

Figure 4.1: An ICMP Header

As an example, consider the problems that may arise when a host tries to resolve an address via DNS. The host will send out a UDP packet to the DNS server to find the corresponding IP address. This will create a state in the state table, but it only has a short lifetime (most implementations are set to about 1 minute). If the response is not received within this lifetime the state will be removed and the packet will not be able to return through the firewall. If the server takes longer than a minute to respond, the user probably will not wait that long for a response anyway.

4.1.3 ICMP

Dynamic filtering of ICMP packets is divided up into two categories: ICMP status messages that are related to another connection that has already created a state in the table; and ICMP queries and response messages.

ICMP messages that are a result of some problem with a TCP or UDP connection, such as ICMP “Time Exceeded”, “Port Unreachable”, “Packet Too Big” etc. are considered part of the established connection that created the state. As ICMP messages may come from a host en route to the destination, e.g. in the case of PMTU where the packet is too big for the link, the packet filter must match the source and destination addresses from the original IP header that is returned in the payload of the ICMP message (see figure 4.1) to the original packet information in the state table.

The other method of dynamically filtering ICMP packets is similar to the way UDP packets are handled. ICMP queries create their own state from the source and destination addresses and replies matching those queries are passed back through the firewall to the originating host[2]. This allows hosts behind the packet filter to ping remote hosts, without allowing remote hosts to ping your internal hosts. Again, a timeout value is set to limit the lifetime of the state in the table.

[2]The ICMP messages relating to TCP and UDP packets can quote the TCP/UDP header, so they may have ports stored as pieces of state indirectly. This is explained further in section 4.5.2.

4.1.4 Keep-State and Check-State

Packet filters implement dynamic filtering by appending "keep-state" to rules that permit connections. Each packet received by the filter is first checked for a match in the state table. This rule is usually put at the top of the ruleset so that packets that are already part of an established connection are checked for a match in the state table and if one is found they are allowed to proceed without being filtered. This helps to optimise the performance of the packet filter. Also, the dynamic filter matches packets to the ruleset using hashing or trees, rather than using linear matching that is usually used for manually configured rulesets.

If the `keep-state` option is specified in a rule, it means that any packet that matches this rule in the ruleset will have a rule dynamically created in the packet filter, which will also allow all the return packets related to this connection through the firewall without having to filter them.

```
add check-state
add allow tcp from myip to any 22 keep-state
add allow udp from myip to any 53 keep-state
```

Another feature of dynamic filtering is the potential to limit the number of connections a user can open/receive simultaneously. Connections can be limited by source address, source port, destination address or destination port, or a combination thereof.

```
add allow tcp from 192.168.41.0/24 to any setup limit src-addr 5
```

As useful as dynamic firewalling is for dealing with complicated filtering rules, there are problems associated with keeping state for connections. Firstly, if the packet filter crashes for some reason then all the states will be lost for current connections and they will have to be re-established, wasting time and network resources[3].

Secondly, the extra load on the packet filter's CPU and memory is intensive and places undue pressure on the running of the packet filter. It also opens up the possibility of the firewall being attacked via a DoS attack because it can only keep a finite amount of states; this problem is alleviated

[3]Some filters, such as IP Filter, allow for the saving of the state table to disk or transferring it to another machine so that the state table can be recovered in the event of the packet filter crashing.

by using "keep-alives" or timeouts for the states in the table. Finally, the packet filter must be positioned so that it "sees" all packets, or it must share state among a group of packet filters that do see all the packets.

4.2 Tracking TCP Sequence Numbers

There are different levels of keeping state. Some packet filters are considered "stateful" because they support the `established` and `setup` keywords described in section 2.3. In reality, these are not stateful because the filter only checks for the presence of an ACK or a SYN flag in the TCP header of that one packet. It knows nothing of the rest of the connection, the position of that packet in the connection, or if the SYN was ever sent!

Thus, an attacker could circumvent the packet filter by crafting a packet with the source and destination addresses and ports of an established connection and one of these flags set in it and it will be allowed through the packet filter. Some packet filters, such as IP Filter and PF, keep track of the sequence and acknowledgement numbers of TCP connections in addition to the source and destination ports and addresses. These extra pieces of state information prevent the insertion of spoofed packets into established connections, as the packets will be denied if their sequence and acknowledgement numbers are not in the expected range of values for these fields.

4.2.1 Real Stateful TCP Packet Filtering in IP Filter

IP Filter has supported the tracking of TCP sequence, acknowledgement and window-size fields for a long time with a view to reducing the opportunity for attackers to perform insertion attacks. The original implementation worked by checking each new packet against the current state table for a match. If no match was found a new state entry was created and added to the table. If a match was found on the source and destination ports and addresses, a subsection of code was called to check that the sequence and acknowledgement numbers corresponded to the next expected values for these fields. If they did come within the window-size of expected data the information in the state table was updated and the lifetime of the state was renewed. If they were outside the expected values, the packet was discarded.

Unfortunately, there were some errors in the way the checking of these TCP sequence and acknowledgement numbers was implemented[79]. The main error in the code was that it assumed that if the filter had seen a packet the end host must also have seen that packet, which is not necessarily the case if packets are lost en route. Secondly, the filter did not cope well when it received out of order packets or when packets were retransmitted, due to flaws in the tracking algorithm.

Guido van Rooij wrote a new version of the state engine code for IP Filter based on the design criteria that conclusions drawn by the state table should be provable, that all kinds of all kinds of TCP behaviour should be taken into account (including out-of-order packets, retransmissions and window-size probing) and in particular, the blocking of packets should be kept to a minimum and never cause connections to "hang".

> *"Ideally, we only want those packets to pass the filtering engine that are absolutely necessary for the correct functioning of the TCP session. Such a way of filtering will at least reveal maliciously inserted packet and might protect against yet unknown vulnerabilities."*

This was achieved by creating upper and lower bounds for the sequence and acknowledgement number fields, in order to correctly determine valid data from malicious data. These boundaries were devised on the premise that if an end host had seen a packet (based on the amount of data it acknowledged in its next packet) then the firewall should also have seen that packet (and hence it should be expecting the same acknowledgement number). The firewall keeps separate variables for the sequence and acknowledgement fields in both directions, instead of overloading them as in the old implementation.

Packet filters that implement this correct form of filtering based on the sequence, acknowledgement and window-size fields are less likely to fall foul of attacks such as the TCP RST attack noted by Paul A. Watson[81]. This is true if the packet filter imposes more restrictions on the valid sequence numbers for a RST than the targeted TCP stack.

4.3 Protocol Checking

Another useful feature of dynamic packet filtering is protocol checking. It is similar to application-layer proxy servers, as it uses its knowledge of the protocol in question to dynamically configure rules to permit or deny traffic of this type. This is particularly useful for protocols such as active-mode FTP, which is difficult to permit with a stateless firewall without leaving large ranges of ephemeral ports open to permit the use of the data channel.

If dynamic filtering is used to process FTP connections, a state is created as normal when a new FTP connection begins. The packet filter listens for the PASV and PORT commands that are used to negotiate the end points of the data channel in the packets of that connection. When it sees the PORT command called, it dynamically creates a rules to permit the sending and receiving of packets over the data channel that are related to that particular FTP session.

```
add check-state
add allow tcp from my-net to any ftp setup keep-state --related
```

Protocol checking is also useful for checking that tunnels are not used to pass other traffic through a normal HTTP session, or by checking for protocol specific information in the headers and the data of the packet. For example, Packet Filter (PF), is able to rate-limit traffic using ALTQ and dynamic filtering to give more bandwidth to SSH connections than SCP/SFTP connections[47].

Obviously, this level of filtering is only available for a limited number of well known protocols, similar to the use of proxy servers. Furthermore, it is often possible for attackers to by-pass these checks by using alternative methods of sending the data. For example, DNS packets can be crafted so they look and behave like the real thing but they may contain unexpected or unusual data that the protocol checker has no way of identifying, e.g. “IP over DNS”[54].

4.4 Another Dynamic Packet Filter

The implementation of dynamic filtering of packets described so far works by making changes to the ruleset on the fly to permit or deny packets. Bellovin and Cheswick note that packet filters are sensitive applications, and changes in their configuration must be handled carefully because the order of the rules is very important [[8], pg 189]. They suggest an alternative method of dynamic filtering where no changes are made to the ruleset. Instead each connection is terminated on the filter, then the filter “re-dials” the connection and sends the packet to its final destination (see figure 4.2).

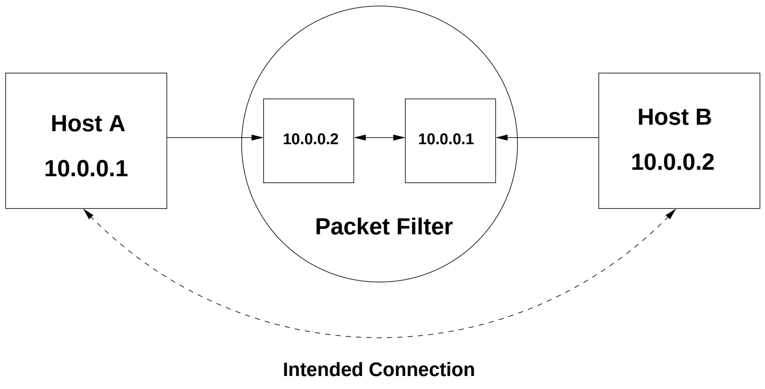

Figure 4.2: Dynamic “Re-dial” Packet Filter

The firewall replaces its source address with that of the host that originated the connection and transfers the data to its intended destination (if it is permitted); so the process appears seamless to both ends of the connection.

In effect, this dynamic filter behaves like an application proxy server. It simplifies TCP connections as there are no special requirements for handling connections. UDP packets still require a timeout of some kind, as with the normal stateful dynamic filter, to indicate the end of the connection to the filter. The handling of ICMP packets is also simplified. If an ICMP error messages is received at the filter for one of the connections it has re-dialled, the filter can just create a response to this and forward it on to the host that instigated the connection in the first place.

The intention of this alternative dynamic packet filter is to simplify the creation of rulesets, but this benefit may be lost in the connection handling of the filter and the complexity of the set up. It seems that the same level of consideration must be employed for the set up of the filter, as that needed for creating a dynamic filter ruleset.

4.5 Dynamic Filtering with IPFW, IP Filter, IP Tables, PF and Cisco ACLs

All of the packet filters described in Chapter 2 can filter packets dynamically as well as statelessly, except for Cisco ACLs. If dynamic filtering is required on Cisco equipment then an extra piece of hardware, a Private Internet EXchange (PIX) card, must be bought to do this.

4.5.1 IPFW

IPFW tracks the source and destination addresses, port numbers and protocol type when keeping the state of a connection for dynamic filtering. The "check-state" keyword is used to check a packet against the current dynamic ruleset. If a match is found the filtering terminates and the packet is passed unconditionally. If no match is found, filtering continues with the next rule until a match is made.

When a packet matches a rule with the "keep-state" keyword, a dynamic rule is created to allow any subsequent packets in either direction. Dynamic rules created by the keep-state rule have a default lifetime, set by a sysctl variable, which is refreshed every time a packet matching the rule is found.

```
ipfw add check-state
ipfw add allow all from any to any 22 keep-state
```

4.5.2 Netfilter/IP Tables

Netfilter is the first Linux based packet filter that provides stateful filtering with "connection tracking". It tracks connections based on the source and destination ports and addresses, status information from the TCP flags (SYN/ACK/FIN flags etc.) and the TCP sequence information.

Dynamic rules are created when packets match rules that have the "-m -state" parameter. There are five defined states: NEW, ESTABLISHED, RELATED, RELATED+REPLY, and INVALID. The NEW state is used to generate a dynamic rule for new connections. The ESTABLISHED keyword is used to match packets against dynamic rules that are already part of an established connection. The RELATED keyword is used to match packets that are related to a connection and are passing in the same direction, such as ICMP messages. The RELATED+REPLY state is for packets that are not part of an existing connection but are related to one that is, such as FTP data channels related to packets from an existing FTP control connection. The INVALID state is used to match packets that do not belong to an existing connection.

```
iptables -A INPUT -s any -d any -dport 22 -m -state --NEW, ESTABLISHED
```

4.5.3 IP Filter and PF

IP Filter's dynamic filtering of packets is described in detail in section 4.2.1. Again, it dynamically filters packets based on the source and destination IP addresses, TCP and UDP port numbers, and also the TCP sequence, acknowledgement and window-size fields.

```
# sample IP Filter rule
pass in all from any to any port = 22 keep state
```

PF performs dynamic filtering with the same state parameters as IP Filter. The current state table can be viewed using the "-s state" option of `pfctl`.

```
# sample PF rule
pass in all from any to any port 22 keep state
```

Chapter 5

Dynamic Firewalling of IPv6

The same principles for the dynamic filtering of packets exist for IPv6. A state table is generated by the filter, which stores information about each connection passing through it based on its “state”. Again, this state is created from the source and destination ports and IPv6 addresses, and the TCP sequence, acknowledgement and window-size fields. Thus, the techniques outlined in Chapter 4 can easily be applied for IPv6 packets.

5.1 Dynamic Filtering with IP6FW, IP Filter, IP6Tables, PF and Cisco ACLs

There is no dynamic filtering of IPv6 packets available with IP6FW, IP6Tables or Cisco ACLs. There are plans to introduce connection tracking for IPv6 packets in a future release of IP6Tables. There are also plans for dynamic handling of IPv6 packets in the updated version of IPFW2 that Luigi Rizzo and his students are working on. As in IPv4, there is no provision for the dynamic filtering of IPv6 packets using Cisco ACLs as stateful filtering is achieved using Cisco PIX hardware.

IP Filter and PF both provide dynamic filtering for IPv6. Both keep state in the same way as they do for IPv4 packets; they use the IPv6 source and destination addresses, the TCP ports for the source and destination hosts, the TCP sequence, acknowledgement and window-size fields.

5.2 Using the IPv6 Flow Label for Dynamic Firewalling

This chapter aims to discover if it is possible to use another part of the IPv6 header as a piece of state, for the purpose of dynamic filtering. I propose using the Flow Label field to do this. First we must discuss what the Flow Label field is and what it is normally used for. Then, we must check that the field remains constant across the lifetime of the connection.

5.2.1 What is a Flow Label?

The Flow Label is a 20-bit field that is part of the IPv6 header. It is used by a source to label certain groups of packets for easy identification and handling by routers. These groups of packets are called a “flow”.

The Flow Label is assigned to a packet by the source node, with the label being chosen pseudo-randomly from the range 0x00001 – 0xFFFFF. A Flow Label should be chosen randomly so that any set of bits within the Flow Label field are suitable for use as a hash key by routers (for looking up the state associated with the flow). The router, or other devices using the Flow Label, should not depend on this being the case if they want to avoid a Denial-of-Service (DoS) attack on the hash table using maliciously chosen flow IDs[12]. The Flow Label 0x000000 (zero) is reserved to indicate that the packet is not associated with any flow[17].

The Flow Label is not covered by the Authentication Header (AH) of IPsec, so in practice you can not tell if it has been interfered with or not en-route to its destination. RFC 3697 states that the Flow Label is immutable[66]:

> *“The Flow Label value set by the source MUST be delivered unchanged to the destination node(s)”*

For IPv4, flow classifiers are based on the quintuple of source and destination addresses and port numbers, and on the transport layer type. It is possible that some of these fields may not always be available; due to fragmentation (in IPv4) or encryption (IPv4/v6), or having to locate them after a chain of IPv6 extension headers. It is also more efficient to base the classifiers solely on the IP header, so that the introduction of newer transport types will not require “flow aware” routers to be updated.

Thus for IPv6, the flow is uniquely identified by the source and destination addresses and the (non-zero) Flow Label. All packets belonging to the same flow must be sent with this same triplet. If there is a Priority label, Hop-by-Hop options header or Routing Header associated with the flow, they must also remain consistent throughout the flow (excluding the Next Header field in the Hop-by-Hop options header and Routing Header obviously).

Routers that do not support the functions of the Flow Label are required to:

- When a packet originates locally, set the field to zero;
- When forwarding a packet, pass the field on unchanged;
- When receiving a packet, ignore the field.

Routers may choose to set up a flow-handling state for any packet, even when one has not been explicitly established by the source of the packet. It can choose to remember the details of the processing of the packet, such as the source address and Flow Label, and cache that information.

There is no requirement that all, or even most, packets belong to flows. There may be multiple concurrent active flows between two hosts, as well as traffic that does not belong to any flow.

5.2.2 Using the Flow Label

The Flow Label field is currently used to identify streams of related traffic to provide Quality of Service (QoS). In particular, real-time flows can make use of the Flow Label field.

We aim to investigate the potential usefulness of the Flow Label for Stateful Firewalling; i.e. as another method of identifying and filtering packets. Can the Flow Label be examined and stored by the firewall? This would at least introduce an extra piece of state that would need to be correctly guessed in order to blindly inject spoofed packets into the network; thus adding another layer of protection to the firewall.

In order to establish whether this is possible, we must first check to make sure that the Flow Label actually stays constant across the lifetime of a flow in practice. Otherwise requiring a fixed Flow Label at the firewall would block legitimate packets. We will now describe the efforts taken to check for this consistency.

Many Operating Systems, such as those that use the Linux kernel, just set the Flow Label to zero. FreeBSD (and other KAME derived IPv6 stacks such as NetBSD and OpenBSD) can enable setting the Flow Label using the sysctl variable: `net.inet6.ip6.auto_flowlabel`.

5.2.3 Examining the Flow Label for consistency

A packet capture program[1] was written which enabled the tracking of all the packets on the network, even those packets potentially destined for other hosts (see Appendix B). Using this program, it was possible to track the source and destination addresses; ports; transport type (i.e. TCP or UDP; we are mostly concerned with TCP) and the Flow Label of all the packets coming and going on the interface. The packet capture program extracts this information from the IPv6 and TCP headers of each packet.

The program was used to check to see if the flow was already in a table of active connections; as determined by the quintuple of source and destination addresses, source and destination ports, and transport type. The program then checked to make sure the Flow Labels matched for each packet; if they did not, it logged a message. If there was no such flow, it was added to the table of active connections. The program also deleted ("flushed") any stale flow once the connection had been closed or timed out.

The main objective of this program was to check to see that the Flow Label was kept constant for the lifetime of a TCP connection. Initial observations of the output of the program showed that no Flow Label was set by the local machine when establishing a connection (on both FreeBSD[2] and Debian Linux). On FreeBSD however, a Flow Label was set by the remote machine on the first response packet. But, another different Flow Label was then sent when the transfer of data began. This problem was identified as a problem that may occur with all implementations of TCP optimisation: SYN Cookies/SYN Caching.

[1]The libpcap library is part of the TCPdump program (`http://www.tcpdump.org`)

[2]I wrote a patch which set a (proper) Flow Label on outgoing connections, which was then merged into the KAME code

5.2.4 Results of the Packet Capture Program

Further tests were run to see if the problem identified by my initial observations was widespread, or just local to KAME implementations of IPv6 and operating systems that use TCP optimisations. A long list of IPv6 enabled websites was obtained to perform tests on using the packet capture program to establish if this was the case[42]. A short Perl script was written to download each of these webpages using the `fetch` command, in order to generate IPv6 traffic between my FreeBSD machine and these hosts. While this program was running, the packet capture program was executed concurrently and the output was saved to a file for examination.

Two biases are observed in the results of the tests performed. Firstly, my FreeBSD machine is always a part of the connections that were captured. Secondly, all of the connections captured are responses to connections initiated by my machine. No observations are made of how the Flow Label is set by these remote hosts when they initiate connections to other machines. In addition, ICMP and UDP packets were also captured while generating traffic with HTTP requests and are incidental to the results obtained for TCP packets.

Table 5.1 details the breakdown of the results obtained. The majority of the packets captured were packets with a global unicast address prefix. The rest were mainly link-local Neighbour Discovery packets which are irrelevant to our analysis. Packets from 1416 unique hosts were captured, with 1105 of these hosts sending TCP packets. Interestingly, one host chose to set a Flow Label for its ICMP packets. Unfortunately this host, `3ffe:202c:ffff:60::1`, was unreachable when later tests were performed to establish its operating system.

The hosts that sent TCP packets are of the most interest to us for checking the consistency of the Flow Label, as UDP and ICMP are not inherently stateful protocols. Table 5.2 contains a detailed break down of the setting of the Flow Label by the various hosts observed. Of the 1105 hosts that sent TCP packets, 427 of them (38.64%) set the Flow Label to something other than “`00000`”; but only 84 of these consistently set the Flow Label during the flow. The remaining 343 set the Flow Label once during the three-way handshake and then again once the transfer of data started, as was noted in my initial observations.

The remaining 678 hosts (61.36%) set the Flow Label to “`00000`” consistently. These hosts are most likely using some Linux distribution or OpenBSD. OpenBSD explicitly states in its source code that it leaves the flow label field set to zero so that it does not require any state management.

Description	Total
Total number of packets captured	55206
Number of globally addressed packets	45680
Number of link-local (fe80::) and multicast packets (ff02::)	9526
Number of unique hosts	1416
Number of ICMP packets	9403
Number of hosts sending ICMP packets	311
Unique hosts setting a non-zero Flow Label on ICMP packets	1
Number of UDP packets	12
Number of hosts sending UDP packets	2
Unique hosts setting a non-zero Flow Label on UDP packets	0
Number of TCP packets	36265
Number of hosts sending TCP packets	1105
Unique hosts setting a non-zero Flow Label on TCP packets	28107
Unique hosts setting a zero Flow Label on TCP Packets	8158

Table 5.1: Table of Frequency of Flow Label Setting on captured packets

TCP Hosts and the Flow Label	Hosts	%
Hosts with Flow Label set to something other than 0	427	38.64
Hosts with the Flow Label = 0	784	
Hosts consistently setting Flow Label to 0	678	61.36
Hosts who set the Flow Label to 0 and set it again after handshake	106	
Hosts who set the Flow Label (but not necessarily consistently)	321	
Hosts who consistently set the Flow Label (never changes)	84	19.67
Hosts who inconsistently set the flow (incl. setting it to 0 at some stage)	343	80.33

Table 5.2: Table of consistency of Flow Label Setting on TCP packets captured for each host

Hosts that consistently set the Flow Label	**Hosts**	**Percentage %**
FreeBSD (KAME stack)	21	25.0
NetBSD (KAME stack)	4	4.76
Sun (Solaris)	5	5.95
Others	36	42.86
Unable to connect (to port 22)	18	21.43

Table 5.3: Table of Consistent Flow Label setters by Operating System

Responsiveness of consistent Flow Label setters	**Hosts**
Responded	62
No response	22

Table 5.4: Table of Consistent Hosts responding to Node Information Queries

Next we attempted to establish the operating system that was used by the hosts that consistently set the Flow Label. To do this another script was written that used the `netcat` program; it collects the banners displayed by the machines when they are remotely accessed. Table 5.3 contains the results of this test. As anticipated, hosts running the FreeBSD operating system were the most consistent setters as they are running a version of FreeBSD that has been patched to fix this problem.

A second test was performed to see which hosts respond to ICMPv6 Node Information solicitations. This is a feature that is only implemented by the KAME IPv6 stack used in FreeBSD, OpenBSD and NetBSD. Table 5.4 shows 62 hosts responded to the queries sent to them and 22 hosts did not. It is possible that some of the hosts were just not responding to any ICMP probes, so they were further checked for reachability using `ping`. Of the 22 hosts that did not respond to the Node Information queries, 16 hosts responded to normal echo request and replies so it was concluded that they were probably using another IPv6 stack (such as the Linux stack); 6 hosts did not respond at all.

The same tests were then performed on the 343 hosts that inconsistently set the Flow Label to establish which operating system they used. Table 5.5 contains the results of the first test. 39.36% of the hosts used the (un-patched) FreeBSD operating system and 10.20% used NetBSD.

The Node Information solicitation test was then performed on the inconsistent Flow Label setters. Table 5.6 shows 295 hosts responded to the ICMPv6 Node Information queries sent to them, which

Hosts who inconsistently set the Flow Label	**Hosts**	**%**
FreeBSD (KAME stack)	135	39.36
NetBSD	35	10.20
Others	50	14.58
No banner	108	31.49
Unable to connect	15	4.37

Table 5.5: Table of Inconsistent Flow Label setters by Operating System

Responsiveness of inconsistent Flow Label setters	**Number of Hosts**
Responded	295
No Response	48

Table 5.6: Table of Inconsistent Hosts responding to Node Information Queries

means that they are running the KAME IPv6 network stack; 48 hosts did not respond to these ICMPv6 messages. Again, these 48 hosts were probed further to establish if they were reachable at all or whether they were just not responding to the Node Information queries. Of the 48 hosts that did not respond to the initial queries, 11 of the hosts responded to the pings and 37 hosts did not respond at all.

In summary, most of the hosts that inconsistently set the Flow Label either used the FreeBSD or NetBSD operating system. Hosts that consistently set the Flow Label to something other than `00000` were mainly running the patched version of FreeBSD and Solaris. Hosts that consistently set the flow label to zero were either using OpenBSD or some version of Linux.

In order to discuss what is actually occurring when different Flow Labels are being set during the TCP connections and how the problem was fixed for FreeBSD, we must first discuss SYN Flooding, SYN Cookies and SYN Caching. The solution to this problem that was later integrated into FreeBSD will also be described.

5.2.5 SYN Flooding

When a TCP server receives the first SYN packet of a new connection, it replies with a SYN/ACK packet; at this point the connection is considered "half-open". If memory is allocated for each half-open connection, then the number of pending connections will be limited by the amount of memory dedicated to this purpose.

Thus, it is possible for someone to cause a Denial-of-Service attack on a host, by repeatedly creating half-open connections and not responding to the resulting SYN/ACK with the final ACK of the 3-way handshake. When this happens, all of the available protocol control blocks (PCB) are utilised, so no legitimate connections can be made to the host. This type of attack is known as SYN flooding.

The first major demonstration of this type of DoS attack occurred in September 1996, when the New York based Internet Service Provider, PANIX, was taken off the Internet for a week by repeated SYN flooding attacks[25]. The attack had been known about for some time, but it was not until an explanation of the attack and sample code was published in the hacker magazines Phrack and 2600[19] that it became popular and was perpetrated on PANIX, as it required a sophisticated knowledge of the workings of TCP. The attack was explained in Phrack[13] as follows:

> *"The attacking host sends several SYN requests to the TCP port she desires disabled. The attacking host also must make sure that the source IP-address is spoofed to be that of another, currently unreachable host (the target TCP will be sending its response to this address. IP may inform TCP that the host is unreachable, but TCP considers these errors to be transient and leaves the resolution of them up to IP (re-route the packets, etc) effectively ignoring them.). The IP-address must be unreachable because the attacker does not want any host to receive the SYN/ACKs that will be coming from the target TCP (this would result in a RST being sent to the target TCP, which would foil our attack)."*

The problem of SYN flooding persisted for some time as it was exploiting a feature of TCP, and there was no immediate solution to the attack. It was generally thought that the only way to fix the problem was to make changes to the TCP protocol. In late September 1996, Dan Bernstein devised SYN Cookies[3] which would alleviate the problem without the need for changes to the TCP protocol.

SYN Cookies are created by making a MD5 hash[69][3] from a number of parameters, which can be

[3]Of course another hashing algorithm could be used, but this is the one Dan Bernstein suggested.

found in the SYN and ACK packets of the 3-way handshake. These parameters include the source and destination addresses and ports. The Maximum Segment Size (MSS) is also encoded in the cookie. Also included in the hash is a time–based server selected secret. This is then sent as the 32-bit TCP initial sequence number (ISN) by the server. When the host receives the ACKnowledgement for the SYN packet, it can rebuild the MD5 hash from the ACK packet and compare it with the ACK number (which is the ISN+1). This removes the need to store the ISN and other information about the connection during the handshake.

The main problem with using SYN Cookies to store connections is that other TCP features, such as TCP options, cannot be used. The server remembers nothing about the connection while waiting for the returning ACK, so options that the originating host sets are lost when the server rebuilds the connection from the information stored in the SYN Cookie.

A second method for optimising a TCP implementation to prevent SYN flooding, SYN Caching, was developed by David Borman for BSDi[4]. SYN Caching is similar to the normal method of storing connection information in a PBC structure, but instead a much smaller PCB is allocated which only remembers the key information for the setup of the connection (such as TCP options like window scaling and timestamps). While it might appear that the SYN Cache structure is just as susceptible to SYN flood attacks as the TCP structure even though it is a much smaller structure, some of the risk is alleviated by the use of a hash table to store the incomplete connections instead of a linear chain. Attacks on the hash table are then mitigated by the addition of a hash secret which is used to obfuscate the information in the hash table.

SYN Caching was integrated into FreeBSD[64] by Jonathan Lemon[51]. FreeBSD now passes all received SYNs through SYN Caching (failing over to SYN Cookies if the SYN Cache is full), so in FreeBSD traditional TCP connection establishment is now defunct.

5.2.6 Problems with TCP Optimisation

It is easy enough to keep the Flow Label constant for a simple TCP implementation, as another variable can just be added to the TCP PCB structure to remember the Flow Label. When TCP optimisations, such as SYN Caching and SYN Cookies, are used to accept connections these techniques must also keep the Flow Label constant. One simple option would be to set the Flow Label to zero for all such connections.

As previously explained, in FreeBSD the socket remembers the information related to the connection using a SYN Cache. When the host receives an ACK to the SYN/ACK it sent to the originating host, it retrieves the SYN Cache entry and turns the information stored within, into a full TCP connection. If there a too many connections being established and the cache fills up, the host can create a SYN Cookie instead of storing the information in a SYN Cache.

The problem we noted in section 5.2.3 arose because the code in the FreeBSD kernel did not support setting the Flow Label correctly for packets generated by the SYN Cache/SYN Cookie code; the Flow Label was not being set on the SYN/ACK part of the handshake. It was only being set when the first packet of data was being ACKed. Instead, whatever junk that was in the memory allocated to the packet was being sent as the Flow Label! Modifications were made to the kernel code to amend this problem and committed to the FreeBSD Operating System source code.

In the SYN Cache case, the Flow Label is generated either sequentially or randomly[4], and stored as a variable added to the syncache structure (`u_int32_t sc_FlowLabel`)[5]. When the ACK is returned a full PCB is constructed and the Flow Label is copied from the SYN Cache, allowing subsequent data packets to be sent using the same Flow Label.

SYN Cookies on the other hand, do not remember anything about the connection while waiting for the ACK to return. The SYN Cookie is derived from the source and destination IP addresses and ports and a secret (which is randomly generated and has a bounded lifetime) which are then sent through an MD5 hash, with 25 bits of the resulting output being used in the Cookie, and another 7 bits which contain the window-size. I propose using a further 20 bits of the MD5 hash to generate the Flow Label. We can do this because we are free to choose any (random) Flow Label[6].

[4]We are free to choose either for the SYN Cache case. There is less potential for collisions if the Flow Label is set sequentially. Strictly speaking the Flow Label (and addresses) should uniquely identify a flow, but this is not enforced by KAME. The risk of collisions could be reduced by checking for existing flows with that ID and if one exists, resorting to a zero flow label. The use of a zero Flow Label could be encoded in a SYN Cookie in a similar way to that used to store TCP MSS.

[5]On most computer architectures, memory is allocated on "power of two" boundaries (i.e. 1, 2, 4, 8, 16 bits etc.). It is important that the extra variable is placed in the correct part of the structure so that the overall size of the structure does expand and fill up with empty padding, and ends up being as big as the original TCP structure; thus defeating the purpose of using SYN Caching.

[6]It is not possible to choose a sequentially increasing Flow Label for the SYN Cookie case. We must use the value generated from the hash because it is always possible to recreate the hash, therefore it is always possible to recreate the Flow Label.

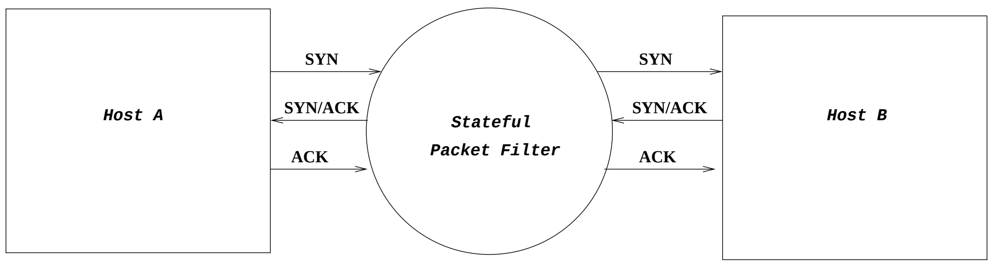

Figure 5.1: Stateful Packet Filter diagram

The host rebuilds the connection from the information stored in the ACK. If the ACK received matches the ISN sent, then the SYN Cookies will be the same. This means that we can reproduce the Flow Label sent with the SYN/ACK (as it is generated from the SYN Cookie), but not other TCP options which are chosen by the originating host.

The addition of this code means that FreeBSD now sets the Flow Label at the beginning of a TCP connection, and it remains constant across the lifetime of the connection. Any operating system that implements techniques like SYN Caching or Cookies will need to use similar measures to the ones described to ensure the Flow Label is consistent across the flow.

While completing the analysis of the packet capture program output described in section 5.2.4, it was observed that on a number of occasions my patched machine did not consistently set the Flow Label. After examining the connections in question, using another Perl script (see appendix C), it was established that the Flow Label sometimes changed from its properly set Flow Label value to `00000`. These inconsistencies only occurred when my machine sent a RST packet in response to receiving a packet it had no knowledge of.

RSTs are sent when the host receives ACK packets for a connection it does not know anything about, or when a remote host tries to connect to a port that no service is listening on. Hence, it knows nothing of the state of the connection (or lack thereof) so it does not know what to set the Flow Label too (nor should it) so it sets it to `00000`. My machine sent RSTs with a Flow Label of `00000` in response to ACKs that were received after the connection had closed or timed out, so it no longer had the Flow Label state stored.

5.2.7 Using the Flow Label for Stateful Firewalling

Once the setting of the Flow Label is consistent, either set to `00000` or some other value, it is possible to use this information as another piece of state remembered by the firewall. Figure 5.1 shows the flow of traffic between two hosts (A and B) that is inspected by the stateful packet filter. Host A sends a SYN packet to host B to initiate a connection with the Flow Label set. The packet filter creates a new state (using "keep state") for this connection from the source and destination addresses and ports and the Flow Label, to allow the return packets from this connection back in: the "local flow label".

```
add allow ipv6 from hostA to hostB tcpflags SYN keep-state
```

Host B sends back a SYN/ACK packet to host A with its own Flow Label set. This is the final piece of state remembered by the filter: the "remote flow label". Host A sends the final ACK to host B to complete the three way handshake. The firewall checks the Flow Label to make sure it matches the one stored from SYN packet. If it does match, the packet is let through as normal. As with normal stateful packet filters, once the packet matches the state stored in the firewall all subsequent packets in the connection are passed through the firewall.

If the Flow Label set on the final ACK packet does not match the value stored in the local Flow Label part of the connection state, there are a number of options available for the firewall to deal with the packet. It may:

- block packets with a different Flow Label.
- If the Flow Label has changed from 00000 to another value, allow this one change and replace the old Flow Label stored in state with the new one. Then, block packets if the Flow Label changes again.
- Allow the Flow Label to change once, assuming that the label set on the SYN or SYN/ACK (depending on direction) might not have been set correctly. Thereafter, block any packets with a different Flow Label set (for that connection).

5.2.8 Problems with these strategies

Flaws in old implementations will cause problems until the patched code for consistent flows propagates. Each of the different strategies proposed to deal with packets from hosts that inconsistently set the Flow Label are problematic in some way.

Blocking packets that have different Flow Labels set during the same connection is the safest solution. It prevents the possibility of spoofed packets circumventing the the packet filter. On the other hand, there are more hosts that inconsistently set the Flow Label at present. There will be a lot of false positives until the patched code propagates with packets from legitimate hosts being blocked by the filter.

The second option — allowing the Flow Label to change once from `00000` to some other value — provides for the correct handling of packets from un-patched hosts by the packet filter. But, this also means that there is an increased possibility of packets circumventing the packet filter. There is also no provision for packets from hosts running the un-patched version of FreeBSD or NetBSD, where the Flow Label is set to whatever was previously stored in the memory, which may not necessarily be zero.

The final solution proposed covers the correct filtering of packets through the packet filter from all hosts, including packets from un-patched machines. Unfortunately, allowing a change to the Flow Label value stored in state opens up the possibility for attackers to blindly inject packets into the connection by replacing the Flow Label in the final ACK of the 3-way handshake with one of its own choosing, which will block all other legitimate packets from the real host.

It should be noted though, that someone blindly injecting packets into the network and trying to inject their own packets will have to create a packet that matches all the other information stored in state: the source and destination addresses and ports and also be in the expected range of the TCP sequence and acknowledgement fields, which is no mean feat.

Chapter 6

Conclusion and Future Work

While the basic techniques of packet filtering are the same for IPv4 and IPv6, there are a number of challenges posed for the filtering of IPv6 packets due to features of the new version of the protocol; such as the optional extension headers, end-to-end encryption of packets and multiple addresses per interface.

IPv6 opens up the possibility of developing new techniques for firewalling, such as Transient Addressing for Related Processes as proposed by Bellovin and Gleitz. I proposed using the Flow Label field of the IPv6 header to provide another piece of state, in order to make it more difficult for attackers to blindly inject spoofed packets into a connection. If the code to set the Flow Label field is set consistently on all operating systems, it appears that this would be a useful addition to dynamic filtering of packets.

There are a number of tasks suggested by this Masters that I would consider as possible future work. Firstly, I would have liked to write a patch for NetBSD to fix their handling of the flow label by TCP's SYN Cache, as was done for FreeBSD.

The `nmap` program provides the very useful feature of operating system fingerprinting for IPv4, but this has yet to be ported to IPv6. This feature would have made establishing the operating system of the packets that consistently and inconsistently set the flow label much easier. Tests could be formed to establish the operating system based on responses to "Node Information queries". It may also be possible to fingerprint systems based on their treatment of the flow label. It has already been established that OpenBSD and Linux IPv6 implementations do not set the flow label at all, and it is now possible to tell whether hosts are running a patched or a pre-patched version of FreeBSD.

All of the results in Chapter 5 were obtained from analyzing the Flow Label as set by remote connections in response to connections initiated by my machine. It would be interesting to do further research in this area, capturing packets on an IPv6 web server and analyzing the setting of the Flow Label by remote hosts to see if there are other operating systems that have similar bugs to the one found in FreeBSD.

In addition to this, I would also have liked to have integrated code into either IP Filter's or Packet Filter's dynamic IPv6 filter code to remember the Flow Label as another piece of state, and to analyse the success of this implementation.

Finally, there is scope for future work in checking the mutability of the Flow Label as it traverses the Internet. This could be achieved by sending probes with an increasing hop limit, like traceroute, to obtain ICMPv6 Time Exceeded messages. The ICMPv6 RFC[11] states that error messages must return as much of the original packet that caused the error as it can, without exceeding the minimum IPv6 MTU of 1280. Thus it should be possible to write a program that examines the header that is returned in the ICMPv6 message, to make sure that the Flow Label set in the original message was not tampered with en route.

Bibliography

[1] Paul Albitz and Cricket Liu. *DNS and BIND.* O'Reilly, fourth edition, 2001.

[2] Bruce R Babcock. *Unicast Reverse Path Forwarding*, oct 2002. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/uni_rpf.pdf`.

[3] Dan J. Bernstein. SYN Cookies. `http://cr.yp.to/syncookies.html`.

[4] David Borman. TCP-SYN and delayed TCB allocation. `http://www.postel.org/pipermail/end2end-interest/2003-May/003118.html`. Last visited 23/05/2005.

[5] R Braden. *RFC 1122: Requirements for Internet Hosts — Communication Layers*, oct 1989. `http://www.faqs.org/rfcs/rfc1122.html`.

[6] Brian E. Carpenter and Cyndi Jung. *RFC 2529: Transmission of IPv6 Packets over IPv4*, mar 1999. `http://www.faqs.org/rfcs/rfc2529.html`.

[7] Brian E. Carpenter and Keith Moore. *RFC 3056: Connection of IPv6 domains via IPv4 clouds*, feb 2001. `http://www.faqs.org/rfcs/rfc3056.html`.

[8] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, second edition, 1994.

[9] Tim Chown. IPv6 Implications for TCP/UDP Port Scanning. `http://www.ietf.org/internet-drafts/draft-chown-v6ops-port-scanning-implications-00.txt`, oct 2003. Work in Progress.

[10] Cisco. *Unicast Reverse Path Forwarding Enhancements*, sep 2002. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios121/121newft/121t/121t2/rpf_plus.pdf`.

[11] Alex Conta and Stephen Deering. *RFC 2463: Internet Control Message Protocol (ICMPv6) for the IPv6 Specification*, dec 1998. `http://www.faqs.org/rfcs/rfc2463.html`.

[12] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security 2003*, aug 2003. `http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf`.

[13] daemon9. IP-Spoofing Demystified. *Phrack*, 48, sep 1996. `http://www.phrack.org/show.php?p=48&a=14`.

[14] Stephen E Deering, Brian Haberman, Tatuya Jinmei, Erik Nordmark, and Brian D Zill. IPv6 Scoped Address Architecture. `http://www.ietf.org/internet-drafts/draft-ietf-ipv6-scoping-arch-02rc1.txt`, aug 2004. Work in Progress.

[15] Steve Deering. *RFC 1112: Host Extensions for IP Multicasting*, aug 1989. `http://www.faqs.org/rfcs/rfc1112.html`.

[16] Steve Deering and Jeffrey Mogul. *RFC 1191: Path MTU Discovery*, nov 1990. `http://www.faqs.org/rfcs/rfc1191.html`.

[17] Steve E. Deering and Robert M. Hinden. *RFC 2460: Internet Protocol version Six (IPv6)*, dec 1998. `http://www.faqs.org/rfcs/rfc2460.html`.

[18] Francis Dupont and Pekka Savola. RFC 3041 Considered Harmful. `http://www.ietf.org/internet-drafts/draft-dupont-ipv6-rfc3041harmful-05.txt`, jun 2004. Expired.

[19] 2600 Enterprises. 2600: The Hacker Quarterly [online]. `http://www.2600.com`. Last visited 23/05/2005.

[20] Paul Ferguson and Daniel Senie. *RFC 2827: Network Ingress Filtering: Defeating Denial of Service Attacks which employ Source Address spoofing*, may 2000. `http://www.faqs.org/rfcs/rfc2827.html`.

[21] Bob Fink. 6bone Home Page [online]. `http://www.6bone.net/`. Last visited 23/05/2005.

[22] Robert L. Fink and Robert M. Hinden. *RFC 3701: 6Bone (IPv6 Testing Address Allocation) Phaseout*, mar 2004. `http://www.faqs.org/rfcs/rfc3701.html`.

[23] Fyodor. Nmap Fingerprinting [online]. `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`, jun 2004. Work in Progress.

[24] Lars Marius Garshol. BNF and EBNF: what are they and how do they work? [online]. http://www.garshol.priv.no/download/text/bnf.html. Last visited 25/05/2005.

[25] James Glave. WebCom Security Software Failed in Server Attack. *Wired*, dec 1996. http://www.wired.com/news/technology/0,1282,1052,00.html.

[26] Mike Gleason. The Ephemeral Port Range [online]. http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html. Last visited 23/05/2005.

[27] Peter M. Gleitz and Steven M. Bellovin. Transient Addressing for Related Processes: Improved Firewalling using IPv6 and Multiple Addresses per Host. In *USENIX Security 2001*, aug 2001. http://www.research.att.com/~smb/papers/tarp.pdf.

[28] Jun-ichiro (itojun) Hagino and Craig Metz. IPv4-Mapped Addresses on the Wire Considered Harmful. ftp://ftp.itojun.org/pub/paper/draft-itojun-v6ops-v4mapped-harmful-02.txt, oct 2003. Expired.

[29] Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *2002 USENIX Annual Technical Conference*, jun 2002. http://www.benzedrine.cx/pf-paper.pdf.

[30] Robert Hinden and Brian Haberman. Unique Local IPv6 Unicast Addresses. http://www.ietf.org/internet-drafts/draft-ietf-ipv6-unique-local-addr-08.txt, nov 2004. Work in Progress.

[31] Robert M. Hinden and Stephen E. Deering. *RFC 3513: IP Version 6 Addressing Architecture*, apr 2003. http://www.faqs.org/rfcs/rfc3513.html.

[32] Nick Holland and Joel Knight. Pf: The OpenBSD Packet Filter [online]. http://www.openbsd.org/faq/pf. Last visited 23/05/2005.

[33] Ken Hollis. The Rose Fragmentation Attack [online]. http://digital.net/~gandalf/Rose_Frag_Attack_Explained.htm. Last visited 23/05/2005.

[34] Christian Huitema. Teredo: Tunnelling IPv6 over UDP through NATs. http://www.ietf.org/internet-drafts/draft-huitema-v6ops-teredo-04.txt, jan 2005. Work in Progress
Last visited 05/01/2005.

[35] Christian Huitema and Brian Carpenter. *RFC 3879: Depricating Site Local Addresses*, sep 2004. http://www.faqs.org/rfcs/rfc3879.html.

[36] IANA. Internet multicast addresses [online]. http://www.iana.org/assignments/multicast-addresses. Last visited 23/05/2005.

[37] IANA. Internet protocol v4 address space [online]. http://www.iana.org/assignments/ipv4-address-space. Last visited 23/05/2005.

[38] IANA. Internet protocol version 6 address space [online]. http://www.iana.org/assignments/ipv6-address-space. Last visited 23/05/2005.

[39] IANA. Ipv6 global unicast address assignments [online]. http://www.iana.org/assignments/ipv6-unicast-address-assignments. Last visited 23/05/2005.

[40] IANA. Port numbers [online]. http://www.iana.org/assignments/port-numbers. Last visited 23/05/2005.

[41] David B. Johnson and Steve E. Deering. *RFC 2526: Reserved IPv6 Subnet Anycast Addressed*, mar 1999. http://www.faqs.org/rfcs/rfc2526.html.

[42] Sander Jonkers. IPv6 Enabled Sites / IPv6 Accessible Sites [online]. http://www.prik.net/list.html. Last visited 23/05/2005.

[43] Juniper. *Internet Software Configuration Guide, Release 5.6*, dec 2002. http://www.juniper.net/techpubs/software/junos/junos56/swconfig56-interfaces/download/swconfig56-interfaces.pdf.

[44] Christopher A. Kent and Jeffery C. Mogul. Fragmentation considered harmful. In *ACM SIGCOMM 1987*, aug 1987. http://ftp.digital.com/pub/DEC/WRL/research-reports/WRL-TR-87.3.pdf.

[45] Stephen Kent and Randall Atkinson. *RFC 2401: Security Authentication for the Internet Protocol*, nov 1998. http://www.faqs.org/rfcs/rfc2401.html.

[46] Stephen Kent and Randall Atkinson. *RFC 2402: IPv6 Authentication Header*, nov 1998. http://www.faqs.org/rfcs/rfc2402.html.

[47] Joel Knight. PF: Packet Queueing and Prioritization [online]. http://www.openbsd.org/faq/pf/queueing.html. Last visited 23/05/2005.

[48] Douglas Knowles, Frederic Perriot, and Peter Szor. Symantec Security Response - W32.Blaster.Worm [online]. http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html. Last visited 23/05/2005.

[49] Suresh Krishnan. Arrangement of Hop-by-Hop Options. http://www.ietf.org/internet-drafts/draft-krishnan-ipv6-hopbyhop-00.txt, jun 2004. Work in Progress.

[50] Jeffrey P. Lanza. US-CERT Vulnerability Note VU#539363 [online]. http://www.kb.cert.org/vuls/id/539363. Last visited 23/05/2005.

[51] Jonathan Lemon. Resisiting SYN flood DoS attacks with a SYN cache. In *BSDCon 2002*, feb 2002. http://people.freebsd.org/~jlemon/papers/syncache.pdf.

[52] Kurt Eric Lindqvist and Joe Abley. Operation of AnycastServices. ftp://ftp.itojun.org/pub/paper/draft-itojun-v6ops-v4mapped-harmful-02.txt, oct 2004. Work in Progress.

[53] Colm MacCarthaigh. [redbrick-ipv6] Re: [Admin-discuss] Redbrick IPv6 Network [online]. http://lists.redbrick.dcu.ie/pipermail/redbrick-ipv6/2003-December/000032.html. Last visited 23/05/2005.

[54] Rob Malda. Slashdot — IP Tunneling Through Nameservers [online]. http://slashdot.org/article.pl?sid=00/09/10/2230242&mode=thread. Last visited 23/05/2005.

[55] Gregory Maxwell, Remco van Mook, and Martijn van Oosterhout. *Linux 2.4 Advanced Routing HOWTO*, dec 2001. http://www.linuxguruz.com/iptables/howto/2.4routing-13.html#ss13.1.

[56] Orla McGann. Investigation of Internet Protocol version 6. Technical report, Dublin City University, Dept. of Electronic Engineering, apr 2000. http://www.redbrick.dcu.ie/~orly/Finalproj/finalreport.pdf.

[57] Sun Microsystems. *Securing Sun Linux Systems: Part II, Network Security*, sep 2003. http://www.phptr.com/articles/article.asp?p=101181&seqNum=2.

[58] Thomas Narten and Richard Draves. *RFC 3041: Privacy Extensions for Stateless Address Autoconfiguration in IPv6*, jan 2001. http://www.faqs.org/rfcs/rfc3041.html.

[59] Thomas Narten, Richard Draves, and Suresh Krishnan. RFC 3041: Privacy Extensions for Stateless Address Autoconfiguration in IPv6. http://www.ietf.org/internet-drafts/draft-ietf-ipv6-privacy-addrs-v2-02.txt, dec 2004. Work in Progress.

[60] Thomas Narten, Erik Nordmark, and William Allen Simpson. *RFC 2461: Neighbour Discovery for IPv6*, dec 1998. http://www.faqs.org/rfcs/rfc2461.html.

[61] RIPE NCC. IPv6 Allocations [online]. http://www.ripe.net/rs/ipv6/stats/index.html. Last visited 23/05/2005.

[62] RIPE NCC. RIPE NCC: RIPE Region Hostcount [online]. http://www.ripe.net/info/stats/hostcount/. Last visited 23/05/2005.

[63] Jon Postel. *RFC 791: Internet Protocol*. Defense Advanced Research Projects Agency, sep 1981. http://www.faqs.org/rfcs/rfc791.html.

[64] FreeBSD Project. The FreeBSD Project [online]. http://www.freebsd.org. Last visited 23/05/2005.

[65] KAME project. Webpage of Kame Project [online]. http://www.kame.org. Last visited 23/05/2005.

[66] Jarno Rajahalme, Alex Conta, Brian E. Carpenter, and Steve E. Deering. *RFC 3697: IPv6 Flow Label Specification*, mar 2004. http://www.faqs.org/rfcs/rfc3697.html.

[67] Barry Raveendran Greene and Neil Jarvis. *Unicast Reverse Path Forwarding Enhancements for the ISP-ISP Edge, version 1.5*, feb 2001. http://www.cisco.com/public/cons/isp/documents/IOSEssentialsPDF.zip.

[68] Barry Raveendran Greene and Philip Smith. *Cisco ISP Essentials, version 2.9*, jun 2001. http://www.cisco.com/public/cons/isp/documents/IOSEssentialsPDF.zip.

[69] Ronald L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*, apr 1992. http://www.faqs.org/rfcs/rfc1321.html.

[70] Pekka Savola. Security of the IPv6 Routing Header and Home Address Options. http://www.ietf.org/internet-drafts/draft-savola-ipv6-rh-ha-security-03.txt, dec 2002. Work in Progress.

[71] Pekka Savola. Firewalling Considerations for IPv6. http://www.ietf.org/internet-drafts/draft-savola-v6ops-firewalling-02.txt, oct 2003. Work in Progress.

[72] Christoph Schuba. Addressing weakness in the Domain Name System protocol. Master's thesis, Purdue University, aug 1993. http://ftp.cerias.purdue.edu/pub/papers/christoph-schuba/schuba-DNS-msthesis.pdf.

[73] Sergei Shevchenko. Symantec Security Response - W32.Sobig.D@mm [online]. `http://securityresponse.symantec.com/avcenter/venc/data/w32.sobig.d@mm.html`. Last visited 23/05/2005.

[74] Richard W. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley, 1994.

[75] Richard W. Stevens. *Unix Network Programming, Volume 1.* Prentice Hall PTR, second edition, 1998.

[76] Fred Templin. Intra-Site Automatic Tunnel Addressing Protocol (ISATAP). `http://www.ietf.org/internet-drafts/draft-ietf-ngtrans-isatap-06.txt`, oct 2002. Work in Progress.

[77] Susan Thomson and Thomas Narten. *RFC 2462: IPv6 Stateless Address Autoconfiguration*, dec 1998. `http://www.faqs.org/rfcs/rfc2462.html`.

[78] Carnegie Mellon University. Packet Filtering for Firewall Systems [online]. `http://www.cert.org/tech_tips/packet_filtering.html`. Last visited 23/05/2005.

[79] Guido van Rooij. Real Stateful TCP Packet Filtering in IPFilter. In *SANE 2000*, may 2000. `http://www.usenix.org/events/sec01/invitedtalks/rooij.pdf`.

[80] David Watson, Matthew Smart, G. Robert Malan, and Farnam Jahanian. Protocol Scrubbing: Network Security Through Transparent Flow Modification. *IEEE/ACM Transactions on Networking*, 12(2), apr 2004.

[81] Paul A. Watson. Slipping in the Window: TCP Reset attacks. In *CanSecWest/core04 Network Security Training Conference*, apr 2004. `http://cansecwest.com/csw04/csw04-Watson.doc`.

[82] Elizabeth D. Zwicky, Simon Cooper, and Brent D. Chapman. *Building Internet Firewalls.* O'Reilly, 2000.

Appendix A

Nmap fingerprinting example

```
root@oscar $ nmap -O ftp.kame.net

Starting nmap 3.55 ( http://www.insecure.org/nmap/ ) at 2004-12-15 11:49 GMT
Insufficient responses for TCP sequencing (3), OS detection may be less accurate
Interesting ports on orange.kame.net (203.178.141.194):
(The 1608 ports scanned but not shown below are in state:  closed)
PORT STATE SERVICE
19/tcp filtered chargen
21/tcp open ftp
22/tcp open ssh
25/tcp filtered smtp
53/tcp open domain
80/tcp open http
81/tcp filtered hosts2-ns
88/tcp filtered kerberos-sec
110/tcp open pop3
111/tcp filtered rpcbind
135/tcp filtered msrpc
136/tcp filtered profile
137/tcp filtered netbios-ns
```

```
138/tcp filtered netbios-dgm

139/tcp filtered netbios-ssn

389/tcp filtered ldap

443/tcp open https

445/tcp filtered microsoft-ds

513/tcp filtered login

514/tcp filtered shell

593/tcp filtered http-rpc-epmap

827/tcp open unknown

887/tcp open unknown

1214/tcp filtered fasttrack

1503/tcp filtered imtc-mcs

1720/tcp filtered H.323/Q.931

1900/tcp filtered UPnP

2049/tcp filtered nfs

2401/tcp open cvspserver

3128/tcp filtered squid-http

4444/tcp filtered krb524

5190/tcp filtered aol

5999/tcp open ncd-conf

6000/tcp filtered X11

6001/tcp filtered X11:1

6002/tcp filtered X11:2

6003/tcp filtered X11:3

6004/tcp filtered X11:4

6005/tcp filtered X11:5

6006/tcp filtered X11:6

6007/tcp filtered X11:7

6008/tcp filtered X11:8

6009/tcp filtered X11:9

6017/tcp filtered xmail-ctrl

6050/tcp filtered arcserve

6112/tcp filtered dtspc
```

```
6346/tcp filtered gnutella
6666/tcp filtered irc-serv
6667/tcp filtered irc
7597/tcp filtered qaz
22273/tcp open wnn6
31337/tcp filtered Elite
Device type:  general purpose
Running:  FreeBSD 4.X
OS details:  FreeBSD 4.6.2-RELEASE - 4.8-RELEASE, FreeBSD 4.7-RELEASE, FreeBSD 4.8-RELEASE through
4.9-STABLE, FreeBSD 4.8-STABLE - 4.9-PRERELEASE
Uptime 153.095 days (since Thu Jul 15 10:33:46 2004)

Nmap run completed -- 1 IP address (1 host up) scanned in 29.534 seconds
```

Appendix B

Packetgrabbing C Code

```
#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>

#include <net/ethernet.h>
#include <netinet/in.h>
#include <netinet/ip6.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void do_packet(u_char *stuff, const struct pcap_pkthdr *pcap_info, const u_char *pkt);

int
main(int argc, char **argv) {
        char *device;
        pcap_t *pcap_dev;
```

```
char errbuf[PCAP_ERRBUF_SIZE];

/*
 * Check that they told us what interface to use.
 */
if (argc != 2) {
        fprintf(stderr, "usgae: %s interface\n", argv[0]);
        exit(1);
}
device = argv[1];

/*
 * Get pcap started.
 */
if ((pcap_dev = pcap_open_live(device, 1500, 1, 10, errbuf)) == NULL) {
        fprintf(stderr, "pcap_open_live: %s\n", errbuf);
        exit(1);
}

if (pcap_datalink(pcap_dev) != DLT_EN10MB) {
        fprintf(stderr, "%s is not an ethernet device.\n", device);
        pcap_close(pcap_dev);
}

if (pcap_setnonblock(pcap_dev, 0, errbuf) < 0) {
        fprintf(stderr, "pcap_setnonblock: %s\n", errbuf);
        exit(1);
}

setlinebuf(stdout);
setlinebuf(stderr);

/*
 * Loop reading packets.
 */
pcap_loop(pcap_dev, -1, do_packet, NULL);
```

```
        /*
         * There's probably no way to get here, but clean up anyway.
         */
        pcap_close(pcap_dev);
        fprintf(stderr, "Bye!\n");
        exit(0);
}

void do_packet(u_char *stuff, const struct pcap_pkthdr *pcap_info, const u_char *pkt) {
        struct ether_header eh;
        struct ip6_hdr ip6h;
        struct ip6_ext exth;
        struct ip6_frag fragh;
        struct tcphdr tcph;
        struct udphdr udph;
        double t; /*time*/
        unsigned int offset;    /* how far down the packet has been processed*/
        unsigned int type;      /* ethernet type*/
        unsigned int payload_length;
        unsigned int next_hdr;
        unsigned int flow_label;
        unsigned int tcp_src;
        unsigned int tcp_dst;
        unsigned int tcp_flags;
        unsigned int udp_src;
        unsigned int udp_dst;
        char srcaddr[INET6_ADDRSTRLEN], dstaddr[INET6_ADDRSTRLEN];

        offset = 0; /* Start of packet */
        if (pcap_info->caplen < offset + sizeof(eh))
                return;  /* not enough data in the packet for ethernet header*/
        memcpy(&eh, pkt + offset, sizeof(eh));
        offset += sizeof(eh);

        type = ntohs(eh.ether_type);
        if (type != ETHERTYPE_IPV6)
                return;

        if (pcap_info->caplen < offset + sizeof(ip6h))
                return;
        memcpy(&ip6h, pkt + offset, sizeof(ip6h));
        offset += sizeof(ip6h);

        payload_length = ntohs(ip6h.ip6_plen);
        next_hdr = ip6h.ip6_nxt; /* nothing to swap, only 1 byte. */
        /* removes version & traffic class fields */
        flow_label = ntohl(ip6h.ip6_flow & IPV6_FLOWLABEL_MASK);
        inet_ntop(AF_INET6, &ip6h.ip6_src, srcaddr, sizeof(srcaddr));
        inet_ntop(AF_INET6, &ip6h.ip6_dst, dstaddr, sizeof(dstaddr));

        /* get the time the packet was captured at*/
        t = pcap_info->ts.tv_sec + 1e-6 * pcap_info->ts.tv_usec;


more_hdr:
        switch (next_hdr) {
        case IPPROTO_HOPOPTS:
        case IPPROTO_ROUTING:
        case IPPROTO_DSTOPTS:
                if (pcap_info->caplen < offset + sizeof(exth))
                        return;
                memcpy(&exth, pkt + offset, sizeof(exth));
                next_hdr = exth.ip6e_nxt;
                offset += (exth.ip6e_len + 1) * 8;
                goto more_hdr;
        case IPPROTO_FRAGMENT:
                if (pcap_info->caplen < offset + sizeof(fragh))
                        return;
                memcpy(&fragh, pkt + offset, sizeof(fragh));
                if (fragh.ip6f_offlg != 0)
                        return;
                next_hdr = fragh.ip6f_nxt;
                offset += 8;
                goto more_hdr;
        case IPPROTO_TCP:
                /* now we know we have TCP header options*/
                if (pcap_info->caplen < offset + sizeof(tcph))
                        return;         /*make sure there's enough data in packet to include TCP header*/
                memcpy(&tcph, pkt + offset, sizeof(tcph));
                tcp_src = ntohs(tcph.th_sport);         /* source port */
                tcp_dst = ntohs(tcph.th_dport);         /* destination port */

                printf("%f %s %s %05x %u %d %d ", t, srcaddr, dstaddr, flow_label, next_hdr, tcp_src, tcp_dst
                tcp_flags = (tcph.th_flags);
                if(tcp_flags & TH_FIN)
                        printf("FIN ");
                if(tcp_flags & TH_SYN)
                        printf("SYN ");
                if(tcp_flags & TH_RST)
                        printf("RST ");
                if(tcp_flags & TH_PUSH)
                        printf("PSH ");
                if(tcp_flags & TH_ACK)
                        printf("ACK ");
                if(tcp_flags & TH_URG)
                        printf("URG ");
                if(tcp_flags & TH_ECE)
                        printf("ECE ");
                if(tcp_flags & TH_CWR)
                        printf("CWR ");
                printf("\n");
        break;
        case IPPROTO_UDP:
                if (pcap_info->caplen < offset + sizeof(udph))
                        return;         /*Make sure there's enough data in packet to include UDP header*/
                memcpy(&udph, pkt + offset, sizeof(udph));
                udp_src = ntohs(udph.uh_sport);         /*udp source port*/
                udp_dst = ntohs(udph.uh_dport);         /*udp destination port*/


                printf("%f %s %s %05x %u %d %d\n", t, srcaddr, dstaddr, flow_label, next_hdr, udp_src, udp_ds
        break;
        default:
                printf("%f %s %s %05x %u\n", t, srcaddr, dstaddr, flow_label, next_hdr);
                return;
        }

}
```

Appendix C

Flow label Checking Script

```
#!/usr/bin/perl

$tcpstate = 30*60*60;   #TCP connections timeout after 30 minutes
$udpstate = 60*60;  #UDP connections timeout after 1 minute
$icmpstate = 1;     #ICMP timeout after 1 second
$otherstate = 1;    #other protocols

#Read from the standard input into $_. "split" assigns the input from $_
#into the various variables definied, dividing it up along white spaces.
while(<>){
    ($time, $sip, $dip, $flow, $proto, $sp, $dp, $tcpflag1, $tcpflag2) = split;

    #creates our quintuple of info associated with a flow
    $flowtuple = "$sip $dip $proto $sp $dp";



    if($proto eq "6"){      #for TCP
        $timeout = $tcpstate;
    } elsif($proto eq "17"){    #for UDP
        $timeout = $udpstate;
    } elsif($proto eq "58"){    #for ICMP
```

```
        $timeout = $icmpstate;
    } else {            #the rest
        $timeout = $otherstate;
    }

    if(($time - $lastpacket{$flowtuple}) > $timeout){
        delete $flowlabel{$flowtuple};
        delete $lastpacket{$flowtuple};
    }

    #if the flowlabel has already been defined, see if it still matches tuple
    if(defined($flowlabel{$flowtuple})){
        if($flowlabel{$flowtuple} ne $flow){
            print "FlowID does not match tuple. Old:$flowlabel{$flowtuple}, New:$flow, $flowtuple Lastflags:$
            $inconsistent{$sip}++;
        }
    }
    #Time that the last change was made to flow tuple. Want to delete tuple if over a certain time.
    $flowlabel{$flowtuple} = $flow;
    $lastpacket{$flowtuple} = $time;
    if($tcpflag1 eq "RST" || $tcpflag1 eq "FIN" ||
       $tcpflag2 eq "RST" || $tcpflag2 eq "FIN"){
        delete $flowlabel{$flowtuple};
        delete $lastpacket{$flowtuple};
    }
    $lastflags{$flowtuple} = "$tcpflag1 $tcpflag2";
    if ($proto eq "6" && $flow ne "00000") {
        $setpackets{$sip}++;
    }
}

foreach $sip (keys %setpackets) {
    print "$sip\n" if ($inconsistent{$sip} == 0);
}
```