Magdeburger Journal zur Sicherheitsforschung

Gegründet 2011 | ISSN: 2192-4260
Herausgegeben von Stefan Schumacher und Jörg Sambleben
Erschienen im Magdeburger Institut für Sicherheitsforschung

Design and Implementation of an IPv6 Plugin for the Snort Intrusion Detection System

Martin Schütte

This work describes the implementation and use of a preprocessor module for the popular open source Intrusion Detection System Snort that detects attacks against the IPv6 Neighbor Discovery Protocol.

The implementation utilizes the existing preprocessor APIs for the extension of Snort and provides several new IPv6-specific rule options that can be used to define IPv6 related attack signatures. The developed module is aimed at the detection of suspicious activity in local IPv6 networks and can detect misconfigured network elements, as well as malicious activities from attackers on the network.

The plugin's source code is available at https://github.com/mschuett/spp_ipv6.

Keywords: IPv6, Snort, IDS, Intrusion Detection System, Plugin

Citation: Schütte, M. (2013). Design and Implementation of an IPv6 Plugin for the Snort Intrusion Detection System. *Magdeburger Journal zur Sicherheitsforschung*, 2, 409–452. Retrieved December 26, 2013, from http://www.sicherheitsforschung-magdeburg.de/publikationen.html
Version 2014/01/06 22:12

1 IPv6 Features and their Security Implications

IPv6 (the Internet Protocol version 6) is the re-engineered successor of the previous Internet Protocol (version 4, thus IPv4), the common protocol layer of all nodes[1] and networks connected to the Internet.

Some important changes from IPv4 to IPv6 are (based on Deering and Hinden (1998), Frankel et al. (2010) and Hogg and Vyncke (2009)):

- Larger Address Space: With 128 bits an IPv6 address is four times as long as an IPv4 address with 32 bits and allows for vastly more addressable nodes and networks. With IPv4 address exhaustion imminent this is the most significant incentive to deploy IPv6.
- Basic and Extension Headers: The number of fields in the IPv6 header has been reduced to a minimum to make packet processing by intermediate routers more efficient (cf. Figures 1 and 2).
- Multicast: IPv6 puts greater emphasis on multicast addressing, and depends on it for autoconfiguration and neighbor discovery.
- Autoconfiguration and Neighbor Discovery: IPv6 allows network devices to configure their own addresses and routes without manual configuration or additional network services (like DHCP).
- Flow Labels: A new field is included to mark sequences of packets (like TCP streams), which might aid routers with similar handling of a packet stream, for example to implement Quality of Service, without having to read every packet's Hop-by-Hop header or upper layer information.
- IPsec: Support for strong authentication, data integrity and encryption is mandatory for all nodes (in contrast to optional support with IPsec for IPv4). Albeit key management problems generally prevent a widespread use, it provides the basis for secure tunnels and authentication of other protocols (e. g. Mobile IP and OSPF). – In this work IPsec is only discussed as a means to protect autoconfiguration (cf. section 8.2).
- Mobile IPv6: To obtain roaming Internet connectivity for mobile devices, one associates hosts with both a fixed Home Address and a changing Care of Address in foreign nets. The use of IPsec enables a secure binding and tunnelling between these addresses. Like with IPsec this is specified for IPv4 as well, but its IPv6 version makes use of several IPv6 improvements (extension headers and neighbor discovery) and no longer requires special router support. – In this work Mobile IPv6 is not discussed.
- Transition mechanisms: To enable coexistence and interconnectivity of IPv4 and IPv6 nets a number of transition mechanisms are specified and implemented, including dual-stack operation, tunnels and protocol translations. As all of these methods introduce new network paths between nodes, they enable new ways to manipulate routing paths and evade Access Control List (ACL) restrictions. – In this work transition mechanisms are not discussed.

2 Larger Address Space

The main incentive to deploy IPv6 is its larger address space of 128 bits, as opposed to 32 bits in IPv4. These 128 bit addresses are split into a 64 bit subnet prefix and a 64 bit interface identifier (with very few exceptions, e. g. for point-to-point connections; Kohno et al. (2011) and Savola (2003)), so every subnet has 2^{64} addresses for hosts to choose from.

At first sight this seems to prevent remote reconnaissance attacks by network scanning because it is infeasible to scan significant parts of such a large address space. But this is only true if the address allocation algorithm leads to a sparse and pseudo-random distribution across the available address space.

The mandatory algorithm is to derive the interface identifier from the network interface's MAC address in EUI-64 format. The IEEE EUI-64 (EUI for *Extended Unique Identifier*) is a mapping of the 48 bit MAC address into the 64 bit address space for IPv6 interface identifiers. It concatenates the first 24 bits/3 octets of the MAC address, the constant `0xfffe`, and the last 24 bits/3 octets of the MAC address.[2]

The network interface's 48 bit MAC address itself is a concatenation of a 24 bit manufacturer ID (the Organizationally Unique Identifier, OUI) and a 24 bit device specific ID. With currently about 15 000 assigned OUIs (many of which are historic and not present in any current hardware), the actually used partition of the EUI-64 address space can be reduced to well below 2^{40} addresses. So the EUI-64 addresses have considerably less entropy than randomly generated interface identifiers, but still enough to prevent exhaustive scanning.

Even more entropy is gained with randomized in-

1 The established IPv6 terminology is Hogg and Vyncke (2009, p. 4):
- A *node* is any system (computer, router, and so on) that communicates with IPv6.
- A *router* is any Layer 3 device capable of routing and forwarding IPv6 packets.
- A *host* is a node that is a computer or any other access device that is not a router.
- A *packet* is the Layer 3 message sourced from an IPv6 node destined for an IPv6 address.

2 The extra modification used for IPv6 consists of one bit flip which is technically irrelevant but simplifies address management: MAC and EUI-64 addresses have the seventh bit set to indicate local scope addresses and unset for global scope (globally unique addresses). Now if one uses the scheme to define local scope addresses as for serial links or tunnels, these would yield interface identifiers like `fe80::200:0:0:1`, `fe80::200:0:0:2`, etc. – By inverting this bit for IPv6 one can use the same uniform address format and still have simple interface identifiers like `fe80::1`, `fe80::2`, etc. Hinden and Deering (2006, section 2.5.1).

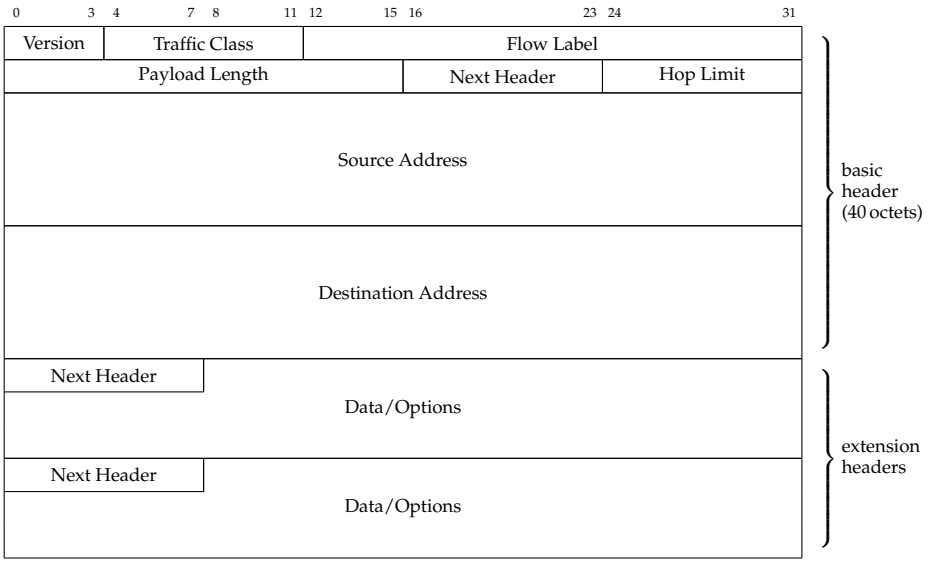


Figure 1: IPv6 header format

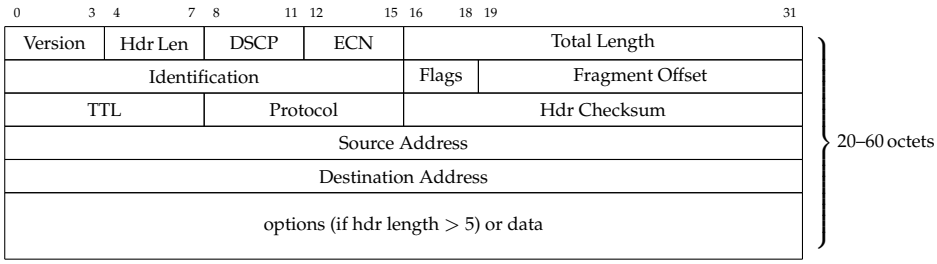


Figure 2: IPv4 header format

terface identifiers, for example when using the privacy extension for stateless address autoconfiguration Narten, Draves et al. (2007) or cryptographically generated addresses (CGAs; Aura (2005)).

On the other hand many networks use a sequential numbering, often due to their DHCP server implementation or because it simplifies manual address management assignment. These addresses have little entropy and it is relatively easy to scan all hosts in such networks Heuse (2010) and Malone (2008).

Other address related security issues might arise from IPv4–IPv6 transition mechanisms. Because IPv4 addresses can also be represented as IPv6 addresses (using IPv4-mapped IPv6 addresses `::ffff:0000/96`, E. B. Davies, Krishnan et al. (2007, Page 2.2)) and the coexistence of two IP versions will lead to many tunneled connections Krishnan, Thaler et al. (2011), multiple opportunities for evasion attacks are created. For the foreseeable future all security devices, ranging from network traffic analysis to firewalls, will have to understand a variety of addressing schemes and encapsulation protocols only to determine the original protocol and source/destination addresses of packets (so they can apply the right restrictions and ACLs).

3 Multicast

The IPv6 addressing architecture defines a hierarchy of multicast addresses for one-to-many communication, ranging from link-local to site-local and global scope (Hinden and Deering (2006, section 2.7); Frankel et al. (2010, section 4.2)). Link-local multicast is used extensively for autoconfiguration, mainly as a more efficient replacement for the broadcast address, which was used in IPv4 and ARP but is no longer defined in IPv6. But having fixed addresses for certain services (e. g. `ff05::101` and `ff05::fb` for site-local NTP and mDNS servers) also simplifies network management and configuration. Because multicast addresses are easily mapped to Media Access Control/link-layer multicast addresses, the link-local scope addresses are always usable whereas addresses with higher scope require the necessary router configuration.

IPv6 routers and hosts use the Multicast Listener Discovery Protocol (MLD, Deering, Fenner et al. (1999) and Vida et al. (2004)) to manage group membership. It uses ICMPv6 and defines the following message types:

- Type 130, *Multicast Listener Query*,
- Type 131, *Multicast Listener Report*,
- Type 132, *Multicast Listener Done*, and
- Type 143, *Version 2 Multicast Listener Report*.

The majority of current layer 2 devices (i. e. Ethernet Switches) implement all multicast messages as broadcast; but some implement MLD Snooping to learn which ports have to receive which multicast destinations. This yields both more efficiency and more se-

curity because it prevents many NDP attacks based on multicast eavesdropping (e. g. the denial-of-service attack against duplicate address detection were no longer possible without receiving all solitextcitednodes multicast messages) or make them much easier detectable (e. g. if an attacker had to join several multicast groups). On the other hand such an implementation has to be robust against flooding; otherwise an attacker could simply fill up the multicast association tables, thus causing a fall-back to broadcasting, and then proceed with one of the conventional attacks.

4 Flow Label

The *Flow Label* field is part of the basic header. So far no standard usage of the flow label has emerged, thus at this time all intended applications are only theoretical and the 20 bit field remains essentially unused.

Proposed use cases include its utilization for Quality of Service (QoS) indication as well as inserting a pseudo-random value to be used for load balancing or as a security nonce to protect against spoofing attacks Hu and Carpenter (2011). The latter case would lead to inclusion of flow labels in packet filter's connection state, thus lowering the chances of packet injection into established "flows" McGann and Malone (2006). On the other hand every stateful of this value would have to be prepared to face denial-of-service attacks.

The IETF working group for *IPv6 Maintenance (6man)* currently discusses a new flow label specification, which also includes a discussion of previously raised security considerations Amante et al. (2011).

5 Extension Headers

IPv6 uses only a small basic header, which is sufficient for most IPv6 packets. In comparison to IPv4 this basic header omits a checksum and fragmentation handling so processing (most importantly routing) packets is simpler and more efficient.

If additional functions are required, a packet is augmented with extension headers. These are supplementary headers that are placed between the IPv6 basic header and the packet's payload (i. e. the upper layer protocol). To make this work the basic header and all extension headers contain a *Next Header* field, and every extension header type is assigned its own protocol number. Using these building blocks, all IP level headers are simply chained one after another by using the *Next Header* field to indicate the protocol number of the following header.

5.1 Extension Header Chaining

An IPv6 packet without extension headers will have the protocol number of its upper layer payload in its *Next Header* field (which is equivalent to the *Protocol* field in IPv4), for example it may use protocol number 17 for UDP. Now if the same packet is encrypted with IPsec then the sender includes an Encapsulated Security Payload (ESP) extension header, which has a protocol number of 50. So the basic header contains a *Next Header* value of 50 to indicate the following ESP header and the ESP header contains a *Next Header* value of 17 to indicate the following UDP payload (see figure 3 for examples).

The Hop-by-Hop Option header has a special status; if it is used it has to be the first extension header because it is read by every router (see also section 5.2). For all other extension headers the IPv6 specification recommends that every header is included only once and in a canonical order. This comes with one exception: in case a Routing header is used then it might be necessary to also include two Destination Option headers – one for the intermediate hosts and one for the final destination. However, the canonical order is a "should"-clause so it is not enforced and in practice packets with any combination of extension headers have to be expected. "IPv6 nodes must accept and attempt to process extension headers in any order and occurring any number of times in the same packet, except for the Hop-by-Hop Options header which is restricted to appear immediately after an IPv6 header only. Nonetheless, it is strongly advised that sources of IPv6 packets adhere to the above recommended order until and unless subsequent specifications revise that recommendation" Deering and Hinden (1998, p. 8).

5.2 Hop-by-Hop and Destination Options

The Hop-by-Hop and the Destination Option header may carry additional options to influence the packet processing. Destination options are examined only by the receiving host, whereas Hop-by-Hop options are also examined by every intermediate router (i. e. *all* nodes along a packet's path). The extension header format is the same for both types and consists of the next header and a length field followed by all options concatenated. All enclosed options are encoded with a type-length-value (TLV) scheme, so the processing node simply iterates through the packet and reads all options one by one.

The option type field has an internal structure of its own: The highest-order two bits encode how nodes have to handle unknown options. A node can either skip the option, silently discard the packet, or discard the packet and reply with an ICMPv6 error message. The third bit indicates whether the option may change during transport Deering and Hinden (1998, section 4.2). This structure enables the use of "optional" options that may be ignored if the processing node does not understand them.

See figure 4 for the TLV schema and an example of a complete extension header including an option and padding

Possible Hop-by-Hop options include Router Alerts and IPv6 Jumbograms. Router Alerts indicate that

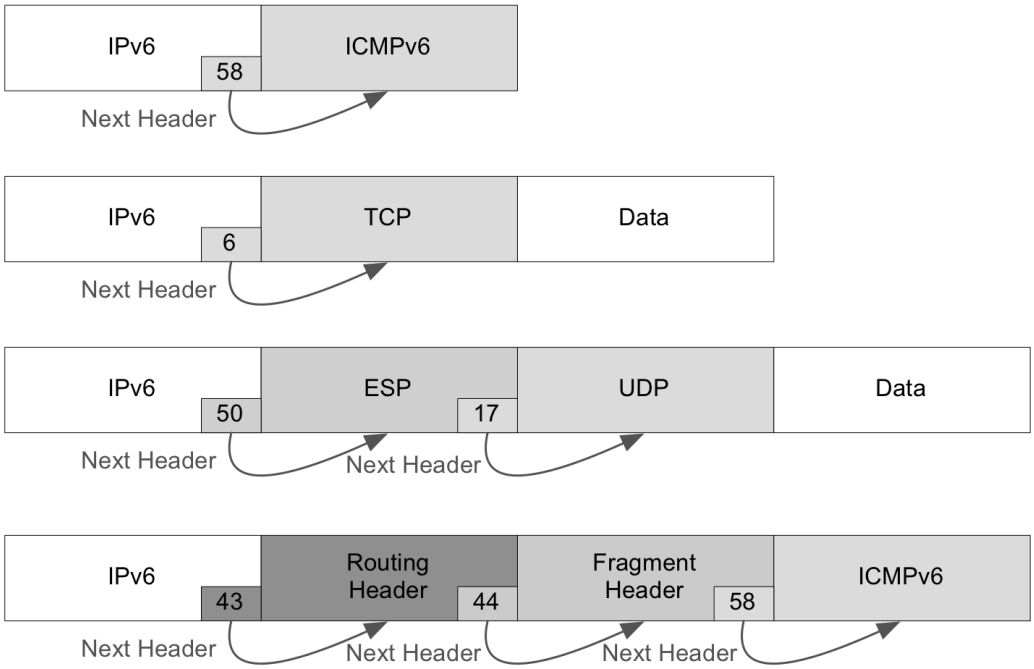
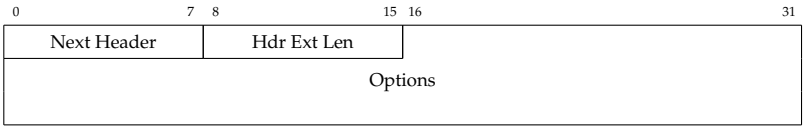
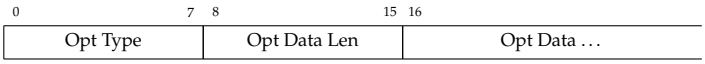


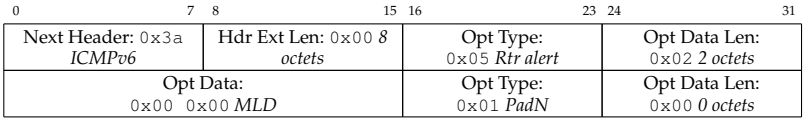
Figure 3: Use of Next Header Field and Extension Headers Biondi (2006, p. 61)



(a) Hop-by-Hop and Destination Header format.



(b) Option TLV encoding.



(c) Example: router alert inside a Hop-by-Hop extension header

Figure 4: TLV encoding for Hop-by-Hop and Destination Options.

the packet's content might have to be processed by routers (albeit the packet is not sent to the router itself). This option is primarily used for Multicast Listener Discovery messages so the local router can keep track of multicast group memberships.

IPv6 Jumbograms Borman et al. (1999) are packets bigger than 65 535 (or 2^{16}-1) octets. The size of packets like that does not fit into the basic header's *Payload Length* field, so it is added as a Hop-by-Hop option instead. This Jumbogram option has a 32 bit length field, so in theory it is possible to send IPv6 packets with 4 Gigabytes (2^{32}-1 octets) of payload. Sending such a packet, possibly using UDP and a wrong checksum to trigger retransmissions, may be considered a denial-of-service attack on its own. In practice the use of IPv6 Jumbograms is largely untested and there is high risk of implementation errors where IP stacks may accept Jumbograms albeit using 16 bit integers to store and process the packet's payload length Frankel et al. (2010, section 4.5).

Finally there are two extra options defined for padding, so a sender can satisfy different alignment requirements: On the one hand the length of every IPv6 extension header has to be an integer multiple of 8 octets (64 bits), and on the other hand multi-byte option values should be sensibly aligned.

The Hop-by-Hop Option extension header might provide great flexibility to adapt new protocol features (like the Jumbogram option), but at the same time it leads to problems because it enables denial-of-service attacks against routers by sending IPv6 packets with many Hop-by-Hop options. To amplify the impact an attacker would use only those option types that have to be ignored by receivers that cannot process them. Then every router along a network path will have to read all TLV encoded options (demanding processing power) and forward the packet to the next hop Krishnan (2011).

5.3 Routing Header

One particular problematic feature of IPv6 was the routing extension header (cf. figure 5) which is basically a re-implementation of IPv4 loose source routing. Sending hosts could add this extension header to include a list (of any length, only limited by MTU) of nodes (both routers and hosts) to be "visited" by a packet along the way to its destination Deering and Hinden (1998, section 4.4).

At first this only lead to security concerns because it allows attackers to evade traffic filtering based on destination addresses and also simplifies reflector attacks Savola (2002). Later it was shown how to abuse the routing header for an amplified denial-of-service attack against a routing path Biondi and Ebalard (2007). Subsequently the use of this type 0 routing header was deprecated Abley et al. (2007) and is now filtered by virtually all routers.

Currently only one other routing extension header is specified (not counting two reserved experimental routing types 253 and 254): the type 2 routing header, used for Mobile IPv6 Johnson et al. (2004, section 6.4). Unlike type 0 this variant is not vulnerable to attacks as it carries only the home address as a single intermediate address and involved nodes have to verify the home address before processing the packet.

5.4 Fragmentation

An essential function of layer 3 protocols is fragmentation of upper layer packets larger than the link-layer maximum transmission unit (MTU). In contrast to IPv4 the fragmentation information in IPv6 is no longer part of the basic header but was moved into an extension header (see figure 6 for the fragmentation header format and figure 7 for the resulting IPv6 packet layout). It is also no longer allowed or required for routers to fragment packets in transit. Instead it is the sender's responsibility to correctly fragment its data. For destinations on-link this is trivial because the host will know the MTU. For remote destinations the intermediate routers check the packet sizes and if if a packet is too big (bigger than the MTU of the link to its next hop) then the sender is notified by ICMPv6 type 2, *Packet Too Big* message, which includes the link MTU that caused the error. In this case the sender will try again by sending smaller messages; and for long routing paths it may take several tries until the sender has determined the path MTU to the destination network. Once it has determined this path MTU, the sender will fragment all subsequent packets to be smaller or equal to this size.

Overlapping fragments are a big concern for IPv4 network security because they enable a variety of attacks and evasion of security measures Novak (2005). Thus IPv6 hosts must never send overlapping fragments and discard received packets with overlapping fragments Krishnan (2009). Nevertheless in practice many implementation (including those in IDS and packet filters) use common fragment reassembly routines for IPv4 and IPv6, thus accepting overlapping fragments, so they are vulnerable to mostly the same fragmentation attacks as with IPv4.

Some other evasion techniques by fragmentation are still possible in IPv6; for example with artificially small fragments (well below 100 octets) and multiple extension headers the upper layer payload, including the next header field, may only start in the second packet – thus preventing protocol or port determination and filtering in intermediate packet filters.

IPv6 mandates a minimum MTU of 1280 octets and Network technologies that cannot process this packet size have to provide their own fragmentation and reassembly on the link-layer Deering and Hinden (1998, section 5). An example for this is the LoWPAN Adaptation Layer to enable IPv6 on IEEE 802.15.4 wireless personal area networks Montenegro et al. (2007). So "there is no reason to have a fragment smaller than 1280 bytes unless the packet is the final fragment and the 'm' more fragments bit is set to '0'. ... To be very secure, one's firewalls should drop all fragments

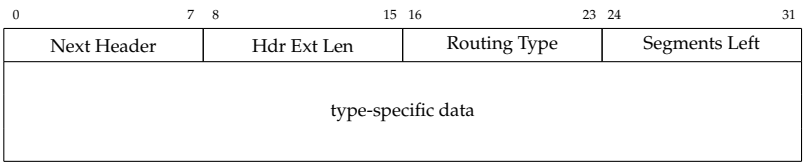


Figure 5: Routing Header format.

that are below a certain size" Hogg and Vyncke (2009, p. 45).

But the difficulty remains to determine the certain size, because the standard does not prohibit smaller fragments and only states that "the lengths of the fragments must be chosen such that the resulting fragment packets fit within the MTU of the path to the packets' destination(s)." Deering and Hinden (1998, p. 20). – So a payload of 2000 octets does not have to be fragmented into fragments of 1280 and 720 octets, but might as well be split into two fragments with 1000 octets each (ignoring headers for simplicity). The use of multiple tunnels or IPv4/IPv6 translation, like the stateless IP/ICMP Translation Algorithm (SIIT; Li et al. (2011)), may also unintentionally reduce a path MTU below 1280.

5.5 Compatibility

The whole concept of extension headers makes IPv6 packet processing more complex and in practice is is difficult to introduce new extension headers, because all hosts involved have to understand all used extension header types. If a receiving host encounters an unknown extension, it cannot process the packet and has to notify the sender with an ICMPv6 type 4, *Parameter Problem* message. Depending on the kind of extension header this may prohibit further communication.

Obviously any packet filter and monitoring device also has to support all extension header types used by any node in its local network. Otherwise it is unable to read the packet's payload and has to either ignore these packets, thus enabling evasion attacks, or drop/reject the packets, thus impairing normal network traffic and revealing the monitoring.

To avoid these backward compatibility problems, it is encouraged to add all new functionality as Destination header options; new extension headers should only be defined if necessary because no existing header is appropriate. It is also proposed that new extension headers should use a uniform format to indicate their header length. This will allow all nodes (and most importantly network monitoring devices) to skip and ignore unknown extension headers but still read following known extension headers and the packet's payload Krishnan, Woodyatt et al. (2011).

6 Neighbor Discovery

As soon as a device is connected to an IPv6 network, it can automatically acquire a unique IP address and obtain all necessary routing information. This autoconfiguration follows the Neighbor Discovery Protocol (NDP) Narten, Nordmark et al. (2007) and Thomson et al. (2007).

The Neighbor Discovery Protocol is based on ICMPv6 messages and defines the following message types:

- Type 133, *Router Solicitation (RS)*,
- Type 134, *Router Advertisement (RA)*,
- Type 135, *Neighbor Solicitation (NS)*,
- Type 136, *Neighbor Advertisement (NA)*, and
- Type 137, *Redirect Message (Redir)*.

Using these messages NDP provides a number of services:

Router Discovery: IPv6 routers send router advertisements to all hosts; both unsolitextcited at regular intervals and upon request when hosts send router solicitation messages. These router advertisement messages contain the basic network configuration, that is the address of the router itself, the subnet prefix, an indication whether clients should use DHCPv6 for configuration, and a lifetime to indicate how long the information is valid. Recent specifications add even more information to IPv6's router advertisements, most importantly a DNS configuration Jeong et al. (2010), but this is not yet implemented in most systems.

Router Redirection: routers can send redirect messages to advise hosts how to use better routes for their packets. This occurs in two cases: If a router receives packets for the same subnet, then it can inform the sender that the destination is on-link and should be addressed directly; or if a subnet has multiple routers and the router determines that it is not on the optimal path, then it can instruct the host to use another first-hop router for some destinations.

Address Autoconfiguration: whenever a host connects to an IPv6 network it will assign itself a link-local IP address and initiate router discovery. The link-local IP is formed by concatenating the link-local subnet prefix (`fe80::/10`) and the modified EUI-64 interface identifier.

By default, i. e. if the router advertisement does not tell it to use DHCPv6, an IPv6 node will use stateless address autoconfiguration (SLAAC) to acquire its global IP address using the concaten-

ation of the global subnet prefix and its interface identifier. But depending on its configuration it might also use other addressing schemes, for example the privacy extensions Narten, Draves et al. (2007) which use a random value as an interface identifier.

Address Resolution: Before any IP (layer 3) communication is possible the sender has to know the link layer (layer 2, e. g. Ethernet) address of the destination host (or of the router, if the destination is not on-link). To resolve this address the hosts uses the IP address to derive the associated solitextcited-node multicast group and sends a neighbor solicitation (NS) to this address. The destination will receive the NS and answer with a neighbor advertisement (NA) that includes its link layer address (cf. figure 9).

The solitextcited-node multicast address used for this mechanism is formed with the subnet prefix `ff02:0:0:0:0:1:ff00:0/104` and the last 24 bits (3 octets) of an IPv6 address. For every one of its unicast and anycast address a host has to join the associated solitextcited-node multicast group. – Thus address resolution becomes more efficient (in comparison to IPv4/ARP using broadcast) because even in large subnets only few hosts have to receive and process the NS message.

Duplicate Address Detection (DAD): Before a host acquires a new IP it verifies that the address is not used by any other host. The mechanism is basically the same as for address resolution, only slightly modified because the requesting host has no IP address yet (cf. figure 10): The host will derive the solitextcited-node multicast group and join it[3]. Then it will send a neighbor solicitation (NS) for the tentative IP (like for address resolution, but it has to use the unspecified address `::` as source IP). If any other node uses this IP address it has to react to the NS by sending a neighbor advertisement (NA) to the solitextcited-node multicast group – so the requesting node will receive the NA even without having an IP. If no host answers to the NS then the IP is assumed to be available and the host will start to use it. The applied timeout is configurable and one second by default. In case of unreliable link layers the hosts should be configurable to send multiple solicitations after several retransmission intervals.

A host might also implement Optimistic DAD, which speeds the algorithm up and allows hosts to use the new address before DAD is completed. It can be used for addresses with very low collision probability like EUI-64 addresses, random values, or DHCPv6 assignments Moore (2006).

Neighbor Unreachability Detection (NUD): As long as IPv6 hosts communicate with each other they regularly verify their peer's reachability. If the upper layer use bidirectional communication (i. e. TCP) that verification is implicit, but if the upper layer protocols are unidirectional then an explicit check is performed by sending a neighbor solicitation message. If a failure is detected a host should start a new address resolution in case the IP address moved to another link layer interface; if the error persists the peer is recognized as unreachable and appropriate errors can be propagated to higher protocol layers.

7 Attacks against the Neighbor Discovery Protocol

With this combination of services it is obvious how important NDP is for reliable network operation because all hosts depend on it for the most basic functions. As basic precaution neighbor discovery messages are only processed on-link (their IP packets have to include a *Hop Limit* of 255) and many attacks require access to link-local multicast messages. But with usually unsecured and unauthenticated layer 2 network access it is equally obvious how vulnerable a local network is to NDP interference by malicious (or misconfigured) nodes on-link. So even though IPv6 is often seen as a re-introduction of the end-to-end principle in network design, the special status of link-local access will continue to require perimeter security.

Possible attacks can be classified by the attacked nodes, either routers or hosts, and the result, which is either a denial-of-service or a man-in-the-middle configuration Chown and Venaas (2011), Ebalard, Combes, Boudguiga et al. (2009), Ebalard, Combes, Charfi et al. (2009) and Nikander et al. (2004).

7.1 Neighbor Solicitation/Neighbor Advertisement Spoofing

These are attacks against the neighbor discovery of normal hosts.

Neighbor cache poisoning: when a host receives an NDP message it will use the message's content to update its neighbor cache (with few exceptions, like neighbor solicitations with unspecified source addresses used for duplicate address detection; also a neighbor advertisement will not cause the creation of a new neighbor cache entry – only the modification of an existing entry).
So an attacker can answer every address resolution with an advertisement message containing the requested IP address and a random link-layer address. The target host will accept the link-layer address, resulting in a denial-of-service for about 30 seconds until the neighbor unreachability de-

3 At first sight this multicast join might be confusing because the host uses the tentative IP address it has not acquired yet to join the multicast group. – With regard to the network this works because the DAD either fails, in this case the original owner of the IP is already part of the multicast group and the join message has no effect, or it succeeds, and in this case the tentative IP address is acquired. If the DAD fails then the host will stop listening to the multicast address; but this change of state is purely local and does not require any additional message.

tection starts (and the attack can be repeated). If the attacker inserts his own link-layer address and answers all following messages accordingly, the result is a man-in-the-middle position.

NUD failure: if a host starts the neighbor unreachability detection (NUD) because a peer no longer responds, then an attacker can send fake neighbor advertisement answers to pretend reachability. This is a subtle denial-of-service attack whose consequences depend on the specific context; in a rather harmless case it will only take longer for the upper layer protocol (i. e. TCP) to detect the connection timeout, while a more severe case would prevent fail-over in high-availability architectures (e. g. if using multiple redundant routers).

DAD DoS: an attacker can listen to neighbor solicitation messages sent for duplicate address detection and respond to them with their own neighbor solicitation (pretending to perform a coincidental second DAD for the same address) or a neighbor advertisement (pretending to already use the IP). This is a denial-of-service situation that prevents hosts from joining the network (or acquiring additional IPs).

7.2 Router Advertisement/Redirection Spoofing

These are attacks against router discovery mechanisms. (In addition to this list RFC 3756 also includes the attack scenario "Good router goes bad" in which an attacker compromises a previously benign and trusted router. This is not listed here because it does not pertain to NDP and the security of connected devices is out of scope here.)

Malicious router: an attacker can simply act as a router by answering router solicitation messages and regularly sending router advertisements; this leads to a man-in-the-middle situation. On its own this is rather unreliable (the host might still receive a benign router advertisement first), so in practice it would be combined with a temporary "kill router" attack.

Kill default router: an attacker has several ways to perform a denial-of-service against a local router. One approach is to send router advertisements with a lifetime of zero, these will case the clients to discard the route; another option is to overload the router, for example with a classic bandwidth denial-of-service attack, or by sending hard to process packets (possibly using hop-by-hop option headers with many options, cf. Krishnan (2011), or packets that require cryptographic verification). If no router is available the hosts will treat all destinations as on-link. So an attacker could additionally use a neighbor cache poisoning to gain a man-in-the-middle position.

Spoofed Redirect: an attacker can spoof a redirect message using the default router's address as source and inserting itself as a better first-hop router to be used for some destination. The result is a man-in-the-middle situation.

Bogus on-link prefix: a spoofed redirect can also indicate that a remote destination is on-link, thus generating a denial-of-service situation because the following address resolution will fail. The approach can be extended with neighbor cache poisoning to gain a man-in-the-middle position. Alternatively an attacker can flood the net with random bogus on-link prefixes, thus performing a bigger denial-of-service attack by filling the host's routing table.

Bogus address configuration prefix: an attacker can send router advertisements with an invalid subnet prefix to perform a denial-of-service attempt against new hosts. New hosts executing stateless address autoconfiguration will use this prefix for their addresses and then will not be able to communicate (they will be able to send packets, but no answers will reach them).

Parameter spoofing: spoofed router advertisement messages can also contain other modified parameters (i. e. other than the router address and the subnet prefix). Exemplary attacks are:

- announcing a low *Cur Hop Limit*, so host will not be able to reach all destinations (denial-of-service);
- unsetting the M/O-bits a DHCPv6-managed net, thus preventing required host configuration (denial-of-service);
- setting the M/O-bits in a net without DHCP server, and then act as rogue DHCPv6 server (man-in-the-middle);
- announcing a random host or oneself as a recursive DNS server (denial-of-service or man-in-the-middle).

7.3 Other

Replay attacks: with plain NDP replay attacks are not an issue because an attacker can generate arbitrary messages anyway. But they have to be considered when adding protocols with cryptographic protection like SEND (which is susceptible to duplicate address detection denial-of-service attacks using replayed messages, Cheneau and Combes (2008)).

Remote NDP DoS: a remote attacker can send messages to many different IPs in a subnet. The subnet's router will have to perform address resolution for every IP and possibly be unable to process local neighbor discovery messages (denial-of-service situation). Hosts might be vulnerable as well if a remote attacker can trick them into sending messages to arbitrary on-link hosts.

8 Security measures to protect the Neighbor Discovery Protocol

Because the vulnerabilities of NDP are well-known, there have been several attempts to strengthen the protocol and prevent attacks.

8.1 Layer 2 filtering and RA-Guard

It should be noted that the early authentication problem is neither new nor unique to IPv6 autoconfiguration, because eventually it is a layer 2 problem.

Thus the Address Resolution Protocol (ARP) used with IPv4 and DHCPv4 have similar vulnerabilities as IPv6 Ramachandran and Nandi (2005). The only difference is that in IPv4 context their relevance has been known for a longer time and many network devices, most notably Ethernet switches, implement mitigation techniques like static ARP entries and DHCP snooping. Similar approaches exist for IPv6 (e. g. MLD snooping, cf. section 3), but they are not as common or mature as their IPv4 predecessors.

Multilayer switches with ACL capabilities can also be statically configured to only accept router advertisements or DHCPv6 server messages on the right ports, thus preventing several man-in-the-middle attacks.

A special case of this filtering is the IPv6 Router Advertisement Guard (RA-Guard, Levy-Abegnoli et al. (2011)) which lets the layer 2 decide whether to pass or drop router announcements (thus it is only applicable where all packets traverse a managed layer 2 device, usually the local switch). This decision can use more or less complex rules, based on different attributes, including the physical port, the source link-layer address, the source IP address, the announced subnet prefix, and SEND-conform signature verification. The implementation might be stateless (with configured access rules) or stateful with automatic discovery during a learning period.

It is also suggested to disallow all IPv6 extension headers for neighbor discovery messages Gont (2011b) to make this filtering effective; otherwise (i. e. with current implementations) it is easy to hide rogue advertisements with extension headers and fragmentation Gont (2011a). So far only few devices support RA-Guard, but it is expected to become more widely available because it is a relatively simple solution with little administrative cost.

One other problem with common layer 2 technology (i. e. Ethernet) is the ability of any sender to use arbitrary source addresses. This gives significant benefits to an attacker: on the one hand it enables several denial-of-service attacks (e. g. flooding the neighbor cache) and on the other hand it minimizes the risk of detection (because even upon detection, it is practically impossible to identify the originating device). – Thus switches should be configured with source address based ingress filtering if possible (i. e. if the switch supports such filters and the network setup is static). Then every switch port only accepts frames from one or more correct link-layer source addresses, attackers are forced to use their real link-layer address (MAC) and detected incidents can at least be tracked to an originating switch port and the thereby connected network device.[4]

A completely closed and secure network will require authenticated layer 2 addresses, as provided by IEEE 802.1X port-based network access control, and a verifiable binding between a host's layer 2 and layer 3 addresses. Strictly speaking this does not solve any layer 3 security problem, but it moves the security perimeter from the network to the hosts, because as long as the involved hosts themselves are not compromised they can rely on each other to behave as intended. In practice this is often not possible, because many networks have to allow public access and not all devices support IEEE 802.1X.

8.2 IPsec

IP Security (IPsec, Kent and Seo (2005)) is a framework to provide security services – including access control, integrity, origin authentication, and confidentiality – at the IP layer (it is specified for IPv4 and IPv6).

On the network level it uses two protocols (in IPv6 added as extension headers): the Authentication Header (AH) to provide integrity, authentication and replay protection for both headers and payload; and the Encapsulating Security Payload (ESP) to provide integrity, authentication, replay protection as well as confidentiality by encryption for the payload only. The main feature of AH, the protection of the IP header, turned out to be a hindrance in IPv4 and NAT environments. Combined with the ability to use ESP with NULL encryption for authentication without encryption, this lead to ESP being used more frequently. – It is still controversial which method should be preferred for different contexts Frankel et al. (2010, section 5.3.6) but the current standard reflects this development by stating "IPsec implementations MUST support ESP and MAY support AH" Kent and Seo (2005, section 3.2).

Both AH and ESP protocols support two modes of operations: transport mode to protect normal end-to-end IP traffic and tunnel mode to protect a point-to-point VPN tunnel (see Figure 12 for the packet formats).

On a policy level every host maintains a database of security associations (SAs) and policies to keep track of all connections, their respective security parameters (i. e. destination IP, used algorithms, and keys), and which security services are used (or required) for different associations. To automatically exchange keys and initiate an IPsec connection an auxiliary protocol,

4 Because the same principle applies to Layer 3 the IETF has an active working group for *Source Address Validation Improvements* (SAVI, http://tools.ietf.org/wg/savi). But to protect neighbor discovery messages a layer 2 filter is necessary, whose presence and implementation is device dependent.

the Internet Key Exchange (IKE), is used to establish the SAs, negotiate necessary parameters (like session keys), and assure mutual authentication.

A general problem of IPsec is its limited support for multicast messages because the whole protocol was designed to protect communication between two end points. Thus the use of IPsec for multicast communication requires key sharing between all receivers of a multicast address, which nullifies IPsec's source authentication and replay prevention, and is also a challenge for key management and configuration (Doraswamy and Harkins (1999, p. 179ff); Frankel et al. (2010, section 5.3.3)). Solving these problems in larger multicast groups (where key sharing is not feasible) necessitates additional protocol extensions and group key management services Hardjono and Weis (2004) and Weis et al. (2008).

One particular drawback for using IPsec and IKE during IPv6 autoconfiguration is a bootstrap problem: the security associations use IP addresses for identification, and IKE uses UDP thus requiring an already established IP address to work. This early authentication problem is profoundly different from normal mutual authentication; it leads to different set of questions, assumptions, and requirements for peer identity and address ownership Nikander (2002). – As a result IPsec can only secure the IPv6 autoconfiguration if multiple SAs are manually set up for every IP in a network on every host (Arkko, Nikander et al. (2003); Frankel et al. (2010, section 5.4.1)).

IPsec (both in IPv4 and IPv6) is normally used in tunnel mode to connect networks (VPN) or in transport mode to secure connections to important servers (e. g. domain controllers and authentication servers that are susceptible to Man in the Middle attacks). But even without considering the early authentication during autoconfiguration, deploying it on all nodes to encrypt all network traffic between them is not practical Hogg and Vyncke (2009, 325ff).

With regard to network security the use of IPsec is also ambivalent: while it defends against a range of network based attacks it also prevents monitoring (and so hides application layer attacks or abnormal traffic). One possibility to minimize the negative implications is to use IPsec just for authentication without encrypting traffic. But this use reveals a problem with ESP-NULL encryption: the transmitted packets do not contain any indication which encryption algorithm is used (this is not necessary because both end points have that information in their security association database). As a result a monitoring device cannot reliably determine whether an IPsec message contains ESP-NULL encryption, it can only use a heuristic approach and always try to interpret the "encrypted" payload Frankel et al. (2010, section 5.3.6).

So far IPv6 specifies mandatory IPsec implementation on all nodes, which is often perceived as a major security feature; but very few nodes are actually configured to use IPsec. Forthcoming IPv6 specification updates will take this into account and downgrade the IPsec support from a MUST to a SHOULD requirement Jankiewicz et al. (2011, section 11.1).

8.3 SEND

SEND, the *Secure Neighbor Discovery* protocol is an approach to secure neighbor discovery (Arkko, Aura et al. 2002; Arkko, Kempf et al. 2005). It is designed for wireless networks (and other "environments where physical security on the link is not assured") and uses two building blocks: cryptographically generated addresses (CGAs) and router authentication.

CGAs are IPv6 addresses whose interface IDs are generated by using a one-way hash function from a RSA public key, the subnet prefix and a security parameter (Aura (2003, 2005); cf. figure 13). This requires a key-pair on every host, but there is no public-key infrastructure to bind keys to specific hosts or certificate verification – so the key-pair can be generated as needed because it is only used to prevent any impersonation (i. e. ICMPv6 spoofing attacks). With SEND the hosts add the CGA parameters and a RSA signature to all neighbor discovery messages. Receiving hosts and routers have to verify the source address (using the parameters to recalculate the one-way hash function) and the signature (using the provided public key) before processing the message.

The router authentication uses a conventional public-key certificate validation: all hosts are to be configured with a trust anchor (e. g. a local CA certificate) and the router have a signed public-key certificate (optionally with limited authorization for only specific subnet prefixes). In addition new ICMPv6 messages are defined to distribute the certification chain to hosts:

- Type 148, *Certification Path Solicitation* and
- Type 149, *Certification Path Advertisement*.

With these preconditions all router advertisements (and redirect messages) are to be signed and the hosts will only accept router advertisements after they successfully verified the router's certificate and the signed advertisements themselves.

As with all cryptographic protections the inevitable downside is a higher demand on the nodes' processing power. While this is insignificant for desktop PCs and servers, low power devices like sensor networks are very restricted in bandwidth, computation and energy resources. Under such constraints it is questionable whether IPsec or SEND are usable for these devices Park et al. (2011); a suggested improvement is the Lightweight Secure Neighbor Discovery Protocol (LSEND) which uses more efficient elliptic curve cryptography (ECC) primitives for CGA and digital signatures Sarikaya et al. (2011). It also enables new denial-of-service attacks against the router that has to verify a potentially large number of signed messages.[5]

5 For example the SENDPEES tool (http://freeworld.thc.org/thc-ipv6/) will flood a target with bogus SEND messages, causing a denial-of-service.

So although SEND could solve most NDP problems, it is still not implemented in major operating systems and thus not currently usable in most environments. It is also incompatible with alternate addressing schemes (i. e. in networks that require privacy extensions for stateless address autoconfiguration, EUI-64 identifiers) its use of message time-stamps does not completely prevent replay attacks Cheneau and Combes (2008), and it allows for new denial-of-service attacks because the required cryptographic verification is computationally expensive.

8.4 DHCPv6

Although IPv6 provides autoconfiguration, it is not a complete replacement for the *Dynamic Host Configuration Protocol* (DHCP) because most implementations only provide clients with an IP address and route information. DHCPv6 has additional options to also announce other services like DNS Jeong et al. (2010) but this functionality is still new and rarely implemented.

Thus in practice DHCP has two advantaged over stateless autoconfiguration (SLAAC): it can provide clients with further information about the network environment (like available name servers, time servers, SIP servers) and it allows the configuration of stable host–IP associations. Because client identification is still based on its MAC address, the difference to stateless address autoconfiguration might be subtle; but with SLAAC the client selects its IP while with DHCP the server decides which IP it assigns, thus enabling a central management.

DHCP is specified both for IPv4 (DHCPv4, Droms (1997)) and IPv6 (DHCPv6, Droms, Bound et al. (2003)). DHCPv4 uses the IPv4 local broadcast address (`255.255.255.255`) for client-server communication: The client sends a `dhcpdiscover` UDP packet with source `0.0.0.0` to destination `255.255.255.255`. And the server replies with a `dhcpoffer` UDP packet, containing the client's MAC address and its designated IP, also to destination `255.255.255.255`. DHCPv6 is more flexible as it uses two steps: the client first obtains a link-local IP and then contacts a server by sending its request to the DHCPv6 multicast group.

To reflect its importance, DHCPv6 has its own place in IPv6 autoconfiguration: The router advertisement contains bitflags, including ones for *managed address configuration* (M-flag) and *other configuration* (O-flag). The O-flag indicates that additional information is available by DHCPv6 (which normal address configuration by autoconfiguration), and the M-flag orders the hosts to use DHCPv6 for all configuration including their IP address Narten, Nordmark et al. (2007, section 4.2).

From a security standpoint the inclusion of DHCP has little impact, because a DHCP server exhibits mostly the same security properties as a router, i. e. it has to be trusted by all hosts and compromising its service is a gateway to all kinds of exploits and denial-of-service attacks against the hosts (Hogg and Vyncke (2009, pp. 208ff); Frankel et al. (2010, section 4.7.3)).

There is work in progress to combine DHCPv6 with CGAs in order to prevent message spoofing attacks Jiang and Shen (2011) and Jiang, Shen and Chown (2011). Yet the security improvement seems to be negligible unless one also provides clients with a trust anchor to authenticate the server.

A more interesting approach is Authenticated DHCP which needs a symmetric key for every client and uses *message authentication codes* to validate messages Droms and Arbaugh (2001) and Droms, Bound et al. (2003). The standard defines two different authentication protocols but only the "delayed authentication" is secure; the other one, which uses a "configuration token" (i. e. a password transmitted in cleartext), is vulnerable to eavesdropping and replay attacks.

Authenticated DHCP is rarely used, because it requires substantial costs for key management and distribution and does not fit with the most common DHCP use case of autoconfiguration in public networks. For centrally managed IPv6 networks the administrative cost would be comparable to deploying IPsec (but every client would only need configuration of one DHCP Unique Identifier and its key instead of a fixed IP and multiple static security associations).

8.5 rafixd & ramond

The RAFIXD[6] tool, written by the KAME project verifies all router advertisements to detect bad prefix announcements (it was originally intended to detect misconfigured 6to4 announcements, but the same principle works for other routing manipulations as well). If it receives a bad router announcement, it sends the same announcement again but with a lifetime of zero. – This should cause all hosts on the network to discard the bad information.

Further developments of this principle include RAMOND[7], written at the University of South Southampton, and NDPMON[8], written at the University of Nancy and INRIA (Beck et al. 2007). The latter program collects neighbor advertisements as well but does not try to reset rogue router advertisements. These tools are, by definition, complete intrusion detection and prevention systems, albeit very specialized and limited ones.

Unfortunately the effectiveness varies, because not all IPv6 stacks behave as intended. So in practice these tools cannot reliably fix the results of configuration errors or routing attacks (an intentional attack could even respond in the same way by resetting the legitimate router advertisements – that way it can no longer gain the man-in-the-middle position but still perform a denial-of-service attack), but their detection ability alone make them very useful for network management.

6 https://github.com/strattg/rafixd v. 2013-12-22

7 http://ramond.sourceforge.net/ v. 2013-12-22

8 http://ndpmon.sourceforge.net/ v. 2013-12-22

9 Conclusion on IPv6 Security

Most IPv6 problems have their roots in layer 2 problems, namely unauthorized link-layer access and no binding between layer 2 and layer 3 addresses. Thus (in wired networks) both the most effective and efficient solution is to use layer 2 filtering. All approaches to solve these issues on the IP layer (layer 3) face an early authentication problem; they loose the "plug and play" property of autoconfiguration because they have to preestablish cryptographically secured identities and trust relationships.

Table 1 gives an overview of neighbor discovery attacks and countermeasures. An additional last column indicates which attacks require an attacker to use their own link-layer source address, so a successful detection of this attack can also identify the responsible host (under normal conditions without filtering).

With the fundamental inability to prevent attacks on basic IPv6 functions in all but the most physically secure network infrastructures, it is only more important to detect them as they occur in networks. The RAFIXD and NDPMON tools show that monitoring the neighbor discovery is a feasible measure to detect and even react to abnormal network configurations. But instead of using custom tools it should rather be explored how to use existing network monitoring applications, namely Intrusion Detection Systems, for this task.

	type	used messages	SEND	IPsec	RA-Guard	auth. DHCPv6	identify src MAC
NS/NA spoofing	MitM	NA, NS	✓	✓(KM)	×	×	✓
NUD failure	DoS	NA, NS	✓	✓(KM)	×	×	×
DAD DoS	DoS	NA, NS	✓	✓(KM)	×	×	×
Malicious router	MitM	RA, RS	✓(PKI)	✓(KM)	✓	(✓)	✓
Kill default router (with RA-reset)	DoS	RA	✓	✓	✓	×	×
Kill default router (with DoS)	DoS	any	×	×	×	×	×
Spoofed Redirect	MitM	Redir	✓	✓	×	×	✓
Bogus on-link prefix	DoS	RA	✓	✓	×	×	×
Bogus SLAAC prefix	DoS	RA	✓	✓	✓	✓	×
Parameter spoofing	DoS	RA	✓	✓	✓	(✓)	×
Remote ND DoS	DoS	NS	n/a	n/a	n/a	n/a	n/a

Table 1: Basic categories of attacks against NDP (based on Nikander et al. (2004)) and appropriate countermeasures to prevent them. A × indicates no protection and a ✓ indicates an effective countermeasure against the specific attack. The deployment of IPsec requires manual key management (KM), and SEND requires either manual key management or a public key infrastructure (PKI).
The last column does not represent countermeasures but indicates whether an attacking node has (✓) or has not (×) to use its real link layer address (assuming no ingress filtering).

10 Intrusion Detection Systems

"*Intrusion detection* is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices. ... An *intrusion detection system* (IDS) is software that automates the intrusion detection process" Scarfone and P. Mell 2007, p. 2-1.

Historically the field of Intrusion Detection Systems stems from two developments: on the one hand the first commercial users of computers had to include these new machines into their financial audit procedures; on the other hand the adoption of computers in the military required trusted systems to process classified information, an effort resulting in the highly influential rainbow series books Bace 2000, chap. 1. Since the 1980s, a large number of different IDSs were developed to increase security of single computers and networks alike. Nowadays, IDSs are a common security measure on all sites that handle sensitive data.

As with all security related techniques an IDS will not "produce" security. Every IDS deployment has to start with a *security policy* and an *acceptable use policy*, because an IDS is only a tool to monitor for adher-

ence or violations of the policy which is encoded in the IDS's configuration (Bechtold and Heinlein 2004, chap. 2.3; Bace 2000, chap. 2.2).

Some elements of security policies are unquestionable or implicit, for example on a host "files in `/sbin` may not be modified" or in a network "MAC addresses may not be forged, and no spoofed ARP or IPv6 neighbor discovery messages may be sent." Others are very site specific, depending on various factors including devices, services, the kind of handled data, and security requirements; for example "all SMTP traffic has to be encrypted" or "connections to IRC servers are not allowed."

The most important characteristic of an IDS is its *detection accuracy* as a combination of *false positives*, i. e. raising a false alarm without actual policy violation (thus imposing higher maintenance cost for manual investigation), and *false negatives*, i. e. not detecting policy violations. The accuracy is usually described using a receiver operating characteristic or ROC curve. Simple percentage values are not meaningful because these do not reflect possible trade-offs between the error types and also because attacks are relatively rare. For example, in a networking context well below 1 % of events might be a policy violation, meaning that even a system that never raises any alarm will automatically have over 99 % accuracy Lazarević et al. 2005, sec. 3.1.

In practice, it is rather difficult to determine an IDS's accuracy under realistic conditions Peter Mell et al. 2003; Zanero 2007. Using real network traffic or system events is infeasible because it is not known which and how many attacks it may contain. On the other hand, constructing a large-scale testbed with a realistic environment (possibly including novel attacks) takes considerable effort.

To date, the most extensive IDS evaluations[9] were performed in 1999 by the Lincoln Laboratory of the Massachusetts Institute of Technology, sponsored by the Department of Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (McHugh 2000). The recorded data was later published as an intrusion detection corpus with several weeks of network traffic and host audit logs, including several hundred documented attacks. This corpus was repeatedly utilized to evaluate and compare IDSs and is still in use today for research in anomaly detection, data mining, and related fields.

11 Taxonomy of Intrusion Detection Systems

Because the term *intrusion detection system* includes a broad variety of approaches and programs, it is usually qualified by additional attributes Bace 2000; Lazarević et al. 2005. Most important distinctions are based on the system's information source, analysis strategy, response capability, and time of analysis.

11.1 Information Source

A *Host IDS* monitors events on one host and uses data sources like applications, filesystems, system log messages, and audit trails. Thus host-based intrusion detection includes all kinds of log analysis both on the system (e. g. syslog events and user login records) and the application layer (e. g. web server access logs). It also includes integrity verification of files as it is implemented for example in the TRIPWIRE[10] or VERIEXEC[11] tools.

Network IDSs on the other hand only use the network traffic as their data source and read all passing data packets. This allows the detection of different events, like attacks on the network stack, and has the advantage of being mostly transparent to users, i. e. invisible on the hosts and not impairing the hosts' performance.

Although these approaches are very different, in some cases they might use the same data sources, like SNMP traps from managed network devices – as these usually include both host based data (e. g. user logins) and network based data (e. g. network traffic metrics or packet drop rates). Some IDS products also use a combination of both sources to correlate more information (for example to supplement network stream information with the communicating application).

11.2 Analysis Strategy

A second dimension is the analysis strategy or detection method: *Misuse detection* encodes possible security policy violations and then tries to match these rules and signatures against all sensor data. This is the approach used by typical anti-virus software that compares all files against a database of virus signatures. This requires previous knowledge of attack vectors, but it also enables wide collaboration and sharing of signatures (e. g. once a new vulnerability is found, one can write and distribute a signature to detect attacks using it).

The opposite approach is *anomaly detection* where the IDS has some conception of "normal system use" and monitors all events for deviations from this baseline profile. The premise for this approach is an adequate system model for statistical analysis (i. e. all events can be represented and quantified) and a sufficiently clear distinction between normal use and undesired misuse (even the collection of "clean" training data can be a problem). In addition the model should be adaptive to changes in normal use patterns. Thus, it

9 http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/ v. 2013-12-22

10 Gene H. Kim and Eugene H. Spafford, 1995, The Design and Implementation of Tripwire: A File System Integrity Checker, in *Proceedings of the 2nd ACM Conference on Computer and communications security*, 18–29, http://doi.acm.org/10.1145/191177.191183

11 Brett Lymn, 2000, *Verified Executables for NetBSD*, http://www.users.on.net/~blymn/veriexec/

would be difficult to use anomaly detection to analyze a site's network traffic if it includes heterogeneous systems and uses (e. g. desktop machines and servers); but anomaly detection is successfully used for specific domains, e. g. to monitor user behavior (with events like login times and used programs).

11.3 Response Capability

Another attribute is the capability to react to detected events. Every IDS supports a *passive response*, which simply means the IDS records useful data and generates events or alarms to notify operators about detected incidents.

All actions beyond this are considered *active responses* and the systems implementing them are called *intrusion prevention systems* (IPS) or *intrusion detection and prevention systems* (IDPS). The most ambitious ideas try to implement automatic real-time countermeasures or even "counterattacks" to cope with the speed and frequency of attempted network attacks. In most cases however, this is not a realistic approach because such automatisms would be susceptible to spoofing, could cause more harm than good, and become part of denial-of-service attacks themselves. Relatively simple response actions are to block attacks by adding packet filter rules or by sending TCP reset packets are used by many systems, but even these could result in a denial-of-service if they are misdirected due to spoofing, or triggered by false positive detection errors. Thus, the most useful actions are often the most harmless ones, for example, the system could collect additional data to help later investigation of the incident.

A subclass of active response capability is the so called *inline mode* found in Network IDPS in combination with packet filters. In this setup all network traffic passes through a network sensor acting as a security gateway (common deployments implement this as part of a layer 3 router or as part of a layer 2 bridge). Such a setup enables the IDPS to perform additional filtering actions like passing, dropping, or rejecting single packets. It can also rewrite packet's content on the fly, for example to perform a normalization of all packets, or remove malicious content after detection.

In the field of Network IDPSs, this latter case is sometimes considered as a third class of response capability (e. g. in Scarfone and P. Mell 2007). But the principle can be found in other kinds of IDS as well: For example, on-access virus scanners or VERIEXEC show the same behavior by checking a resource and then deciding whether to grant or deny access to it.

11.4 Time Aspects

The final technical property is the time of data analysis. IDSs can process their data either in *real time (online)* or in a *batch mode (off-line)*. Curiously, most early IDSs worked in batch mode because limited memory and processing bandwidth did not allow for real-time monitoring, whereas current hardware supports real-time Network IDSs but limited disk I/O bandwidth makes it infeasible to record all traffic passing a 1G or 10G network link for off-line analysis.

11.5 Operation

One last distinction is the project size and the range of use. On the one hand, there are numerous academic projects with few users. On the other hand there is a relatively small number of mature products with a big number of installations. While the academic projects usually implement very specialized IDSs based on new concepts or for new environments, the bigger systems aim for more coverage (to monitor multiple protocols, network layers, and hosts) and often create their own ecosystem by enabling plugins and thus becoming the basis for subprojects.

12 Architecture of Network Intrusion Detection Systems

All IDS solutions require the following parts to provide basic functions, even though the actual module naming, implementation, and boundaries may differ. Figure 14 depicts a general IDS architecture.

12.1 Network sensors

One or more network sensors are required to get data from the network. It is a common setup for small networks to connect an IDS to the switch's monitor port to have access to all packets on the network. Likewise, an IDPS can run on a packet filter for in-band detection and prevention. Larger and more complex networks often require multiple sensors, e. g. before and behind packet filters or in every subnet. Some designs even allow for network and host sensors, e. g. to correlate network traffic with running programs.

With continuously rising network data rates, the packet capturing performance is still a critical factor Braun et al. 2010. To increase efficiency, capturing is normally implemented in the operating system's kernel (but accessible from user space, for example with the the widely used LIBPCAP[12] library,) and allows for immediate filtering of captured packets (McCanne and Jacobson 1993). So if resources are limited and the IDS cannot analyze all network traffic, packets can be ignored, for example with an FTP or media streaming server, it would be sensible to monitor all control connections but ignore the data connections.

12.2 Decoder

The decoder reads the packets, checks their protocol format and checksums (just like a receiving host would), and usually copies a packet's attributes (like

12 http://www.tcpdump.org/ v. 2013-12-22

protocols, addresses, port numbers) into an internal data structure.

This stage checks the protocol formats and thus might already trigger events and drop packets, e. g. those with incorrect checksums.

12.3 Preprocessing

The preprocessing stage contain different functions that process single packets (usually from low-level protocols) and transform them into more abstract events to allow signatures and rules to operate on the application layer.

The most important functions are IP (Layer 3) defragmentation and TCP (Layer 4) stream reassembly. Other common modules normalize packets, detect portscans and decode most important application layer protocols (HTTP, SMTP, DNS, etc).

12.4 Detection

The heart of every IDS is the detection process itself. Generally speaking all packets and events are checked against the configured set of policies and rules at this stage. Finding a good optimization algorithm and applying only relevant rules is a decisive factor to achieve high throughput.

12.5 Output

When detecting an intrusion, the IDS has to record the incident.

The simplest types of recording are writing syslog and similar logfiles, but only the smallest installations can be adequately monitored with textual logfiles only. So most products will not only write log messages but also preserve the relevant network traffic (from single packets to complete application sessions) not only in text files but using multiple output channels including SQL databases.

For IDSs with *active response* mechanisms, their actions (and an event log thereof) are also part of the output.

12.6 Other

Besides the previously described parts constituting the IDS itself, a usual setup will also include auxiliary programs for signature management, log/event archiving, and log analysis.

Because systems use a large number of signatures and also require regular signature updates, they often include management tools e. g. to select the right signature groups or automate signature updates.

Equally important for maintenance are tools for monitoring alerts and accessing collected log data, both to investigate attacks and to eliminate false positives. Such tools should provide comprehensive status information about current detection events, prepare filters and correlations to find bigger patterns, and give access to all available details for incidents under analysis (e. g. see Figure 16 for a screenshot of the Snorby[13] IDS front-end).

13 Example Open Source Network Intrusion Detection Systems

Currently, there are three widely-used open source IDS projects: Bro, Snort, and Suricata.

Proprietary IDS solutions are not considered here because they are usually not extensible by self-written plugins, nor is it easy to determine and verify their actual detection capabilities.

13.1 Snort

SNORT[14] was created in 1998 by Martin Roesch and is now maintained by his company Sourcefire Inc. (Roesch 1999). It is probably the most widely spread open source IDS and provides multiple interfaces for third-party rules and plugins. One of its most successful features is its rule language, which allows everyone to define own signatures in a relatively simple plain-text format.

Its currently stable version is 2.9.1 (released in August 2011). Basic IPv6 support was added in version 2.8. Following versions added a few checks to the decoder (e. g. resulting in decoder alerts for packets with a multicast source address) and the ability to normalize IPv6 packets. In regard to detection signatures the developers decided to pass all IPv6 values the same way as IPv4 values to the detection engine, thus a signature for the IPv4 time-to-live (`ttl: 100`) will also match an IPv6 hop limit. This has the advantage that all existing signatures continue to work well on IPv6 but has the disadvantage that signatures cannot distinguish between IPv4 and IPv6 packets.

13.2 Bro

The BRO[15] project was written in 1998 by Vern Paxson at the UCB (Paxson 1999). Bro IDS emphasizes a clean distinction between the decoding stage, implemented in C, and the analysis and alarm generation, which is implemented in a domain-specific language called "bro script". This offers great flexibility but also requires everyone to use "bro script" to implement new policies.

The currently stable release is version 1.5.3 (released in March 2011). Basic IPv6 support at the decoder level exists since version 0.8 from 2003, but as in Snort, the policy engine receives the same information as for IPv4 packets and there are no default policies for IPv6-specific patterns.

13 http://snorby.org/ v. 2013-12-22

14 http://www.snort.org/ v. 2013-12-22

15 http://www.bro-ids.org/ v. 2013-12-22

13.3 Suricata

The youngest project is SURICATA[16]: It was founded in 2009 with the creation of the Open Information Security Foundation. Its goal is to create an IDS that is backward compatible to existing Snort rule sets, but not limited by Snort's development and architecture history. Thus, it places emphasis on new and experimental features, most importantly parallelism and multicore support.

Version 1.0 was released in early 2011 (with the current stable version 1.0.5 released in July 2011). IPv6 was supported from the start but similar to Snort's implementation, the support is limited to the decoder stage and IPv6-specific information is not passed to the detection routines.

13.4 Other Tools

There are also some specialized programs to track IPv6 neighbor discovery, most notably those described in section 8.5 (RAFIXD, RAMOND, and NDPMON). These tools monitor IPv6 autoconfiguration messages and perform configurable actions when new routers or hosts join the network.

By definition these tools are small but complete Network IDSs by themselves. In practice, however, it is preferred to integrate their functionality into bigger IDS applications. Advantages of such a centralized approach are ease of use, maintainability, and integration into existing log analysis and incident handling procedures.

13.5 Conclusion

The use of Intrusion Detection Systems is a mature and established method to discover, and in some cases also to prevent, security violations. In the special field of Network Intrusion Detection there are several widely used open source products with basic IPv6 support in their decoder modules, but without IPv6-specific detection routines.

Given that none of the available open source IDS's are mature enough in their IPv6 support in order to detect the most commonly discussed attacks against IPv6 neighbor discovery, the best course of action is to modify and extend one of these IDS applications.

Snort turns out to be the best basis for a custom plugin, not only because it is currently the most widely used open source IDS, but even more so because it offers dynamic plugin APIs to develop and deploy plugins without the need to patch and recompile the complete IDS application.

16 http://openinfosecfoundation.org/ v. 2013-12-22

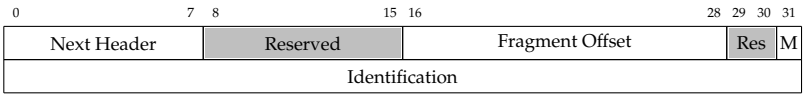


Figure 6: Fragment Header format (M is the “more fragments”-bit).

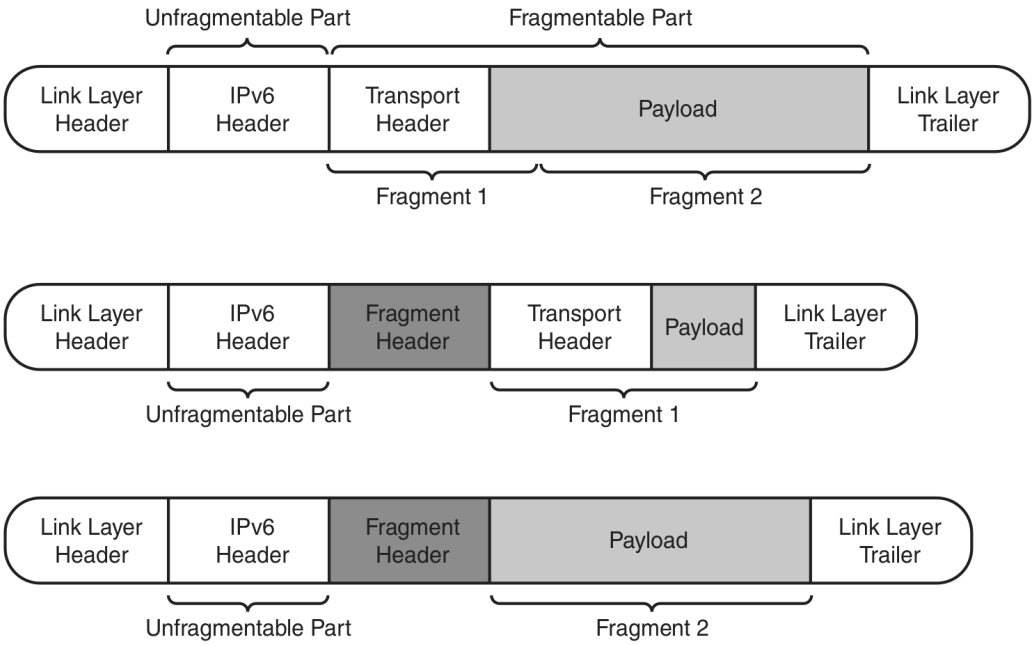


Figure 7: IPv6 Fragmentation Hogg and Vyncke (2009, p. 44)

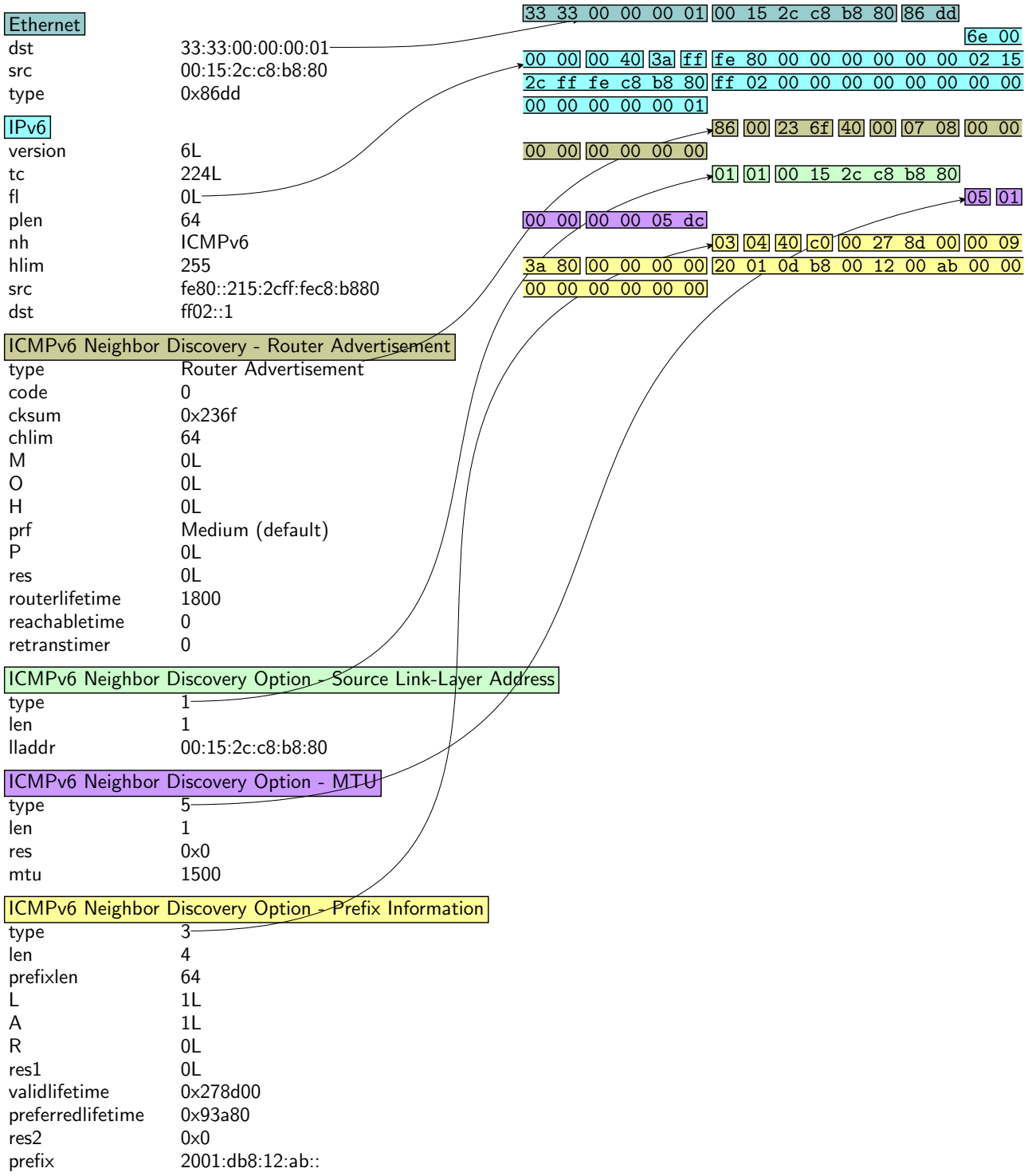


Figure 8: Detailed example of a router advertisement message, including options for the router's link-layer address, the subnet's MTU, and the subnet prefix for stateless address autoconfiguration.

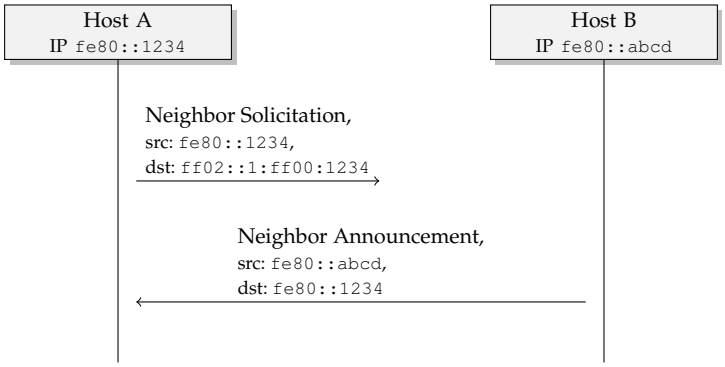


Figure 9: Address Resolution. Host A uses the solitextcited node multicast address to query Host B's layer 2 address; then Host B responds with a neighbor advertisement, including its layer 2 address in a Destination Link-Layer Address option (like shown in figure 11).



Figure 10: Duplicate Address Detection with collision. After receiving the Neighbor Advertisement the new host recognizes the collision, chooses another IP, and repeats the DAD.

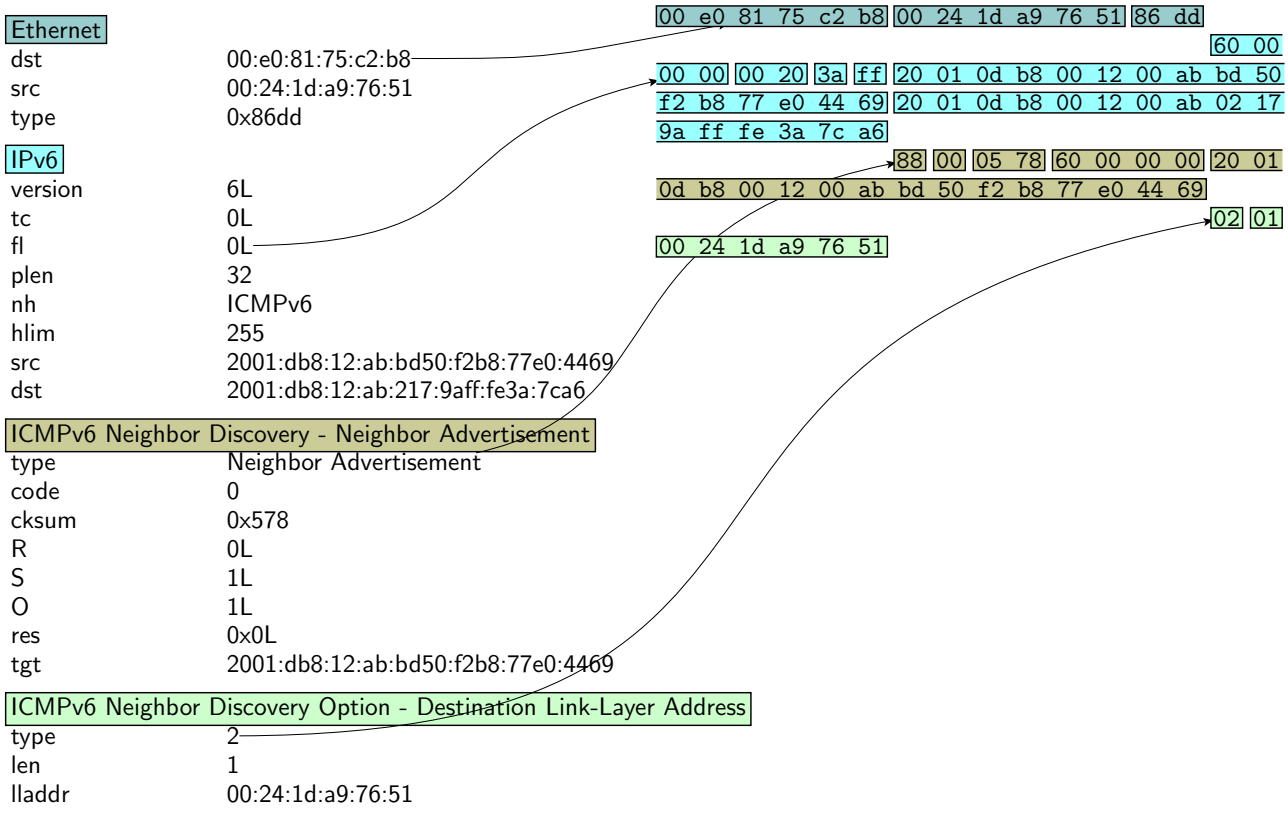


Figure 11: Detailed example of a neighbor advertisement message. It is sent to a unicast address and has a set S-flag to indicate it is a solitextcited advertisement; thus it is a reply to an address resolution request.

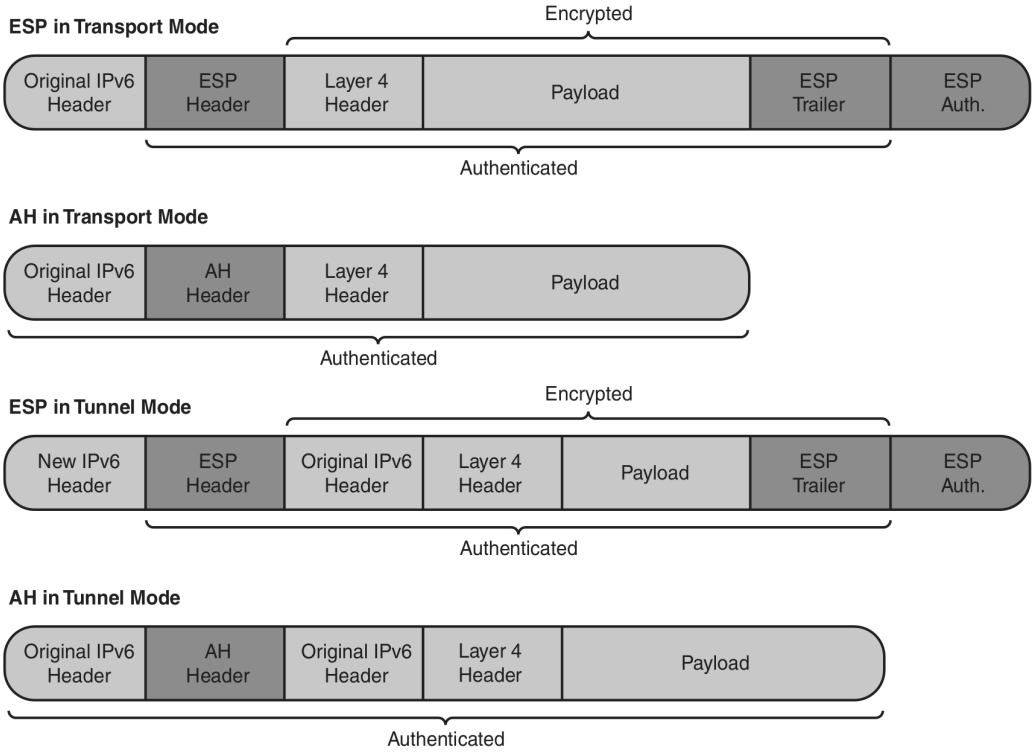


Figure 12: IPsec Packet Formats Hogg and Vyncke (2009, p. 323)

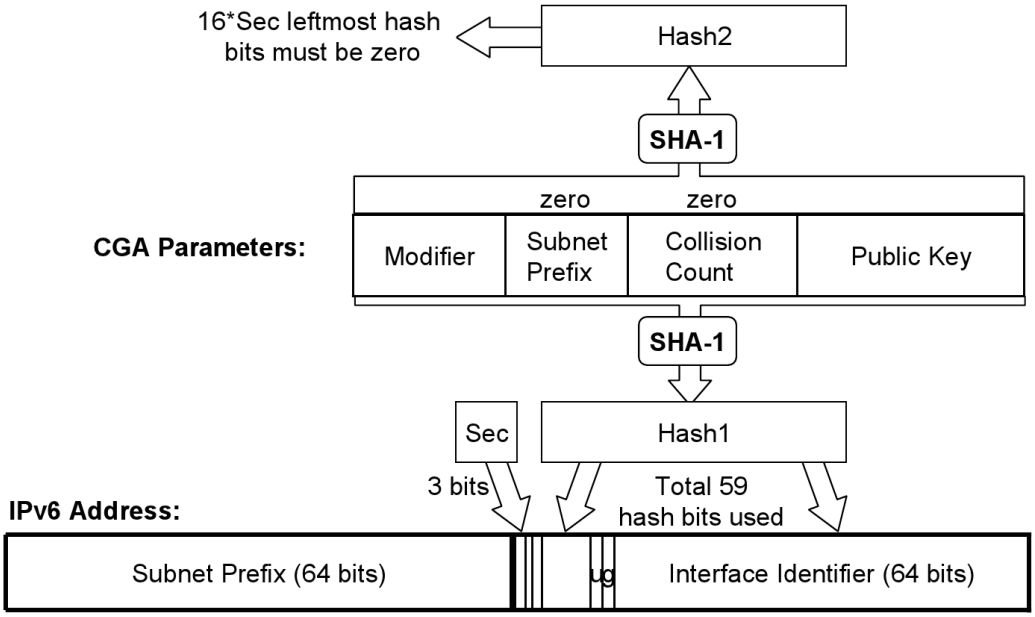


Figure 13: Composition of Cryptographically Generated Addresses Aura (2003, p. 34)

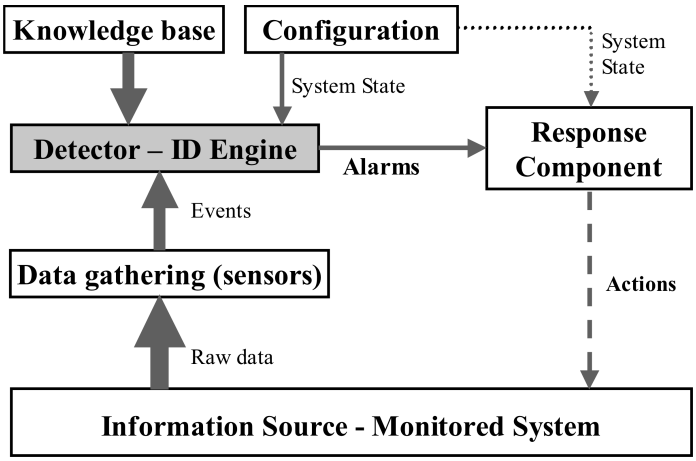


Figure 14: Basic architecture of an IDS Lazarević et al. 2005.

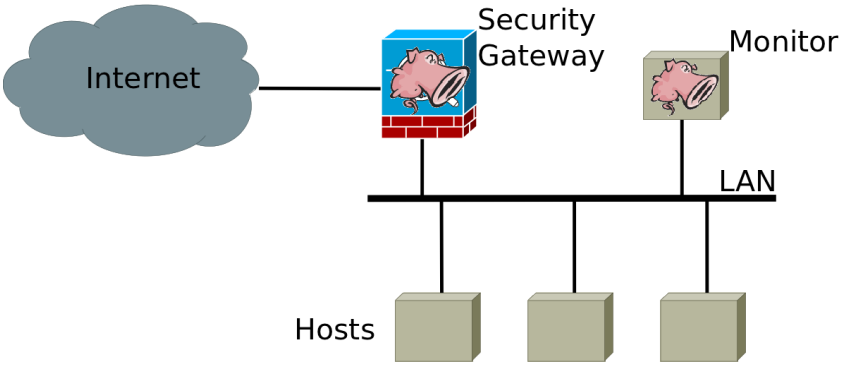


Figure 15: IDS placement options for pure monitoring or additional packet filtering.

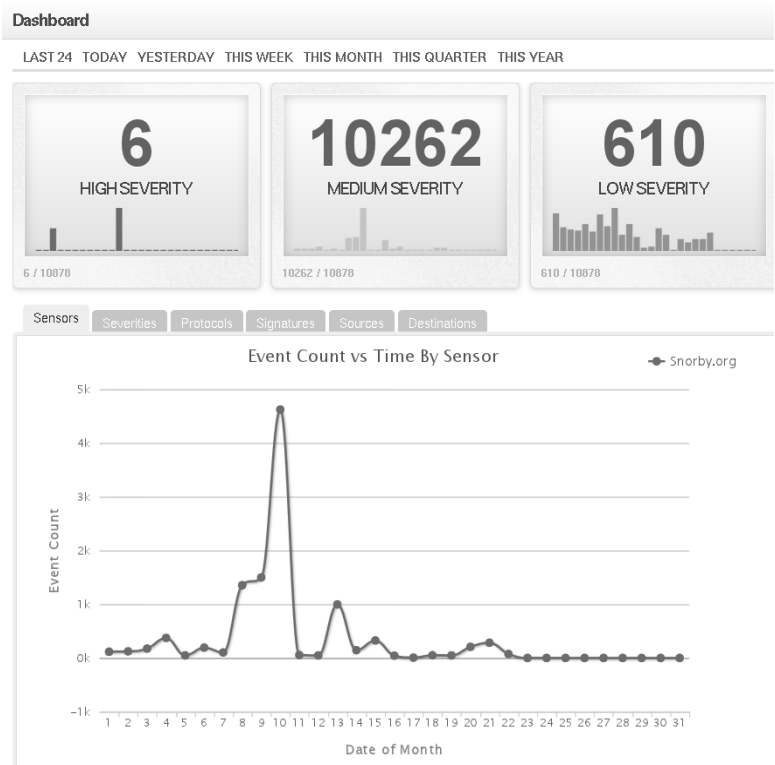
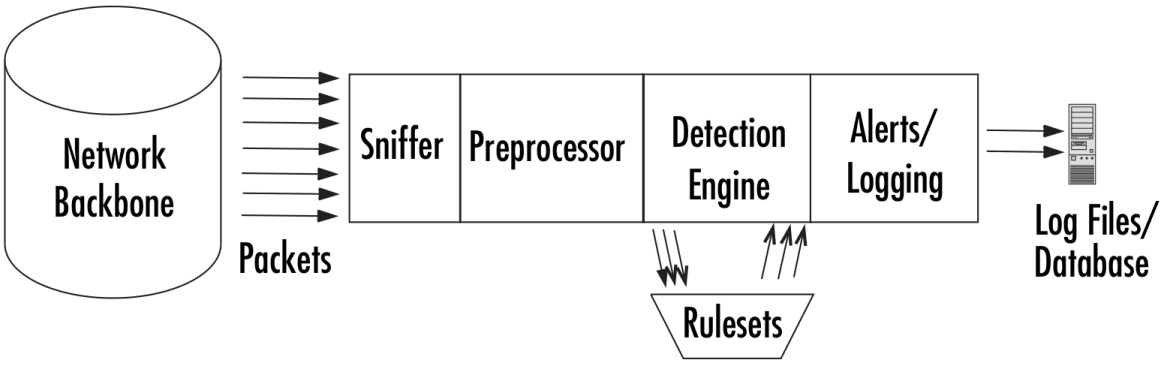
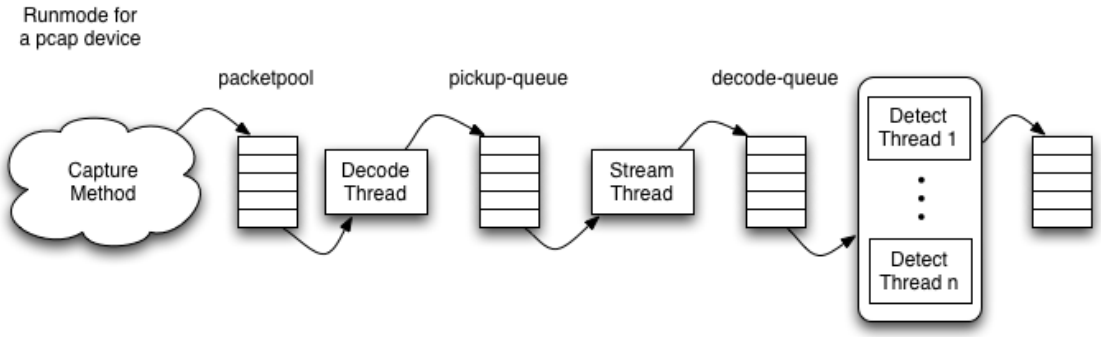


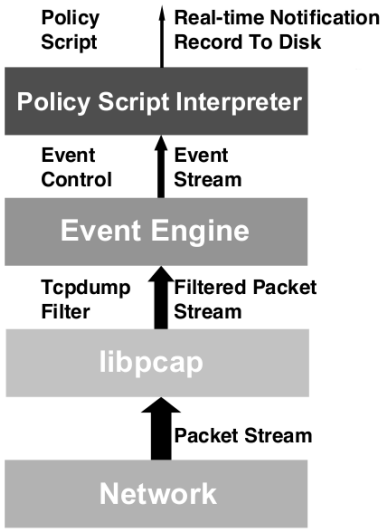
Figure 16: Example of an IDS log data summary using the Snorby (http://snorby.org/) web interface.



(a) Snort Beale, Baker, Caswell et al. 2004, p. 40



(b) Suricata (http://openinfosecfoundation.org/)



(c) Bro (http://www.bro-ids.org/)

Figure 17: Schematic data flow in different Open Source IDS

14 Snort Architecture

Two aspects are important to evaluate Snort as a development framework: its stages of packet processing and its plugin APIs. The former should give insight on the general program structure and potential to adopt new protocols and modes of operation. The latter should determine the potential for custom plugins, namely what information and services a plugin can use and what actions are available to influence the packet processing Beale, Baker, Esler et al. 2007; Bechtold and Heinlein 2004; Olney 2008; Roesch 1999.

Snort processes its data single-threaded in five stages: network packets are acquired, decoded, preprocessed, rules are applied, and alerts or logs are written (see figure 18). Snort can use threads to parallelize configuration reloading (which is important in an IDPS configuration that should have minimal downtime). However, all stages of packet processing are run sequentially in a single thread of execution.

Except for the capturing stage, all options and settings are given in one configuration file: `snort.conf`. Its syntax allows for inclusion of other files, so the rule set can be distributed across multiple files (usually it is organized by protocol or classification).

14.1 Data Acquisition/Packet Capturing

With the release of version 2.9.0 the packet capturing was moved into a separate library, *Data AcQuisition library* (DAQ/LIBDAQ). This library provides a small API towards Snort and encapsulates all (often system dependent) packet capturing code.

Notable capturing modules are *pcap*, which provides access to LIBPCAP, *nfq* to interact with Linux's IPTABLES packet filter, and *ipfw* for the BSD IPFW packet filter. The *pcap* module is particularly useful for development because it can read a previously prepared PCAP file, whereas modules like *nfq* and *ipfw* are required for inline mode operation.

As one would expect, Snort's main control structure is built around a central event-loop (or `PacketLoop()`) that reads network packets from the DAQ layer (in `DAQ_Acquire()`).

14.2 Decoding

Once a packet is read from the network, its decoding follows the network protocol stack. Depending on the currently used DAQ module an appropriate layer 2 decoder is used, e. g. `DecodeEthPkt()` for Ethernet.

The L2 decoders in turn calls the L3 decoders, most importantly `DecodeIP()` (for IPv4) and `DecodeIPv6()`, whereas these will pass the L3 payload to L4 decoders: `DecodeTCP()`, `DecodeUDP()`, `DecodeICMP()`, `DecodeICMP6()` (cf. figure 19).

To monitor other networks than Ethernet Snort also includes L2 decoders for IEEE 802.11/WiFi, Token Ring, FDDI, and Cisco HDLC, as well as decoders for MPLS packets and the special *pcap* link layer types LinuxSLL ("cooked sockets") and OpenBSD PF log. This modular design should make it relatively easy to add new decoders for other protocols.

Likewise the L3 and L4 decoders support a number of encapsulation protocols so Snort can decode IP-in-IP, GRE and Teredo packets; for example when using a simple encapsulation with Ethernet – IPv4 – GRE – IPv6. However, Snort can only process one level of encapsulation; the reason for this limitation is the `struct Packet` data structure, (cf. figure 20) which only provides fields for one "outer" and one "inner" packet.

14.3 Preprocessor

The preprocessor stage includes modules for defragmentation, stream reassembly, portscan detection, and a number of application layer protocols such as HTTP, SMTP, DNS. This is also the first layer with a dynamic plugin API; so it is possible to provide third-party preprocessors as dynamically linked libraries.

Snort preprocessors may have three functions: They implement their own checks to trigger alerts, they normalize data to simplify detection rules, or they provide new rule options for the detection layer. For example the included *ssl* preprocessor will trigger alerts on certain SSL error messages, and it also provides the `ssl_state` and `ssl_type` rule options; as another example the *http_inspect* preprocessor implements preprocessor alerts, rule options, and additionally normalizes HTTP-specific data such as URIs.

The calling order is primarily determined by the preprocessors, as they register themselves for one of several predefined stages (network, normalization, application, etc.). This serves as a simple but sufficient dependency control, for example it letts all application layer preprocessors (like *ssl* and *http_inspect*) be called after fragment and stream reassembly (*frag3* and *stream5*).

A more detailed description of the preprocessor API follows in section 15.2.

14.4 Rule Engine

The core of Snort's processing is the evaluation of configured rules. Its simple rule notation (cf. Figures 21) allows developers and users to easily configure the IDS policy.

Because IDS installations will need a large number of rules (the official Snort rule sets contain over 15 000 rules with 4 000 of them active by default) their efficient rule application is crucial for system performance and observable network bandwidth. This has led to highly optimized data structures and evaluation algorithms including bucketing all rules into port groups (by protocol and source/destination

```
SnortMain
  PacketLoop
    DAQ_Acquire --> pcap ...
      PacketCallback
        ProcessPacket {
          DecodeEthPkt   --> DecodeIPV6 --> ...
          Preprocess     --> [loop through all active PPs]
          Detect         --> [apply rules]
          SnortEventqLog --> [output alert/log events]
        }
```

Figure 18: Schematic outline of Snort's packet processing loop

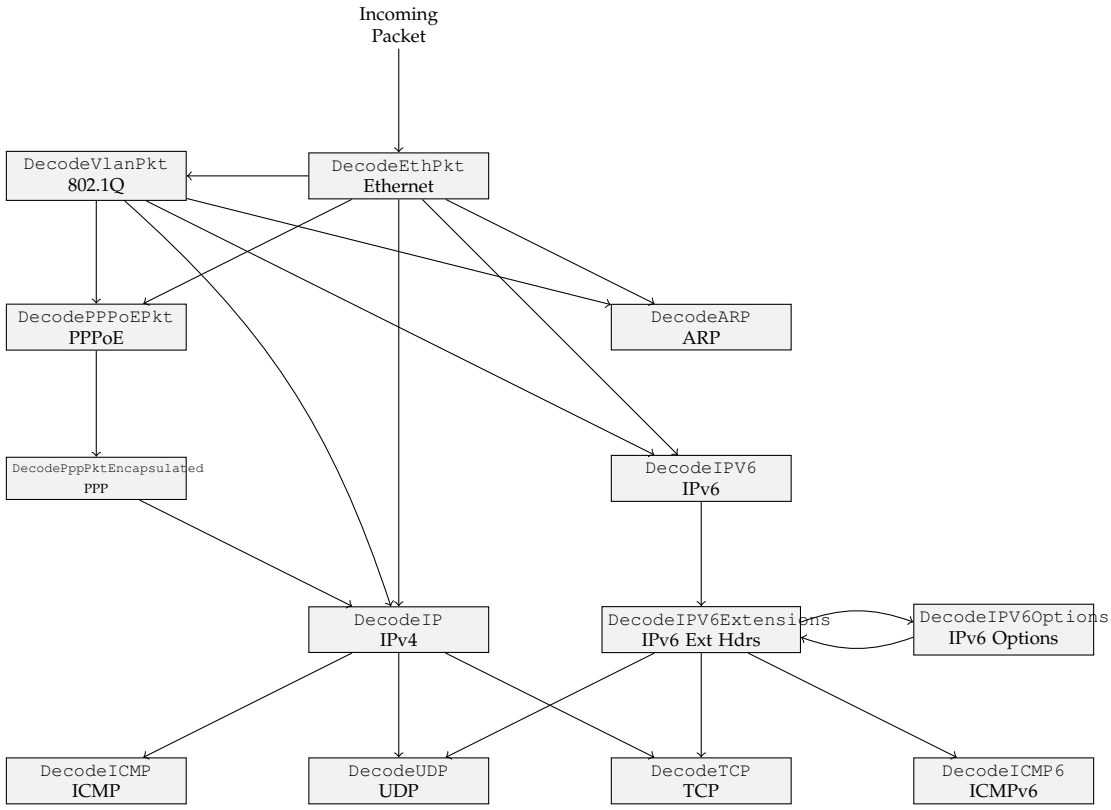


Figure 19: Call Graph of Snort's Decoding Stage for Ethernet (based on Beale, Baker, Esler et al. 2007, p. 184.

ports) and additional optimization for content pattern matching.

Every rule requires a header with an action (`log`, `alert`, `drop`, ...), protocol (`ip`, `icmp`, `tcp`, `udp`), source address with port, direction, and destination address with port. Source and Destination may contain single addresses, lists of addresses, or the "any" keyword; to simplify configuration one uses variables (like `$SMTP_SERVERS` in figure 21b) but these are simply substituted at parsing and have no relevance to the underlying evaluation.

This header is followed by a number of options, enclosed in parenthesis. Several options are not used for evaluation, but contain meta-data about the rule; the most important ones are `sid` and `rev` to unambiguously identify a single rule (with revision number) and `msg` to provide a useful log message. For evaluation the most important option is `content` because it is used for many rules (nearly all application-specific rules search for content in TCP streams) and enables further optimization based on length and offset (implemented as a so called fast pattern matcher).

To facilitate application-specific rules the set of options is not fixed but extensible by preprocessors (described in section 15.2).

Most rules are designed to be stateless, which is an advantage because rule management does not have to consider dependencies among different rules; but also has the disadvantage because it prevents detection of patterns that involve more than one packet. The *stream5* preprocessor implements session tracking for TCP connections and series of UDP packets between the same endpoints. This session tracking also includes a simple mechanism to add state to sessions with the `flowbit` option. Using this option a rule can set (or unset, or toggle) a named bit for a session (e. g. when the application performs a user lo-

```
typedef struct _Packet
{
    const DAQ_PktHdr_t *pkth;     // packet meta data
    const uint8_t *pkt;           // raw packet data

    EtherARP *ah;
    const EtherHdr *eh;          /* standard TCP/IP/Ethernet/ARP headers */
    const VlanTagHdr *vh;

    const IPHdr *iph, *orig_iph;/* and orig. headers for ICMP_*_UNREACH */
    const IPHdr *inner_iph;      /* if IP-in-IP, this will be the inner  */
    const IPHdr *outer_iph;      /* if IP-in-IP, this will be the outer  */

    IP4Hdr *ip4h, *orig_ip4h;    /* SUP_IP6 members */
    IP6Hdr *ip6h, *orig_ip6h;
    ICMP6Hdr *icmp6h, *orig_icmp6h;

    uint32_t preprocessor_bits; /* flags for preprocessors to check */
    uint32_t preproc_reassembly_pkt_bits;

    uint8_t frag_flag;           /* flag to indicate a fragmented packet */
    uint8_t mf;                  /* more fragments flag */
    uint8_t df;                  /* don't fragment flag */
    uint8_t rf;                  /* IP reserved bit */

    uint8_t uri_count;           /* number of URIs in this packet */
    uint8_t error_flags;         /* flags indicate checksum errors, etc. */
    uint8_t encapsulated;

    uint8_t ip_option_count;     /* number of options in this packet */
    uint8_t tcp_option_count;
    uint8_t ip6_extension_count;
    uint8_t ip6_frag_index;

    uint8_t ip_lastopt_bad;      /* flag to indicate that option decoding
                                    was halted due to a bad option */
    uint8_t tcp_lastopt_bad;     /* flag to indicate that option decoding
                                    was halted due to a bad option */

    uint8_t next_layer;          /* index into layers for next encap */

    // ...

    IPOptions ip_options[MAX_IP_OPTIONS];
    TCPOptions tcp_options[MAX_TCP_OPTIONS];
    IP6Extension ip6_extensions[MAX_IP6_EXTENSIONS];

    const IP6RawHdr* raw_ip6_header;
    ProtoLayer proto_layers[MAX_PROTO_LAYERS];
    LogFunction log_funcs[MAX_LOG_FUNC];
    uint16_t max_payload;

    /**policyId provided in configuration file. Used for correlating configuration
     * with event output
     */
    uint16_t configPolicyId;

} Packet;
```

Figure 20: Data structure for Snort's packet processing (shortened to show only 39 out of 114 fields); `decode.h:1465ff`

```
var EXTERNAL_NET any
var HOME_NET     [192.0.2.0/24,2001:db8:12:ab::/64]
var SMTP_SERVERS [192.0.2.123,2001:db8:12:ab::123]
```

(a) Snort configuration variables.

```
alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (
  msg:"SMTP sendmail 8.6.9 exploit";
  flow:to_server,established;
  content: "|0A|Croot|0A|Mprog";
  metadata:service smtp;
  reference:arachnids,142;reference:bugtraq,2311;reference:cve,1999-0204;
  classtype:attempted-user;
  sid:669; rev:9;
)
```

(b) All TCP traffic to port 25 of defined SMTP servers will be examined and an alert is raised if the given `content` is found.

```
 alert icmp $EXTERNAL_NET any -> $HOME_NET any (
  msg:"ICMP traceroute"; itype:8; ttl:1;
  reference:arachnids,118;
  classtype:attempted-recon;
  sid:385; rev:4;)
```

(c) All incoming ICMP traffic is checked for packets with ICMP type 8 (Echo request) and a *time to live* of 1; these packets cause an alarm, because they indicate a traceroute reconnaissance.

Figure 21: Snort rule examples.

gin) and can also test a named bit (e. g. a rule might detect a critical situation but only trigger an alert if there was, or was not, a prior user login). The dependency on sessions make the `flowbit` mechanism quite effective for tracking application layer protocols, but obviously unsuitable for monitoring the network layer.

Snort also includes an API to load complete rule evaluation routines from dynamic libraries. This serves three purposes: the first is simply to take existing rules and compile them for higher performance, the second is to implement checks that cannot be expressed in the normal rule notation, the third is to allow the distribution of rules in binary form without an easily human-readable version (see section 15.1).

14.5 Output

All alarms and log messages from preprocessors and matching rules are collected in an event queue (one for every packet). The length of the event queue and its ordering is configurable, and superfluous events are ignored.

An important function of the event queue is the detection, rate, and event filtering. The main configuration commands are the `detection_filter` option, which is used as part of a signature, and the `event_filter` command, which is used standalone and applies to generic events (i. e. to both rules and preprocessor alerts). These statements implement either a rate limit (log only a given number of events per interval and ignore additional events), a threshold (log only if an event occurs more often than a given number of times per interval), or a combination of both (cf. figure 22). A related command for IDPS configurations is the `rate_filter`, which changes the kind of rule action when a configured rate is exceeded. This enables more complex rules with multiple thresholds, for example a denial-of-service prevention that logs when some rate is reached and drops the packets when a second, higher rate is reached.

After the detection stage this event queue is processed. For every event its associated action is triggered, and if events are associated with rate filters, their thresholds are also checked here. In IPS or inline mode the `pass`, `drop`, or `reject` actions are triggered by setting corresponding flags. After completion of this processing stage the DAQ module will then read these flags and perform the requested action.

The usual actions `log` and `alert` can be more complex, so traditionally they have been encapsulated in another (static) Plugin API to support various output channels ranging from syslog messages to SQL databases; but because of Snort's single-threaded design, problems in output modules (e. g. a slow or lost database connection) can impair the whole IDS. For this reason the current best practice is to decouple IDS and output processing. One way to configure this is to write all Snort output (logs and alerts) into local binary log files using the *unified2* output module. The *unified2* data format is a type-length-value (TLV) encoding (cf. figure 24) and provides types for different events (like IPv4 event, IPv6 event) and packets (cf. figure 23 for important data structures).

Other programs can read the *unified2* files and post-process the log data, either by analyzing it directly or by converting it into more suitable output formats. Common conversions include forwarding the events via syslog and writing events and payload into an SQL database. The most commonly used tool for post-processing is BARNYARD2[17], which uses a plugin architecture itself so it is extensible with new and customized output formats.

17 http://www.securixlive.com/barnyard2/ v. 2013-12-22

```
event_filter \
    gen_id 1, sig_id 1851, \
    type limit, track by_src, \
    count 1, seconds 60
```

(a) Limit to logging 1 event per 60 seconds.

```
event_filter \
    gen_id 1, sig_id 1852,
    type threshold, track by_src, \
    count 3, seconds 60
```

(b) Limit to logging every 3rd event per 60 second interval.

```
event_filter \
    gen_id 1, sig_id 1853, \
    type both, track by_src,  \
    count 30, seconds 60
```

(c) Limit to logging just 1 event per 60 seconds, but only if we exceed 30 events in 60 seconds

Figure 22: Examples of different `event_filter` types http://www.snort.org/ file README.filters.

```
typedef struct _Serial_Unified2_Header
{
    uint32_t   type;
    uint32_t   length;
} Serial_Unified2_Header;

//UNIFIED2_PACKET = type 2
typedef struct _Serial_Unified2Packet
{
    uint32_t sensor_id;
    uint32_t event_id;
    uint32_t event_second;
    uint32_t packet_second;
    uint32_t packet_microsecond;
    uint32_t linktype;
    uint32_t packet_length;
    uint8_t packet_data[4];
} Serial_Unified2Packet;

//UNIFIED2_IDS_EVENT_IPV6_VLAN = type 105
typedef struct _Unified2IDSEventIPv6
{
    uint32_t sensor_id;
    uint32_t event_id;
    uint32_t event_second;
    uint32_t event_microsecond;
    uint32_t signature_id;
    uint32_t generator_id;
    uint32_t signature_revision;
    uint32_t classification_id;
    uint32_t priority_id;
    struct in6_addr ip_source;
    struct in6_addr ip_destination;
    uint16_t sport_itype;
    uint16_t dport_icode;
    uint8_t  protocol;
    uint8_t  impact_flag;
    uint8_t  impact;
    uint8_t  blocked;
    uint32_t mpls_label;
    uint16_t vlanId;
    uint16_t pad2; /*could be IPS Policy local id*/
} Unified2IDSEventIPv6;
```

Figure 23: Important *unified2* data structures `Unified2_common.h`.
Different event types are very similar; for example the only difference to the IPv4 event format is that the latter uses a `uint32_t` for source and destination addresses.



Figure 24: TLV structure of *unified2* output files.

15 Snort Plugin APIs

Snort provides different APIs for customized components, both static (i. e. requiring changes in Snort itself) and dynamic (i. e. extensible with a dynamically linked library). These components range from data acquisition in *libdaq* to message output plugins. For extended protocol support and detection it has a dynamic detection API and a dynamic preprocessor API.

15.1 Snort Dynamic Detection API

The dynamic detection API Beale, Baker, Esler et al. 2007, chap. 5 allows to replace rules with code from dynamically shared object files (`.so` on unix-like systems). When Snort initializes its rules it will read all shared object files in the configured directory (given with `dynamicdetection directory`) and use the rules defined therein. Snort offers two ways to write these rules: either in the same way as in the rule language or by implementing an evaluation function.

The first option is a more or less direct mapping of the rule language to C data structures. The rule is written as a `struct _Rule` with values for the rule header, essential meta-data, and an array of rule options. When these rules are loaded they are treated just like text rules and their options are inserted into the rule evaluation tree.

The alternative is to use an evaluation function, which has no resemblance to the textual rule language. Instead, a function is written that will receive a packet as its argument and returns whether the packet matches the rule or not. This obviously allows for greater flexibility. For example such a function could inspect data patterns that would be very complex to describe in the rule language, or it might combine rule options by disjunction instead of normal conjunction.

The biggest limit of the detection API is the binary return value: for every rule and every packet it can only indicate a match or no match.

15.2 Snort Dynamic Preprocessor API

The dynamic preprocessor API is more powerful and more appropriate for protocol level verification, so it will be described in greater detail. It uses a typical event handler design in which every plugin gets initialized and registers a number of callback routines or handler functions to be called for certain events.

Library Initialization

At the most basic level, every dynamic preprocessor is a shared object file with a small set of defined symbols. On Snort start-up all configured plugin library files (either single files given with `dynamicpreprocessor file` or all files in one directory with `dynamicpreprocessor directory`) are opened and their respective entry function is called. This entry function receives a list of all function pointers (in a `struct _DynamicPreprocessorData`) which it will need to interact with other Snort subsystems. Besides saving this pointer it also registers the plugin with its name, version number, and initialization routine.

Preprocessor Initialization

For those preprocessors that are activated in `snort.conf` (with a `preprocessor` directive), their previously registered initialization routine is called. A usual preprocessor initialization will process the given configuration parameters (used to set processing options or to pass knowledge about the environment), allocate memory, prepare internal data structures, and register further handlers for packet processing.

Additionally it will save the created configuration data structure as its “context”. Because Snort supports multiple configurations, a single Snort instance can monitor different VLANs or IP subnets with different settings; it can also reload its configuration file at runtime. For both of these functions it expects all preprocessors to use only one pointer to hold their configuration and state. This pointer is later passed at every invocation of the preprocess handler. Thus Reloading is implemented by replacing this user data; likewise the use of several pointers to different contexts enable multiple configurations to coexist in one Snort instance.

Rule Option Handlers

The first set of handlers are used to provide option keywords for detection rules (if the preprocessor provides any). For this the preprocessor registers a keyword to be used as a rule option, along with a set of handlers to process the rules. The parameters for this registration are first a keyword for the option, followed by a handler for rule option initialization. This one is called when the rule option is parsed and it has to convert the given textual parameter into an internal data structure. The third parameter is the handler for rule evaluation, called at runtime whenever the rule option is evaluated for a packet, this handler tells whether the rule option matches or not. The fourth parameter is a handler for memory cleanup, called when exiting or reloading Snort; usually the `free` function is used as only complex data structures require more sophisticated cleanup routines.

Actually every occurrence of an option and the subsequent call to the rule option initialization will create a new instance of this rule option and allocate a new data object When the detection engine builds the option tree for rule evaluation it uses a simple hash function to compare new options to existing ones; because this hash function includes the memory object address it cannot recognize redundant instances and will add all of them as new option tree nodes (OTNs); i. e. if one creates 1000 rules including the option

```
typedef int (*eval_func_t)(void *option_data, Packet *p);

typedef struct _detection_option_tree_node
{
    void *option_data;
    option_type_t option_type;
    eval_func_t evaluate;
    int num_children;
    struct _detection_option_tree_node **children;
    int relative_children;
    int result;
    struct
    {
        struct timeval ts;
        uint64_t packet_number;
        uint32_t pipeline_number;
        uint32_t rebuild_flag;
        char result;
        char is_relative;
        char flowbit_failed;
        char pad; /* Keep 4 byte alignment */
    } last_check;
} detection_option_tree_node_t;
```

Figure 25: Data structure for option tree nodes. For preprocessor rule options `option_data` points to the instance data and `evaluate` to the handler function.

`ssl_version:sslv2` then all instances of this option become separate option nodes which have to be evaluated at runtime. For rule options that only compare integer values this is often acceptable because both the data object (holding the integer) and the evaluation function (performing a comparison) need few resources. Nevertheless, this is a suboptimal situation which is avoided if the preprocessor also provides its own hash and comparison functions.

The hash function has to return a hash value based on the rule option and parameter. When two option hashes are equal the comparison function will be called and it decides whether the provided data objects represent the same rule option. When these callbacks are provided, the detection engine can recognize duplicate options as identical. It will then discard the new one as redundant (and free its memory) and reuse the existing option node. Figure 25 shows the OTN data structure and figure 26 shows the tree optimization for four rules and four instances of the four different rule options. It also shows how the detection engine uses leaf nodes to signify a completed path and thus a rule match. Their evaluation causes a lookup of which rule(s) have matched and queues the associated action (alert, log, etc.) in the packet's event queue.

The rule option registration may receive additional parameters. These may register a custom OTN handler, which may be used to write rule options that are not inserted into the detection option tree (currently only used by the *sdf* preprocessor for `sd_pattern` rules) Plugins may also register handlers to access and optimize the fast pattern matcher (currently only used by the *dce2* preprocessor for `dce_iface` rules).

Preprocessor Handler

The most important handler for many preprocessors is a callback for every packet. Besides the callback and an ID the registration uses a priority and a selector. The priority determines the preprocessor ordering (available priorities are: first, normalize, network, transport, tunnel, scanner, application, and last) and selectors allow the preprocessor to be called for certain protocols only (available selectors are IP, ARP, TCP, UDP, ICMP, Teredo, and All). The given function will be called for every packet matching the selector (including "pseudo-packets" after defragmentation and stream reassembly). There are no limits to a preprocessor's access to the packet data or the actions it can cause upon detecting an incident. Just like the built-in preprocessors it has full access to the current packet and can inspect the packet with access to both the original packet data in memory and the already decoded representation of its lower layers (up to TCP or UDP, in a `struct _SFSnortPacket`). It can keep state in arbitrary data structures of its own and also access some of Snort's other subsystems in order to log messages or add to a packet's event queue.

Other Handlers

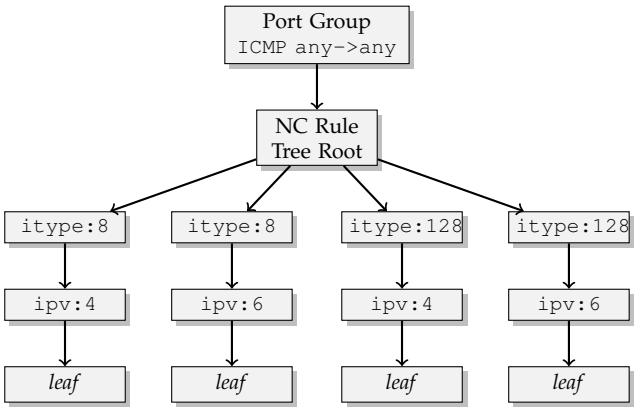
Additional handlers can be registered for configuration checking at startup, printing module statistics at shutdown, profiling module performance, and reloading the configuration.

To trigger alerts Snort provides its preprocessors with a function to add an alert to the packet's event queue using a generator ID (unique for one preprocessor), Snort rule ID (unique for one rule or preprocessor alert situation), and a log message. Besides these alerts preprocessors can also generate different log messages, ranging from debug information to normal notices up to error messages.

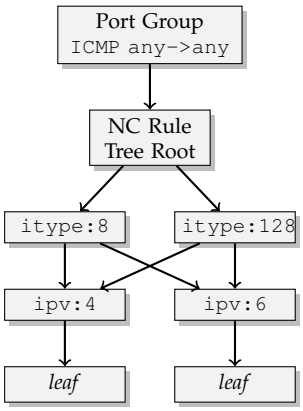
For more complex preprocessors Snort also provides functions to access several of its subsystems (policy, stream, search, and obfuscation functions), to influence the packet's processing (e. g. evaluate rules, disable further detection, or create new packets), and to manipulate the packet's event queue. Finally some

```
alert icmp any any -> any any (itype:8;   ipv: 4;
    msg:"ICMPv4 PING in v4 pkt"; sid:100000; rev:1;)
alert icmp any any -> any any (itype:8;   ipv: 6;
    msg:"ICMPv4 PING in v6 pkt"; sid:100001; rev:1;)
alert icmp any any -> any any (itype:128; ipv: 4;
    msg:"ICMPv6 PING in v4 pkt"; sid:100002; rev:1;)
alert icmp any any -> any any (itype:128; ipv: 6;
    msg:"ICMPv6 PING in v6 pkt"; sid:100003; rev:1;)
```

(a) Example signatures using the (built-in) `itype` and the (plugin-provided) `ipv` rule options.



(b) Without optimization every rule would be stored as a list of its options (similar to the three dimensional linked list used in Snort versions before 2.0).



(c) The actually generated optimized evaluation tree.

Figure 26: Example of detection rules and the resulting rule option evaluation tree. The rules will yield a single port group for `ICMP any->any` without content rules.

common utility functions are provided as well (most of them to parse configuration options like splitting tokens and converting strings to integers).

16 IPv6 Plugin Requirements

An IDS plugin for IPv6 should be able to detect the known attacks (described in section 1) and have at least as much functionality as the existing "small" tools (mentioned in sections 8.5 and 13.4). Additionally it should fit into the IDS's toolchains for data processing (for generated logs) and administration (for configuration and rule management).

The detection of known attacks can be divided into sets of stateless and stateful checks. Stateless checks should be easy to implement and be sufficient to verify used protocol options, like the use of routing headers. To accommodate IDS deployments on hardware with limited memory, it should be possible to enable or disable all checks with higher resource consumption.

It still needs to be decided whether to implement these checks with hard-coded values inside the plugin, with configurable values inside the plugin, or as a rule outside of the plugin; in the last case the plugin has to implement the rule option to access the IPv6-specific fields, but the logic and the matching/non-matching values are part of the rule set, thus accessible to signature management tools. The best choice depends on how often the checks are expected to change. New ICMPv6 types and routing headers take a long time to specify and implement in routers, so it is reasonable to hard-code them and release a new version of the plugin. On the other hand all site-specific settings have to be user configurable.

In order to facilitate future rule development, a plugin should make all protocol fields accessible for Snort rule options. This affects in particular the IPv6 header fields *Traffic Class* and *Flow Label* because these have no direct IPv4 equivalent; so they are not accessible with Snort's existing built-in rule options. Likewise the extension headers and option types inside these extension headers should be testable with rule options. One may be able to test the option's values as well. – Yet as the option values have no common format, this function may be limited to a subset of options with 8 or 16 bit values.

All neighbor discovery checks are stateful, because they will have to keep track of currently active hosts. Also stateful by nature are fragment reassembly and host/port scanning detection – but in common IDS architectures these functions are already implemented in existing preprocessors, thus out of scope for an IPv6 plugin.

The desired integration into the IDS is a non-functional requirement, but it is important because this integration makes a plugin more valuable than standalone tools like NDPMON. As a first step available services offered by the IDS should be used, because Snort and other IDSs already include tested code for decoding and fragmentation reassembly and there is no need to re-implement these functions. A second step is the configuration: it should follow the

configuration of similar plugins to make it easier to manage and prevent surprises. Likewise the generated alerts and messages should be useful for later analysis, for example alerts should be descriptive, and support individual activation/deactivation. Finally a good plugin should enable future development. By providing rule options it allows developers and users to write more complex or customized signatures using the plugin's detection capability.

17 IPv6 Plugin Functionality

This project implements three layers of functionality: the preprocessor tracks neighbor discovery messages and generates preprocessor alerts for significant events; independently a set of rule options provide access to IPv6 values for Snort detection signatures; finally some detection signatures (using the new rule options) are prepared to detect IPv6-specific anomalies.

17.1 Preprocessor Alerts

The preprocessor maintains its own network view by tracking all neighbor discovery messages. On every change in this network view it triggers a snort event; this happens every time a new host or router appears on the net, a router changes its advertisement, and a duplicate address detection fails.

The effectiveness and accuracy of this tracking depends on the network topology and sensor placement. The preprocessor is designed to use a sensor on the switch's monitoring port, which will receive all neighbor discovery messages of the subnet. A sensor running on a gateway or a packet filter will receive fewer packets and only has access to partial status information.

Most of these events are intended to directly cause a Snort alert, for example whenever a new router appears or a router changes the advertised prefix. If Snort is configured and compiled with the `enable-decoder-preprocessor-rules`, then it is also possible to change the rule type (e. g. to change an `alert` into a `drop` rule). Some other events, like the DAD failure, are subject to Snort's rate filtering mechanism, for example to detect flooding attacks. Table 2 lists all preprocessor alerts.

The preprocessor accepts a small number of optional parameters for site-specific customization. These allow for static configuration of router and host MAC addresses and the local subnet prefix. For an example see figure 27.

Size and Duration of NDP Tracking State

The number of tracked nodes is limited to control memory usage. The configuration parameters `max_routers` *n*, `max_hosts` *n*, and `max_unconfirmed` *n* specify for how many routers, hosts, and tentative IP addresses (i. e. started duplicate address detections) the preprocessor keeps state information. The defaults are `max_routers 32, max_hosts 8192, max_unconfirmed 32768`. Every tracked IP address requires 60 bytes of memory, so by default the memory usage for the preprocessor's net view is capped to about 2.4 Mb.

The parameters `expire_run` *m* and `keep_state` *n* determine how quickly the preprocessor forgets about inactive nodes; the defaults are `expire_run 20 keep_state 180`. Every *m* minutes the state information is cleaned up and nodes which have been inactive for more than *n* minutes are removed from the preprocessor's network model. – In a strict sense the preprocessor determines activity only by seeing ICMPv6 messages from the node. Still this is sufficient, because IPv6's neighbor cache and the regular neighbor unreachability detection require link-local ICMPv6 messages for every IPv6 communication between hosts.

Generally speaking, these parameters have little impact on the preprocessor's function and there should be no reason to change the defaults. Only under very tight memory constraints would it be sensible to decrease the limit of tracked nodes (to reduce memory usage in case of a denial-of-service attack), or in unusually large subnets it could be useful to give a higher `expire_run` parameter to reduce the performance impairment of state cleanups.

White Lists

Several parameters help to adapt the preprocessor to local network environments. With `net_prefix` $Prefix_1$ ... $Prefix_n$ one or more subnet prefixes are specified in Classless Inter-Domain Routing (CIDR) notation; if this option is used then the preprocessor will alert whenever it detects a router announcing a different prefix.

The parameter `router_mac` MAC_1 ... MAC_n tells the preprocessor to verify the MAC address of router advertisements and raise an alert if any other device acts as a router. Similarly the parameter `host_mac` MAC_1 ... MAC_n tells it to check all packets for unknown source MAC addresses.

It is generally advisable to use the `net_prefix` and `router_mac` options, because they enable useful IDS checks with very little effort. The `host_mac` option is only appropriate for small and static networks where the administrative cost of maintaining a list of MAC addresses is viable.

Disable Tracking

Finally, the configuration parameter `disable_tracking` completely disables the tracking of neighbor discovery messages and network state. In this mode the preprocessor will still inspect every ICMPv6 packet and perform all stateless checks. – But it will not save any MAC or IP addresses as its network state.

Table 2: IPv6 preprocessor alerts (using GID 248)

SID	Message
1	RA from new router
2	RA from non-router MAC address
3	RA prefix changed
4	RA flags changed
5	RA for non-local net prefix
6	RA with lifetime 0
7	new DAD started
8	new host in network
9	new host with non-allowed MAC address
10	DAD with collision
11	DAD with spoofed collision
12	mismatch in MAC and NDP source linkaddress option
13	ipv6: extension header has only padding options (evasion?)
14	ipv6: option lengths != ext length

```
preprocessor ipv6:              \
  router_mac 00:16:76:07:bc:92  \
  net_prefix 2001:db8:1::/64    \
  keep_state 120
```

Figure 27: Example preprocessor activation with configuration parameters.

17.2 Rule Options

As described previously (cf. section 13.1), the Snort detection engine tries to unify IPv4 and IPv6 processing and applies existing IPv4 options to IPv6 packets as well. While this is useful in many cases, it does not allow the creation of IPv6-specific signatures – so this plugin implements an additional set of rule options.

Table 3 lists the available rule options to match the different IPv6 header fields. The built-in options are already provided by Snort and apply to IPv4 and IPv6 packets alike; the other options (marked in the "new" column) are implemented by the IPv6 preprocessor. There are no explicit tests for addresses, because IP addresses and network prefixes are already matched in the rule's header.

Several options support the use of modifiers to negate a match (like `ip_proto: !17;`), to use comparison operators (like `ip6_extnum: >2;` and `ttl: <=10;`), to specify ranges (implemented with different symbols in `dsize: 640<>1280;` and `ttl: 200-240;`), or to apply the boolean functions XOR, AND and NAND (like `ip6_tclass: &0x00ff;`).

Except for `ip6_optval`, all `ip6_*` options have a uniform syntax: They take an optional comparison operator ($\star$) and a numeric argument (n) in either decimal or hexadecimal notation. The following operators are implemented (but not all are applicable to every option):

= equality (default)
< less than
^ binary exclusive-or
! negation
> greater than
& binary and
| binary nand

These binary operators are sufficient to cover all use cases because all options of a rule form a logical conjunction. To prevent ambiguity the operators $\geq$ and $\leq$ are not implemented, nor are range operators (as in `ttl: 200-240;`); in simple cases ranges can be expressed with two options (e. g. `ip6_extnum: >0; ip6_extnum: <4;`). However, this kind of chaining does not work for elements that may occur multiple times in a single packet. For example the signature with `icmp6_nd_option: >10; icmp6_nd_option: <17;` does not necessarily select SEND-specific neighbor discovery options, but would also match a router advertisement containing a type 1, *Source Link-layer Address* and a type 25, *Recursive DNS Server Option*.

17.3 Rules

In most cases the use of Snort signatures is preferable to preprocessor alerts, because signatures are more flexible and customizable by users.

Nearly all of the plugin's stateless detection functions are implemented as Snort signatures using the IPv6-specific rule options described in section 17.2. This enables users to easily enable/disable the signatures, modify them as they deem appropriate for their network environment (e. g. by changing an `alert` into a `log` action, or adding a rate filter to frequent events), and organize them using existing signature management tools.

The prepared rules fall into three categories: The first set matches unusual IPv6 packets, which indicate net-

```
alert icmp any any -> any any (ipv: 6; itype: 130<>138; ttl: <255; \
    msg:"ICMPv6/NDP msg routed";    sid:124800; rev:1;)
```

(a) Signature to detect NDP messages which have been routed from another subnet.

```
alert icmp any any -> any any (ipv: 6; itype: 137; ttl: 255;       \
    msg:"ICMPv6/NDP Redirect msg";  sid:124803; rev:1;)
```

(b) Signature to detect redirect messages.

```
alert icmp any any -> any any (ipv: 6; itype: 134;                 \
    detection_filter: track by_dst, count 5, seconds 1;            \
    msg:"ICMPv6/RA flooding";     sid:124850; rev:1;)

event_filter gen_id 1, sig_id 124850, type limit, track by_dst, count 1, seconds 60
```

(c) Signature to detect router announcement flooding, including both a threshold (five messages per second) and a rate limit (one alert per minute).

Figure 28: Example rules from `ipv6.rules`.

work problems. These might be invalid packets, for example neighbor discovery messages with a *Hop Limit* $\neq 255$ (as neighbor discovery messages should be link-local and not be forwarded by the router, e. g. in figure 28a), or legit but rarely used messages, such as *Router Renumbering* messages.

The second set has to be enabled/disabled depending on the local network configuration. It contains alerts for SEND, DHCPv6, IPsec and *Redirect* messages (cf. figure 28b), which are either very common or should never appear at all on a given subnet – depending on whether the network and its servers are configured to use these protocols or multiple routers.

The third set is intended to alert on flooding attacks. Its signatures select normal neighbor discovery messages but use `detection_filter` options to add a threshold of several messages per second and also a `event_filter` to limit the number of events to one alert per minute (cf. figure 28c).

18 Implementation and Snort Integration

The next sections will show how the described functionality is implemented. The plugin's interface follows the specification of the Snort dynamic preprocessor API as described in section 15.2.

18.1 Plugin Initialization

As soon as the shared object is loaded, a minimal library initialization handler is called and registers the preprocessor plugin with name *ipv6* and initialization handler `IPv6_Init`.

This preprocessor initialization handler (schematic shown in figure 29) is called if the the user activates the plugin by adding the line `preprocessor ipv6` to their `snort.conf`. It receives the given configuration parameters (if any), parses them (using the separate function `IPv6_Parse`), allocates memory for the plugin's data structures, and finally registers all further callback handlers. This includes the most important handler functions `IPv6_Process` for the preprocessor and `IPv6_Rule_Init` / `IPv6_Rule_Eval` for the rule options.

18.1.1 Preprocessor Handler Registration

Most of this initialization concerns the preprocessor functionality. First the memory for the preprocessor's data structures is allocated and all configuration parameters are parsed. Then the calls to `sfPolicyUserPolicySet` and `sfPolicyUserDataSetCurrent` associate the newly created configuration with the current Snort context (only relevant if Snort uses multiple configurations/contexts). After this internal preparation, the different callback handlers are registered.

The `_dpd.addPreproc` service adds the preprocessor routine `IPv6_Process` as a preprocessor (figure 29). Snort holds all preprocessor callbacks as a linked list, so the second argument indicates where the preprocessor is inserted.

The different `PRIORITY_*` symbols provide a simple way to influence, but not to control, the order of preprocessors. Preprocessors with the same priority are simply added behind one another (in order of their configuration/registration) and static (i. e. Snort built-in) preprocessors take precedence over dynamic plugins. For example if the *normalize_ip6* preprocessor is used, it would still run before any dynamic plugin because it also uses `PRIORITY_FIRST` and is added first. – Most importantly the chosen priority decides whether the plugin receives fragment packets (as it does with `PRIORITY_FIRST`) or not (as would be the case with the alternatives `PRIORITY_NORMALIZE` or `PRIORITY_NETWORK`). This is because the *frag3* preprocessor uses the second priority `PRIORITY_NORMALIZE` and disables all following preprocessors for non-UDP fragments as an optimization.

The third argument (`PP_IPv6`) registers a flag bit to identify the preprocessor. This is used to selectively

```
void IPv6_Init(char *args)
{
    struct IPv6_State  *context;
    struct IPv6_Config *config;
    context = (struct IPv6_State *)  calloc(1, sizeof (struct IPv6_State));
    config  = (struct IPv6_Config *) calloc(1, sizeof (struct IPv6_Config));
    // allocate all other structures ...

    IPv6_Parse(args, config);
    // ...

    sfPolicyUserPolicySet(ipv6_config, _dpd.getParserPolicy());
    sfPolicyUserDataSetCurrent(ipv6_config, context);

    _dpd.addPreproc(IPv6_Process, PRIORITY_FIRST, PP_IPv6, PROTO_BIT__IP);

    _dpd.registerPreprocStats("ipv6", IPv6_PrintStats);
#ifdef PERF_PROFILING
    _dpd.addPreprocProfileFunc("ipv6", (void *)&ipv6PerfStats, 0, _dpd.totalPerfStats);
#endif

    _dpd.preprocOptRegister("ipv", IPv6_Rule_Init, IPv6_Rule_Eval,
            free, IPv6_Rule_Hash, IPv6_Rule_KeyCompare, NULL, NULL);
    _dpd.preprocOptRegister("ip6_tclass", IPv6_Rule_Init, IPv6_Rule_Eval,
            free, IPv6_Rule_Hash, IPv6_Rule_KeyCompare, NULL, NULL);
    // register all other rule options with same handlers ...
}
```

Figure 29: Plugin initialization and registration in `spp_ipv6.c`

enable or disable preprocessors for a given packet. Snort provides a field of 32 bits and every established preprocessor is associated with one of the 23 currently assigned bits. When processing a packet, the preprocessors' flags are matched against this bitfield (`u_int32_t preprocessor_bit_mask;` in `struct _SFSnortPacket`) to decide whether the preprocessor is called or skipped.

The fourth argument is a selector, it indicates that a preprocessor only wants to be called for certain types of packets. – The IPv6 preprocessor uses the value `PROTO_BIT__IP` to get called for all IP packets (which is a tiny bit better than `PROTO_BIT__ALL` because the latter would also give access to ARP packets).

Besides the main functionality, a few minor handlers are registered for statistics and profiling. Among them the `IPv6_PrintStats` function is registered with `registerPreprocStats` to be called when Snort exits. It will then print some (hopefully) informative numbers about processed packets (cf. figure 30).

The last registration provides the data structure `ipv6PerfStats` for preprocessor performance profiling. This feature has to be enabled at compile time, hence is conditionally compiled depending on the definition of `PERF_PROFILING`. If it is enabled and also activated (using the configuration statement `config profile_preprocs`) then Snort will collect internal profiling data for preprocessor calls.

This registration affects only the profiling of preprocessor calls. Snort's detection engine also collects profiling data of all rule option evaluations, but aggregates all preprocessor rule options into one entry `preproc_rule_options`. There is no way to influence this data collection or to get more detailed information.

Snort prints a summary statistics of the collected profiling data when the program terminates, an example is shown in figure 34 (on page 446).

18.1.2 Rule Option Handler Registration

The rule options do not access any of the preprocessor's configuration or state data. So they are quite independent from the preprocessor part, except they are registered in the preprocessor initialization handler.

The service `_dpd.preprocOptRegister` is used to add rule options, receiving the so created option keyword as a first argument. The IPv6 plugin registers its different rule options (as listed in table 3), but uses the same handler functions for all of them (cf. figure 29): the first one (`IPv6_Rule_Init`) initializes one instance of a rule option, the second one (`IPv6_Rule_Eval`) applies a rule option instance to a packet, and the third one removes the instance (`free`). The next two handlers (the fifth and sixth, `IPv6_Rule_Hash` and `IPv6_Rule_KeyCompare`) are also provided for hashing and comparison (as explained in the API description, section 15.2). The callbacks for the custom OTN handler and fast pattern matcher are not used.

18.2 Rule Option Initialization

All rule options are registered with the same handlers: every rule option's "entry point" is the `IPv6_Rule_Init`, which is called when the configuration parser encounters an `ip6_*` option as part of a rule. The function parses the option's parameter and allocates a data object (of type `struct IPv6_RuleOpt_Data`, cf. figure 31) to store all parsed values of this option instance in memory. For most options the instance simply encodes the used option, the used comparison operator, and the value to compare against.

```
===============================================================================
IPv6 statistics:
   4979660 seen Packets
         0 invalid Packets
         0 Fragments
    143197 IPv6
     17276 ICMPv6
     17438 UDP
    107457 TCP
         0 SCTP
         0 Mobile IPv6
         0 Encapsulated
      1026 Other Upper Layer

         5 router solicitation
       176 router announcement
      4981 neighbour solicitation
      3252 neighbour announcement
       270 Mcast query
      5556 Mcast report
       197 dst unreachable
      2839 Other
===============================================================================
Snort exiting
```

Figure 30: IPv6 preprocessor statistics output.

```
struct IPv6_RuleOpt_Data {
    enum IPv6_RuleOpt_Type type:4;
    u_int8_t op:4;
    union {
        u_int32_t number;
        struct {          // for ip6_optval
            u_int8_t  ext_type;
            u_int8_t  opt_type;
            u_int16_t opt_value;
        } exthdr;
    } opt;
};
```

Figure 31: Rule option data structure in `spp_ipv6.h`

After `IPv6_Rule_Init` returns, the Snort parser will call `IPv6_Rule_Hash` and on collision also `IPv6_Rule_KeyCompare`. This is to check whether the created object instance is equal to any previously created instance (instances are equal if they represent the same option, with the same modifier, and the same value, cf. figure 32). – In case of equality it is redundant and removed; otherwise the instance, including the data object and a pointer to the registered evaluation function, is inserted into the rule evaluation tree (cf. section 15.2).

18.3 Rule Option Evaluation

At runtime all rule option evaluation is handled by the registered evaluation function `IPv6_Rule_Eval`. The detection engine will call it and pass the current packet and the instance's data object as arguments. So the evaluation will read the instance data and perform the appropriate comparison.

The following action is determined by the instance's rule type and for most rules this means the appropriate field in the packet is compared to the value using the operator of the rule option instance (`number` and `op`). A little more overhead is induced by the extension header and neighbor discovery option tests: these have to iterate through all present extension headers (or neighbor discovery options).

18.4 Preprocessor Alerts and Neighbor Discovery Tracking

At runtime the registered preprocessor function `IPv6_Process` receives all ICMPv6 packets for inspection.

The main functionality, which cannot be implemented with rule options, is tracking neighbor discovery protocol messages. This enables the plugin to keep its own model or network view of all active nodes on-link. Using this model it can issue alerts whenever the advertised routing configuration changes, a node enters the local network, or a duplicate address detection fails.

The plugin remembers every node on-link with its MAC address, IP address, and the last time it was seen on the network. It was decided to model a node as the combination of a MAC and IP address to allow for multiple nodes using the same MAC address (as is the case with virtualization), i. e. there are multiple entries for normal hosts with both link-local and global addresses. Nodes are categorized into three groups: *unconfirmed* (for new addresses in DAD), *routers*, and "normal" *hosts*; each one with its own data structure for fast lookup, currently implemented as red-black trees.

The distinction between unconfirmed/tentative and confirmed IP addresses enables the detection of denial-of-service attacks against the neighborhood

```
static int IPv6_Rule_KeyCompare(void *l, void *r)
{
    struct IPv6_RuleOpt_Data *left = (struct IPv6_RuleOpt_Data *)l;
    struct IPv6_RuleOpt_Data *right = (struct IPv6_RuleOpt_Data *)r;

    if (left && right
        && left->type       == right->type
        && left->op         == right->op
        && left->opt.number == right->opt.number) {
        return PREPROC_OPT_EQUAL;
    }
    return PREPROC_OPT_NOT_EQUAL;
}
```

Figure 32: Rule option instance comparison `spp_ipv6_ruleopt.c`

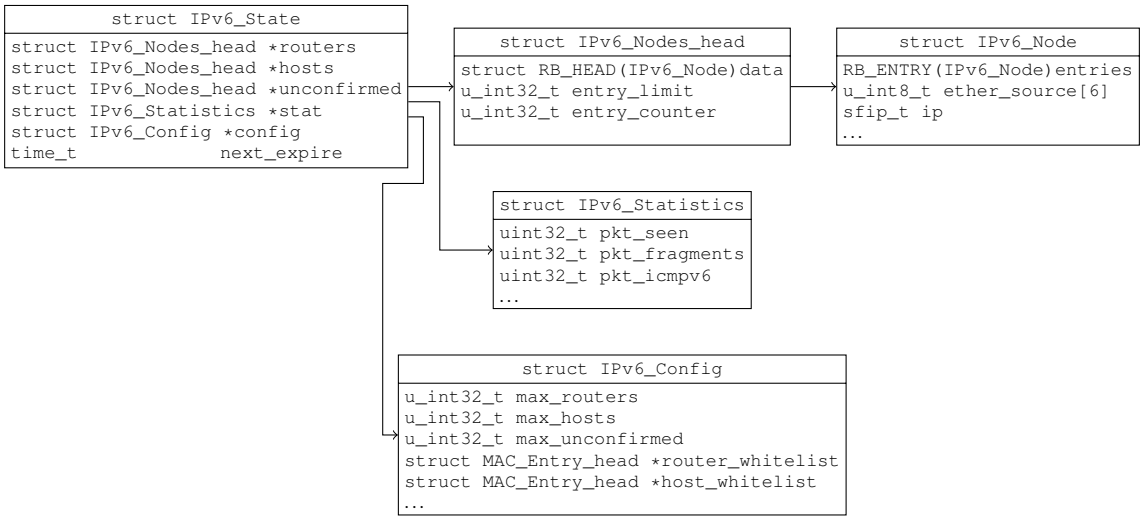


Figure 33: Data structures to keep the preprocessor state (configuration, statistics, and network model)

cache and against duplicate address detection. The use of separate data structures also facilitates a faster lookup for common ICMPv6 message processing (e. g. the number of *routers* will be very small and checking a router advertisement will not have to search through the larger storage tree for normal *hosts*).

To be confirmed, i. e. to be moved from the *unconfirmed* to the *host* state, a host has to send an ICMPv6 packet (appear as source address), then receive one or more ICMPv6 packets (appear as a destination), and later send another ICMPv6 packet of its own (appear as source address). This simple message exchange occurs whenever a host is active, because it has to send and receive the ICMPv6 messages for duplicate address detection, address resolution, and neighbor unreachability detection. On the other hand the source addresses of simple flooding or spoofing attacks will remain in the *unconfirmed* state because the fake addresses are just announced once.

It has to be emphasized that this model is very simple and is designed to prevent very simple (flooding) attacks. Determined attackers can circumvent the protection by keeping their own state; then they can send multiple messages with the same source and destination IP addresses to better simulate the behaviour of a real node. As with most countermeasures, the main goal here is not to prevent all flooding attacks (which is not possible), but to change the cost-effectiveness ratio of these attacks. Even simple measures drastically reduce this ratio, forcing an attacker to either use less effective attacks or to expend more resources for the same effect.

To protect the plugin from memory exhaustion due to denial-of-service attacks, all dynamic data structures use an element counter. The `max_routers`, `max_hosts`, and `max_unconfirmed` configuration options specify how many nodes are tracked at any time. When one of the trees has reached this capacity an alarm is generated and all subsequent nodes are ignored (because they cannot be added). An expiry function is called periodically (intentionally not on demand, as it would only add system load in a denial-of-service situation) and purges node entries that have been inactive for several hours.

Other parts of the `IPv6_state` object are the configuration options and some counters to print the statistics at program exit (cf. figure 33).

18.5 Performance and Resiliency

Preliminary concerns about the computational costs of such a plugin turned out to be unfounded. Because Snort's decoder processes every packet and fills the `struct _SFPacket` structure, the plugin itself has to perform very little computation and only has to decode neighbor discovery options on its own.

All rule options build upon the readily decoded

```
Run time for packet processing was 2.1795 seconds
Snort processed 100000 packets.
Snort ran for 0 days 0 hours 0 minutes 2 seconds
   Pkts/sec:        50000
Preprocessor Profile Statistics (all)
=====================================================
 Num            Preprocessor Layer     Checks      Exits          Microsecs  Avg/Check Pct of Caller Pct of Total
 ===            ============ =====     ======      =====          =========  ========= ============= ============
  1                   detect     0     101870     101870            1443000      14.17         81.39        81.39
   1                    mpse     1      71224      71224            1214449      17.05         84.16        68.50
   2              rule eval     1      31114      31114             191204       6.15         13.25        10.78
    1         rule tree eval     2     132998     132998             186698       1.40         97.64        10.53
     1                  pcre     3       1301       1301               6181       4.75          3.31         0.35
     2             byte_test     3         29         29                 24       0.84          0.01         0.00
     3            uricontent     3       1819       1819               1292       0.71          0.69         0.07
     [...]
     9                  flow     3      16812      16812                684       0.04          0.37         0.04
    10  preproc_rule_options     3    1157036    1157036              40146       0.03         21.50         2.26
    11              fragbits     3     101870     101870               2946       0.03          1.58         0.17
     [...]
    15                 itype     3     363957     363957               2532       0.01          1.36         0.14
    2               rtn eval     2      17294      17294               1486       0.09          0.78         0.08
  2                      s5     0      82563      82563             143429       1.74          8.09         8.09
   1                   s5tcp     1      82563      82563             123399       1.49         86.03         6.96
    [...]
  3                   decode     0     100092     100092              60006       0.60          3.38         3.38
  4              httpinspect     0      53619      53619              16085       0.30          0.91         0.91
  5               DceRpcMain     0      42103      42103              12510       0.30          0.71         0.71
   1           DceRpcSession     1      42103      42103               9586       0.23         76.63         0.54
  6                      ssl     0       5983       5983               1494       0.25          0.08         0.08
  7                   eventq     0     201856     201856              32926       0.16          1.86         1.86
  8                     ipv6     0     101856     101856              13498       0.13          0.76         0.76
  9                      ssh     0      48226      48226               4422       0.09          0.25         0.25
 10                     smtp     0      53467      53467               4886       0.09          0.28         0.28
 11                      dns     0       8283       8283                203       0.02          0.01         0.01
 total                 total     0     100000     100000            1773047      17.73          0.00         0.00
```

Figure 34: Snort preprocessor performance profiling (on IPv6-only input). The plugin's performance is relatively good and shown by the lines `preproc_rule_options` for all rule options and `ipv6` for the preprocessing.

packet representation; – except for the `ip6_rh` and the `ip6_nd_option` options, which require additional decoding of routing headers and neighbor discovery options. These might be more expensive because they have to examine a potentially large number of extension headers and options; thus their runtime is not constant but bound by $O(n)$ with n being the number of extension headers and options therein (or in case of neighbor discovery options n being the number of included options). In normal operation this does not have any significant impact as the number of defined extensions and options is small and very few packets contain any extension headers or options at all. But in theory this makes the plugin vulnerable to similar denial-of-service attacks as routers (cf. section 5.2).

The preprocessor's NDP tracking is more expensive, because it requires several memory lookups to check the node entry, possibly memory allocation to add new node entries, and periodic purging to free stale node entries. This may be a concern for very big subnets or when using Snort in inline mode; but in practice the cost is relatively small and well below that of the decoder or the fragmentation and streaming preprocessors (cf. figure 34).

For some neighbor discovery messages their content is verified, for example to detect whether router advertisement parameters have changed. It was decided to limit these checks to simple comparisons, because they should not open new denial-of-service attack vectors. Specifically the plugin does not verify SEND signatures.

So as a result the plugin's resource usage is similar to that of other Snort plugins. As such it does not require special consideration and it should be possible to add the plugin to every existing Snort installation, assuming deployed hardware is adequate for the network bandwidth to monitor.

19 Conclusion

The IPv6 protocol has several weaknesses in its neighbor discovery and autoconfiguration services. Most of these problems arise from the unsolved early authentication problem and the implicit assumption that all link-local nodes are trustworthy. Thus, an attacker with physical network access and control over a connected node is usually able to assume a man-in-the-middle position and also to perform various denial-of-service attacks against particular hosts or the router.

In exceptional cases, a trusted network can be established with an expensive combination of link-layer access control and cryptographic authentication of all devices. However, in most cases, the only feasible precaution is a separation into multiple subnets to confine every attack, and the only practical defense is a fast detection of the attack and the responsible device.

The evaluation shows that no current open source IDS product has sufficiently advanced IPv6 support to detect the documented attacks. Even though previous research projects wrote special purpose tools to monitor IPv6 neighbor discovery, these tools are rarely deployed because they require additional maintenance and do not integrate into the existing infrastructure.

The new IPv6 plugin was developed to extend the Snort IDS with integrated IPv6-specific detection routines. It adds an neighbor discovery tracking mechanism to alert when new hosts and routers appear on-link. It also provides additional rule options that expose IPv6 specific header fields to the Snort detection module. The rule options facilitate the writing of new detection signatures using the flexibility of Snort's rule language, for example to detect attacks from the THC toolkit. This integration into the Snort infrastructure facilitates an easy deployment and integration into existing IDS setups.

Table 3: Snort rule options to test IPv6 packets

Field	Option Format	Modifiers (⋆)	new[18]	Note
Version	`ipv: ⋆n;`	`= ! < > ^ & |`	✓	
Traffic Class	`tos: ⋆n;`	`!`		
Traffic Class	`ip6_tclass: ⋆n;`	`= ! < > ^ & |`	✓	
Flow Label	`ip6_flow: ⋆n;`	`= ! < > ^ & |`	✓	
Payload Length	`dsize: ⋆n;` `dsize: min<>max;`	`! < >`		
Next Header	`ip_proto: ⋆n;`	`! < >`		tests upper-layer protocol, i. e. with extension headers it tests the *Next Header* field of the last extension header.[19]
Next Header	`ip6_exthdr: ⋆n;`	`= !`	✓	tests presence of extension header `n` (ignoring the upper-layer protocol)
Hop Limit	`ttl: ⋆n;` `ttl: min-max;`	`< <= > >=`		
Source/Destination Address	`sameip;`			matches if *Source* and *Destination Address* are identical.
Routing Headers	`ip6_rh: ⋆n;`	`= !`	✓	tests the routing header type
Fragmentation Header[20]	`fragoffset: ⋆n;`	`! < >`		tests the fragment offset
Extension Headers	`ip6_extnum: ⋆n;`	`= ! < >`	✓	tests the number of extension headers
Extension Headers	`ip6_ext_ordered;` `ip6_ext_ordered: ⋆;`	`= !`	✓	tests whether all extension headers occur only once and in the recommended order
Hop-by-Hop/ Destination Options	`ip6_option: ⋆o;` `ip6_option: ⋆e.o;`	`= ! < > ^ & |`	✓	tests for the presence of an option type `o` in any extension header or in a specific extension header `e`
Hop-by-Hop/ Destination Options	`ip6_optval: e.o⋆n;`	`= ! < > ^ & |`	✓	tests for the value `x`[21] in option `e.o`[22]
Neighbor Discovery	`icmp6_nd;` `icmp6_nd: ⋆;`	`= !`	✓	matches ICMPv6 neighbor discovery messages[23]
Neighbor Discovery	`icmp6_nd_option: ⋆o;`	`= ! < > ^ & |`	✓	tests the neighbor discovery option type `o`

18 Options marked with ✓are implemented in the IPv6 preprocessor. Other options are built into Snort.

19 Functionality overlaps with the protocol selector in the rule's header, which selects `ip` (matches IPv4 and IPv6), `tcp`, `udp`, or `icmp` (matches ICMP and ICMPv6) packets.

20 Snort also includes the options `fragbits` and `id` to access IPv4 fragmentation bits and the *Identification* field. These are not included in this table because they do not work for IPv6 packets. – A patch to support the `fragbits` option was written and submitted to SourceFire.

21 Option have variable lenghts. The current implementation only uses 16 bit values for `x` and compares them to the first two octets of the option. – This is not sufficient for all possible values (e. g. for the Jumbogram option with a 32 bit value.)

22 The = operator is a valid modifier for all `ip6_*` options. But only the `ip6_optval` option requires it as a seperator. – In all other cases the test for equality is the default, so the operator is redundant (i. e. `ip6_extnum: 2;` and `ip6_extnum: =2;` are the same).

23 Including SEND and Mobile IPv6 messages. `ip6_nd;` is basically a shortcut to select/negate the ICMPv6 types 133–137, 141–142, 147–149, or 154.

20 About the Author

Martin Schütte studied political science and computing science in Potsdam. This article is an edition of his Diploma Thesis (*Diplomarbeit*) at the Chair of Operating Systems and Distributed Systems at Potsdam University. He is a system administrator and contributor to different open source software projects. Currently he is working as a consultant for DECK36 in Hamburg.

You can contact the author at info@mschuette.name.

The plugin's source code is available at https://github.com/mschuett/spp_ipv6.

References

Abley, J., Savola, P. & Neville-Neil, G. (2007, December). Deprecation of Type 0 Routing Headers in IPv6. RFC 5095 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc5095

Amante, S., Carpenter, B., Jiang, S. & Rajahalme, J. (2011). IPv6 Flow Label Specification. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-ietf-6man-flow-3697bis-09

Arkko, J., Aura, T., Kempf, J., Mäntylä, V.-M., Nikander, P. & Roe, M. (2002). Securing IPv6 neighbor and router discovery. In *Proceedings of the 1st ACM workshop on Wireless security* (Pages 77–86). ACM. Retrieved from http://koti.welho.com/pnikande/publications/WiSe2002-Arkko.pdf

Arkko, J., Kempf, J., Zill, B. & Nikander, P. (2005, March). SEcure Neighbor Discovery (SEND). RFC 3971 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3971

Arkko, J., Nikander, P., Kivinen, T. & Rossi, M. (2003, March). Manual Configuration of Security Associations for IPv6 Neighbor Discovery. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-arkko-manual-icmpv6-sas-02

Aura, T. (2003). Cryptographically Generated Addresses (CGA). In C. Boyd & W. Mao (Editors), *Information Security* (Volume 2851, Pages 29–43). Lecture Notes in Computer Science. Berlin/Heidelberg: Springer. Retrieved from http://dx.doi.org/10.1007/10958513_3

Aura, T. (2005, March). Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard). Updated by RFCs 4581, 4982. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3972

Bace, R. G. (2000). *Intrusion Detection*. Macmillan technology series. Indianapolis: Macmillan Technical Publishing.

Beale, J., Baker, A. R., Caswell, B., Alder, R. & Poor, M. (2004). *Snort 2.1 Intrusion Detection*. Open source security series. Burlington: Syngress.

Beale, J., Baker, A. R., Esler, J. & Northcutt, S. (2007). *Snort: IDS and IPS toolkit*. Jay Beale's open source security series. Burlington: Syngress.

Bechtold, T. & Heinlein, P. (2004). *Snort, Acid & Co*. Munich: Open Source Press. Retrieved from http://www.fosdoc.de/downloads/OSP_heinlein-bechtold_snort.pdf

Beck, F., Cholez, T., Festor, O. & Chrisment, I. (2007, March 6). Monitoring the Neighbor Discovery Protocol. In *The Second International Workshop on IPv6 Today - Technology and Deployment - IPv6TD 2007*. Guadeloupe/French Caribbean Guadeloupe. Retrieved from http://hal.inria.fr/inria-00153558/PDF/IPv6TD07_beck.pdf

Biondi, P. (2006, September). *Scapy and IPv6 Networking*. presented at Hack in the Box Security Conference 2006, Kuala Lumpur, Malaysia. Retrieved from http://www.secdev.org/conf/scapy-IPv6_HITB06.pdf

Biondi, P. & Ebalard, A. (2007). IPv6 Routing Header Security. Retrieved from http://www.secdev.org/conf/IPv6_RH_security-csw07.pdf

Borman, D. A., Deering, S. E. & Hinden, R. M. (1999, August). IPv6 Jumbograms. RFC 2675 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc2675

Braun, L., Didebulidze, A., Kammenhuber, N. & Carle, G. (2010). Comparing and improving current packet capturing solutions based on commodity hardware. In *Proceedings of the 10th annual conference on Internet measurement* (Pages 206–217). ACM. Retrieved from http://conferences.sigcomm.org/imc/2010/papers/p206.pdf

BSI und ConSecur GmbH. (2002). BSI-Leitfaden zur Einführung von Intrusion-Detection-Systemen. Retrieved from https://www.bsi.bund.de/ContentBSI/Publikationen/Studien/ids02/index_htm.html

Cheneau, T. & Combes, J.-M. (2008, October). Une attaque par rejeu sur le protocole SEND. In *3ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information*. Loctudy, France: Editions Publibook. Retrieved from https://www - public . it - sudparis . eu / ~cheneau / papers/article-SAR-SSI-2008.pdf

Chown, T. & Venaas, S. (2011, February). Rogue IPv6 Router Advertisement Problem Statement. RFC 6104 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6104

Davies, E. B., Krishnan, S. & Savola, P. (2007, September). IPv6 Transition/Co-existence Security Considerations. RFC 4942 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4942

Davies, E. B. & Mohacsi, J. (2007, May). Recommendations for Filtering ICMPv6 Messages in Fire-

walls. RFC 4890 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4890

Davies, J. (2008). *Understanding IPv6* (2nd). Microsoft Press Series. Redmond: Microsoft Press.

Deering, S. E., Fenner, W. C. & Haberman, B. (1999, October). Multicast Listener Discovery (MLD) for IPv6. RFC 2710 (Proposed Standard). Updated by RFCs 3590, 3810. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc2710

Deering, S. E. & Hinden, R. M. (1998, December). Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard). Updated by RFCs 5095, 5722, 5871. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc2460

Doraswamy, N. & Harkins, D. (1999). *IPSec: The New Security Standard for the Internet, Intranets, and Virtual Private Networks*. Upper Saddle River: Prentice Hall.

Droms, R. (1997, March). Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard). Updated by RFCs 3396, 4361, 5494. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc2131

Droms, R. & Arbaugh, B. (2001, June). Authentication for DHCP Messages. RFC 3118 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3118

Droms, R., Bound, J., Volz, B., Lemon, T., Perkins, C. E. & Carney, M. (2003, July). Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315 (Proposed Standard). Updated by RFCs 4361, 5494. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3315

Ebalard, A., Combes, J.-M., Boudguiga, A. & Maknavicius, M. (2009). IPv6 autoconfiguration mechanisms security (node part). Retrieved from http://www.mobisend.org/SP2.pdf

Ebalard, A., Combes, J.-M., Charfi, M., Maknavicius, M. & Fainelli, F. (2009). IPv6 autoconfiguration mechanisms security (router part). Retrieved from http://www.mobisend.org/SP1a.pdf

Frankel, S., Graveman, R. & Pearce, J. (2010, December). Guidelines for the Secure Deployment of IPv6. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-119/sp800-119.pdf

Gont, F. (2011a). IPv6 Router Advertisement Guard (RA-Guard) Evasion. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-gont-v6ops-ra-guard-evasion-00

Gont, F. (2011b). Security Implications of the Use of IPv6 Extension Headers with IPv6 Neighbor Discovery. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-gont-6man-nd-extension-headers-01

Hardjono, T. & Weis, B. (2004, March). The Multicast Group Security Architecture. RFC 3740 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3740

Heuse, M. (2010, December). Recent advances in IPv6 insecurities. In *27th Chaos Communication Congress*. also presented at the IPv6 Workshop, Beuth-Hochschule, 17 February 2011. Berlin. Retrieved from http://events.ccc.de/congress/2010/Fahrplan/attachments/1808_vh_thc-recent_advances_in_ipv6_insecurities.pdf

Hinden, R. M. & Deering, S. E. (2006, February). IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard). Updated by RFCs 5952, 6052. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4291

Hogg, S. & Vyncke, E. (2009, January). *IPv6 Security*. Indianapolis: Cisco Press.

Hu, Q. & Carpenter, B. (2011, June). Survey of Proposed Use Cases for the IPv6 Flow Label. RFC 6294 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6294

Jankiewicz, E., Loughney, J. & Narten, T. (2011, May). IPv6 Node Requirements. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-ietf-6man-node-req-bis-11

Jeong, J. P., Park, S. D., Beloeil, L. & Madanapalli, S. (2010, November). IPv6 Router Advertisement Options for DNS Configuration. RFC 6106 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6106

Jiang, S. & Shen, S. (2011). Secure DHCPv6 Using CGAs. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-ietf-dhc-secure-dhcpv6-03

Jiang, S., Shen, S. & Chown, T. (2011). DHCPv6 and CGA Interaction: Problem Statement. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-ietf-csi-dhcpv6-cga-ps-07

Johnson, D. B., Perkins, C. E. & Arkko, J. (2004, June). Mobility Support in IPv6. RFC 3775 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3775

Kent, S. & Seo, K. (2005, December). Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard). Updated by RFC 6040. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4301

Kohno, M., Nitzan, B., Bush, R., Matsuzaki, Y., Colitti, L. & Narten, T. (2011, April). Using 127-Bit IPv6 Prefixes on Inter-Router Links. RFC 6164 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6164

Krishnan, S. (2009, December). Handling of Overlapping IPv6 Fragments. RFC 5722 (Proposed

Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc5722

Krishnan, S. (2011, June). The case against Hop-by-Hop options. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-krishnan-ipv6-hopbyhop-05

Krishnan, S., Thaler, D. & Hoagland, J. (2011, April). Security Concerns with IP Tunneling. RFC 6169 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6169

Krishnan, S., Woodyatt, J. H., Kline, E., Hoagland, J. & Bhatia, M. (2011, March). An uniform format for IPv6 extension headers. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-ietf-6man-exthdr-06

Kumari, W., Gashinsky, I. & Jaeggli, J. (2011, June). Operational Neighbor Discovery Problems and Enhancements. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-gashinsky-v6nd-enhance-00

Lazarević, A., Kumar, V. & Srivastava, J. (2005). Intrusion Detection: A Survey. In *Managing Cyber Threats: issues, approaches, and challenges* (Volume 5, Pages 19–78). Massive Computing. New York: Springer. Retrieved from http://dx.doi.org/10.1007/0-387-24230-9_2

Levy-Abegnoli, E., de Velde, G. V., Popoviciu, C. & Mohacsi, J. (2011, February). IPv6 Router Advertisement Guard. RFC 6105 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6105

Li, X., Bao, C. & Baker, F. (2011, April). IP/ICMP Translation Algorithm. RFC 6145 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc6145

Malone, D. (2008). Observations of IPv6 addresses. In *Proceedings of the 9th international conference on Passive and active network measurement* (Pages 21–30). PAM'08. Cleveland, OH, USA: Springer. Retrieved from http://eprints.nuim.ie/1470/

McCanne, S. & Jacobson, V. (1993). The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*. Berkeley, CA, USA: USENIX Association. Retrieved from http://www.usenix.org/publications/library/proceedings/sd93/mccanne.pdf

McGann, O. & Malone, D. (2006). Flow Label Filtering Feasibility. In A. Blyth (Editor), *First European Conference on Computer Network Defence (EC2ND 2005)* (Pages 41–49). London: Springer. Retrieved from http://eprints.nuim.ie/1505/

McHugh, J. (2000, November). Testing intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. In *ACM Transactions on Information and System Security* (Volume 3, Pages 262–294). New York: ACM. Retrieved from http://doi.acm.org/10.1145/382912.382923

Mell, P. [Peter], Hu, V., Lippmann, R., Haines, J. & Zissman, M. (2003, June). An Overview of Issues in Testing Intrusion Detection Systems. Retrieved from http://csrc.nist.gov/publications/nistir/nistir-7007.pdf

Montenegro, G., Kushalnagar, N., Hui, J. W. & Culler, D. E. (2007, September). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4944

Moore, N. (2006, April). Optimistic Duplicate Address Detection (DAD) for IPv6. RFC 4429 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4429

Narten, T., Draves, R. & Krishnan, S. (2007, September). Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4941

Narten, T., Nordmark, E., Simpson, W. A. & Soliman, H. (2007, September). Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard). Updated by RFC 5942. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4861

Nikander, P. (2002). Denial-of-Service, Address Ownership, and Early Authentication in the IPv6 World. In B. Christianson, J. Malcolm, B. Crispo & M. Roe (Editors), *Security Protocols* (Volume 2467, Pages 12–21). Lecture Notes in Computer Science. Berlin/Heidelberg: Springer. Retrieved from http://koti.welho.com/pnikande/publications/cam2001.pdf

Nikander, P., Kempf, J. & Nordmark, E. (2004, May). IPv6 Neighbor Discovery (ND) Trust Models and Threats. RFC 3756 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3756

Novak, J. (2005, April). Target-Based Fragmentation Reassembly. Columbia: Sourcefire Inc. Retrieved from http://www.snort.org/assets/165/target_based_frag.pdf

Olney, M. (2008). *Performance Rules Creation (part 1)*. Retrieved from http://assets.sourcefire.com/snort/vrtwhitepapers/performance_rules_creation_1.pdf

Park, S. D., Kim, K.-H., Haddad, W. M., Chakrabarti, S. & Laganier, J. (2011). IPv6 over Low Power WPAN Security Analysis. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-daniel-6lowpan-security-analysis-05

Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. *Computer Networks*, *31*(23-

24), 2435–2463. Retrieved from http://www.icir.org/vern/papers/bro-CN99.html

Ramachandran, V. & Nandi, S. (2005). Detecting ARP Spoofing: An Active Technique. In S. Jajodia & C. Mazumdar (Editors), *Information Systems Security* (Volume 3803, Pages 239–250). Lecture Notes in Computer Science. Berlin/Heidelberg: Springer. Retrieved from http://www.vivekramachandran.com/docs/arp-spoofing.pdf

Roesch, M. (1999). Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX conference on System administration* (Pages 229–238). Retrieved from http://www.usenix.org/publications/library/proceedings/lisa99/roesch.html

Sarikaya, B., Xia, F. & Zaverucha, G. (2011). Lightweight Secure Neighbor Discovery for Low-power and Lossy Networks. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-sarikaya-6lowpan-cgand-01

Savola, P. (2002). Security of IPv6 Routing Header and Home Address Options. (Internet-Draft). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/draft-savola-ipv6-rh-ha-security-03

Savola, P. (2003, September). Use of /127 Prefix Length Between Routers Considered Harmful. RFC 3627 (Informational). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3627

Scarfone, K. & Mell, P. [P.]. (2007, February). Guide to Intrusion Detection and Prevention Systems. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf

Schneider, F., Wallerich, J. & Feldmann, A. (2007). Packet capture in 10-gigabit Ethernet environments using contemporary commodity hardware. In S. Uhlig, K. Papagiannaki & O. Bonaventure (Editors), *Proceedings of the 8th international conference on Passive and Active Network Measurement* (Volume 4427, Pages 207–217). Lecture Notes in Computer Science. Berlin / Heidelberg: Springer. Retrieved from http://dx.doi.org/10.1007/978-3-540-71617-4_21

Thomson, S., Narten, T. & Jinmei, T. (2007, September). IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc4862

Vida, R., Costa, L. H. M. K., Fdida, S., Deering, S., Fenner, B., Kouvelas, I. & Haberman, B. (2004, June). Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810 (Proposed Standard). Updated by RFC 4604. Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc3810

Weis, B., Gross, G. & Ignjatic, D. (2008, November). Multicast Extensions to the Security Architecture for the Internet Protocol. RFC 5374 (Proposed Standard). Internet Engineering Task Force. IETF. Retrieved from http://tools.ietf.org/html/rfc5374

Zanero, S. (2007, June). Flaws and frauds in the evaluation of IDS/IPS technologies. In *FIRST 2007 Security Conference*. Seville. Retrieved from http://members.first.org/conference/2007/papers/zanero-stefano-paper.pdf