Developing an Advanced IPv6 Evasion Attack Detection Framework

Mostafa Tajdini

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of Doctor of Philosophy

August 2018

To my dad, Mohammad Hossein

To my mom, Firouzeh

To my brothers, Amir Hossein and Mohammad Amin

To my grandparents, Fereydoun and Seyedeh Masoumeh

To my secret beloved one, N.

For their support all these years. Without their help, I could not be in this stage.

Acknowledgement

Firstly, I would like to express my sincere gratitude to the School of Computing and Mathematical Sciences at Liverpool John Moores University for giving me the opportunities to undertake this research degree.

I would like to thank my supervisory team, Dr. Hoshang Kolivand, Dr. Kashif Kifaiat and Dr Bo Zou for their continuous support of my PhD study, for their patience, motivation, and immense knowledge. Their guidance helped me to continually progress and build upon my ideas, ultimately resulting in the completion of this thesis.

I would like to thank Professor Qi Shi for his continued advice and support.

I would like to thank, Ahmad Milani and Mohammad Khoshzaban for their continued support.

I would like to thank Mrs Tricia Waterson for her support during my study at LJMU.

Abstract

Internet Protocol Version 6 (IPv6) is the most recent generation of Internet protocol. The transition from the current Internet Version 4 (IPv4) to IPv6 raised new issues and the most crucial issue is security vulnerabilities. Most vulnerabilities are common between IPv4 and IPv6, e.g. Evasion attack, Distributed Denial of Service (DDOS) and Fragmentation attack. According to the IPv6 RFC (Request for Comment) recommendations, there are potential attacks against various Operating Systems. Discrepancies between the behaviour of several Operating Systems can lead to Intrusion Detection System (IDS) evasion, Firewall evasion, Operating System fingerprint, Network Mapping, DoS/DDoS attack and Remote code execution attack. We investigated some of the security issues on IPv6 by reviewing existing solutions and methods and performed tests on two open source Network Intrusion Detection Systems (NIDSs) which are Snort and Suricata against some of IPv6 evasions and attack methods. The results show that both NIDSs are unable to detect most of the methods that are used to evade detection. This thesis presents a detection framework specifically developed for IPv6 network to detect evasion, insertion and DoS attacks when using IPv6 Extension Headers and Fragmentation. We implemented the proposed theoretical solution into a proposed framework for evaluation tests. To develop the framework, “dpkt” module is employed to capture and decode the packet. During the development phase, a bug on the module used to parse/decode packets has been found and a patch provided for the module to decode the IPv6 packet correctly. The standard unpack function included in the “ip6” section of the “dpkt” package follows extension headers which means following its parsing, one has no access to all the extension headers in their original order. By defining, a new field called all_extension_headers and adding each header to it before it is moved along allows us to have access to all the extension headers while keeping the original parse speed of the framework virtually untouched. The extra memory footprint from this is also negligible as it will be a linear fraction of the size of the whole set of packet. By decoding the packet, extracting data from packet and evaluating the data with user-defined value, the proposed framework is able to detect IPv6 Evasion, Insertion and DoS attacks. The proposed framework consists of four layers. The first layer captures the network traffic and passes it to second layer for packet decoding which is the most important part of the detection process. It is because, if NIDS could not decode and extract the packet content, it would not be able to pass correct information into the Detection Engine process for detection. Once the packet has been decoded by the decoding process, the decoded packet will be sent to the third layer which is the brain of the proposed solution to make a decision by evaluating the information with the defined value to see whether the packet is threatened or not. This layer is called the Detection Engine. Once the packet(s) has been examined by detection processes, the result will be sent to output layer. If the packet matches with a type or signature that system admin chose, it raises an alarm and automatically logs all details of the packet and saves it for system admin for further investigation. We evaluated the proposed framework and its subsequent process via numerous experiments. The results of these conclude that the proposed framework, called NOPO framework, is able to offer better detection in terms of accuracy, with a more accurate packet decoding process, and reduced resources usage compared to both exciting NIDs.

Contents

Publications

- Kolivand, H., El Rhalibi, A. Tajdini, M. Abdulazeez, S. and Praiwattana, P., 2018. Cultural Heritage in Marker-Less Augmented Reality: A Survey. In Advanced Methods and New Materials for Cultural Heritage Preservation. IntechOpen.
- Mostafa Tajdini, Hoshamg Kolivand, Kashif Kifayat, and Bo Zhou. IPv6 New Features or New Vulnerabilities, 5th International Conference on Interactive Digital Media (ICIDM), ICIDM2018, LJMU, U.K – Accepted
- Mostafa Tajdini, Hoshamg Kolivand, Kashif Kifayat, and Bo Zhou. Detecting IPv6 Advance Evasion Detection Techniques, LNCS Transactions on Edutainment, Springer – Accepted
- Mostafa Tajdini, Hoshamg Kolivand, Yousef Al-Hamar, Amjad Rehman. Detecting IPv6 Evasion Taking Extension Headers Into Account, Information Systems Journal, - Submitted
- Mostafa Tajdini, Hoshamg Kolivand, Yousef Al-Hamar, Amjad Rehman. Detecting IPv6 Evasion, Insertion and DoS Attack Taking Fragmentation into Account, Networks, - Submitted

List of Figures

List of Tables

List of Abbreviations

IP:	Internet Protocol
IPv4:	Internet Protocol Version 4
IPv6:	Internet Protocol Version 6
DoS:	Denial of Service
DDoS:	Distributed Denial of Service
IDS:	Intrusion Detection Systems
NIDS:	Network Intrusion Detection Systems
HIDS:	Host-based Intrusion Detection System
TTL:	Time to Live
OS:	Operating System
IPS:	Intrusion Prevention System
ICMP:	Internet Control Message Protocol
ICMPv6:	Internet Control Message Protocol Version 6
TCP:	Transmission Control Protocol
UDP:	User Datagram Protocol
RFC:	Request for Comment
DARPA:	Defence Advanced Research Projects Agency
SYN:	Synchronize
ACK:	Acknowledge
EH:	Extension Header
RS:	Router Solicitation
RA:	Router Advertisement
NS:	Neighbour Solicitation
NA:	Neighbour Advertisement
SSH:	Secure Shell
URL:	Uniform Resource Locator

Chapter 1
Introduction

1.1. Introduction

Our world has become a big network where everyone connect to it by the Internet. Most people living on the earth rely on this network. They read news, transfer money, and check their emails and much more in their daily basic life. The goal of the new modern world is the availability, integrity and confidentiality of this network. The rapid growth and widespread use of electronic devices and data processing (cloud computing, web application, internet network, wireless networks, and private network) will raise the need for a solution that can provide a safe and secure infrastructure for a safe communication.

To use the Internet, each device needs to have an Internet Protocol (IP) address. An IP address is a unique number that is assigned to every device that is connected to the network or Internet. The IP address enables our devices to communicate with each other. There are two different version of IPs, Internet Protocol Version 4 (IPv4) and Internet Protocol Version 6 (IPv6). IPv4 was developed in the early 1980s, but because of the rapid growth of the internet, IPv4 has been fully allocated to Internet Services Providers and Internet users, and then there was a shortage of IPv4 available address [1].

IPv6 was standardized in 1996 to replace the current version of IPv4 and covers the biggest limitation of IPv4 which is the lack of enough addresses for all internet users. In recent years, the major service providers have started to offer IPv6 addresses to their users [1]. Based on a report from Google on 16 Aug 2018, 23.91% of the users that access Google are over IPv6. This report shows how usage of IPv6 has grown during the last couple of years.

The transition from IPv4 to IPv6 should have eliminated any related security issue to the new protocol. The security mechanisms for network layer protocol should be examined in many different areas. One of these areas is how Operating Systems handle the IPv6 fragmented packet and how Network Intrusion Detection Systems can detect an attack on the IPv6 network.

If used properly by an attacker, this feature in IPv6 can lead to Network Intrusion Detection System (NIDS) evasion, Firewall evasion, Operating System fingerprint, Network Mapping, Denial of Service (DoS)/ Distributed Denial of Service (DDoS) attack and Remote code execution attack.

Here is where the Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) take place to provide an early detection or/and prevention to detect, prevent and reduce more serious damage to the network and their users while keeping those networks safer. IDS and IPS can be used to record forensic evidence and data in case of any criminal breaches. The IDS evasion problem gets more complicated in the meantime due to the complex and rapid growth of the network and the structure of IPv6. The evasion techniques have different types such as IP fragmentation, DoS/DDoS attack, Internet Control Message Protocol (ICMP) attack, TTL evasion attack and encrypted traffic [2] [3] [56] [107] .

Insertion, Evasion and Denial of Service are three different categories of attacks, which were proposed by Ptacek et al. [54] for the first time. The aim of these attacks is to make the IDS or victim host process different data or process the same data but differently [54]. By using an insertion attack, IDS accepts a packet(s) that is rejected by the host. The packet looks valid only to the IDS. The attacker can bypass the signature based IDS by inserting the traffic in such way that the signature is never matched or found. This process is different in Evasion attack; in evasion attack the IDS rejects the packet that the end host accepts. The attacker can send some or all malicious traffic into the network without being caught by the IDS. The main reason behind the insertion or evasion attack are ambiguities in traffic which make NIDS or the end host have a different view of the same data stream and this can be caused by differences in the protocol implementation. As explained in Request for Comment (RFC1858), firewall evasion can happen by using IP fragmentation to hide IP/ Transmission Control Protocol (TCP) packets from IP filters used in firewall, detection system and hosts.

Apart from expanded addressing capabilities, one of most important and significant changes in IPv6 is the improvement of supporting extension header with the options (RFC 2460 [20]). In IPv4, some of the header fields have been dropped to reduce the cost of packet handling and processing. However, this has been changed in IPv6 where the Extension Headers function is added. Each IPv6 packet can have zero, one or more extension headers.

Most of common NIDS and Firewalls do the packet reassembly for fragmented packets, but the implementation of how they do the reassembly is different. That is one of the reasons which makes IPv6 evasion detection difficult for them.

An Evasion can occur on three levels:

- Hardware Level – Level 1-2
- Operating System (OS) Level – Level 3-4
- Application Level – Level 5-7

This thesis does not focus on Hardware Level evasion. We have categorised the most common attack techniques on OS layer and Application layer as follows:

- **Insertion**: happens when IDS accepts a packet, that the end-host rejects. In this method, IDS believes that end host has accepted the packet and processed it.
- **Evasion**: is quite similar to Insertion attack, but this time the end-host accepts the packet that the IDS has rejected, however because the end-system accepted the packet that the IDS rejected, therefore the information that the IDS misses is critical to the detection of an attack and an evasion attack can occur.
- **Denial of Service (DoS)**: is causing end-host resource outage by sending too many requests to the end-host.
- **Encryption and tunnelling**: Attacker hides their activity on an encrypted traffic where the IDS cannot see the traffic content.
- **Fragmentation**: when attacker breaks the attack into multiple parts to evade detection.

The outcomes of our literature review showed us that most of the security issues inherited from IPv4 to IPv6 such as DoS/DDoS, IP Fragmentation, TTL Evasion and Source routing attack. Such attack can lead to IDS evasion, Firewall evasion, Operating System fingerprint, Network Mapping, DoS/DDoS attack and Remote code execution attack [109] [112-113] [115-118] [142].

An example of using a mixture of methods (IP Fragmentation and DoS) can be the created Smurf attack. In normal circumstances NIDS are able to detect simple and normal Smurf attacks. By breaking and hiding the attack into IPv6 Extension Headers, the attacker can easily evade detection and perform a successful attack.

We found that the current gaps in this area are: limited research on evasion techniques that can be used to bypass detection or prevention systems; lack of good methods for packet decoding and lack of implemented solution into a real world scenario to test their detecting effectiveness. Moreover, most of the methods and solutions evaluated their solutions with sample data such as DARPA98 and KDD99Cup, rather than evaluating by practical tests and methods available on the Internet or creating their own test. By reviewing those solutions and methods, we have provided a framework that improves the result of detection compared with other Network Intrusion Detection Systems (NIDS). In addition, our solution works on detecting ICMPv6 Flooding and Amplification attack when attackers use fragmentation to evade detection [111-113] [127-129].

We can categorised the problem areas as follows:

Evasion: bypass NIDS with valid packets [6]
Insertion: bypass NIDS with invalid packets [6]
Denial of Service (DoS): using method 1 and 3 to bypass detection and perform DoS/DDoS attack (ICMPv6 Flood attack and Smurf Attack) [117-118].
Fragmentation: Breaking attacks into multiple packets [6,107,112]

To tackle these problems, we proposed a new solution to combat them. The proposed framework has four processes, Packet Capturing, Packet Decoder, Detection Engine and Output. One of the advantages of our solution is that can work with live traffic or offline traffic. When traffic reaches the packet decoder, it will start to decode the packet and extract the information from the packet. Once the packet has been successfully decoded, it will send all data to the Detection Engine. To detect an attack, the user should select one of the following options:

- Rule: Detection Engine will start to match the traffic with the existing rule file.
- Signature: Signature of an specific attack that the user is looking for
- Type: This option is useful for detection of ICMPv6 attack, as it will be looking for the packet type.

During detection if there is any match found, the framework will send a message to the Output process and Output will generate an alarm and save the details of that packet into a log file for review.

During our experimental tests, we used Snort and Suricata beside our framework. We sent different packets with different techniques and method, but with the same data, which was a ping request to the end host. We sent those packets three times to see which one could bypass the detection. We found that Snort only detected 41%, Suricata detect 70% and our framework detected all evasion packets. In addition, our framework was able to detect ICMPv6 Amplification and DoS attack where the others could not.

1.2. Problem Statement

By completing the literature review, we found that most security issues were inherited by IPv6 from IPv4, such as DoS/DDoS, IP Fragmentation, TTL Evasion and Source routing attack. As explained in the introduction, apart from expanded addressing capabilities, one of most important and significant changes in IPv6 is the improvement of supporting extension headers. As the implementation of IPv6 is different on each system and because most NIDSs and Firewall do the packet reassembly for fragmented packets, this can be used by attackers to perform an attack. Such attacks can lead to IDS evasion, Firewall evasion, Operating System fingerprint, Network Mapping, DoS/DDoS attack and Remote code execution attack. To prove this we provided comparative detection analysis results with two Open Source NIDSs, Snort and Suricata. By comparing the results, we found both NIDSs are vulnerable to advanced attack method (such as using a mixture of packets with different extension headers or moving fragmented packet order (fragment overlapping)). In addition we found that there is limited technical research for IPv6 security challenges, limited technical research for IPv6 evasion detection methods and there are a limited number of Network Intrusion Detection System (for test and research purposes).

A good technical example is during the test experiment we found Snort only able to decode packets with nine or less extension headers in each packet and could not decode the packet with more than nine extension headers.

1.3. Aims and Objectives

The aims of this research are:

- To develop a network-based solution to detect IPv6 Evasion techniques
- To improve evasion detection accuracy based on IPv6 when using fragmentation and Extension Headers and to detect Evasion, Insertion, DoS and ICMPv6 Amplification attacks
- To improve IPv6 packet decoding accuracy

The main objectives of this thesis are:

- To identify the limitations of existing solutions - demonstrated in demonstrated in Chapter 3
- To investigate IPv6 evasion, detection methods and attack techniques - demonstrated in Chapter 2
- To improve IPv6 packet decoding (translating packets in correct format) - Demonstrated in § 4.3.2, [110]
- To propose a new framework that helps to detect IPv6 Evasion, Insertion and DoS that use Fragmentation and Extension headers. - Demonstrated in Chapter 4
- To detect some different evasion methods which are using Fragmentation and Extension Headers to evade Network Intrusion Detection Systems - Demonstrated in Chapter 5
- To present a new framework to detect evasion attacks in IPv6 network in real time traffic and offline traffic - Demonstrated in § 4.3.1

1.4. Research Contributions

This thesis makes the following contributions to the field of Network Intrusion Detection Systems to overcome the existing limitations:

1- Proposing a new framework that can detect evasion attacks in IPv6 network in real time traffic and offline traffic.

2- Detecting 37 different evasion methods and techniques. The test results in § 5.1.11 show the proposed solution is able to detect all those methods where the other two IDSs are unable to detect all those methods.
3- Improving a python module that is used in our framework for parsing, capturing and decoding packets. During the development stage, we found a bug in that module which would not let the user read and decode the IPv6 packet header and extension header correctly.
4- Proposing a system that is able to detect DoS/DDoS attack (ICMPv6 flooding, Amplification and Smurf attack) and OS Fingerprint method.

The outcome of this thesis is to highlight the current research gap, identify the existing solutions, designing and developing a framework that is able to detect evasion attack based on IPv6 and to improve IPv6 packet decoding.

During the research we found that by using some of the new features (fragmentation and Extension Headers) in IPv6, we can bypass the security devices like NIDSs. During our tests, we were able to evade detection completely and launch an attack.

To overcome the security challenges that we found during our research, we designed and developed a framework that increases the rate of evasion detection on IPv6 based on using fragmentation and extension headers.

1.5. Research Framework

To achieve the aims and objectives highlighted in Chapter 1, the following methodology is introduced for a successful achievement. This chapter will discuss the process of completing each objective (Figure 1.1).

The evaluation and analysis of the incoming network packets within a network using the proposed framework includes the following phases:

Phase 1: A detailed background research on IPs to help a better understanding how Internet Protocols (IPs) work. Then an analysis of current vulnerabilities in Internet Protocols and the tools that attackers use to perform an attack. The last two steps in Phase 1 are research and evaluating current methods that most of the Intrusion Detection Systems use to analyse and detect an attack. At the end of this phase, we achieved one of our objectives, which is investigating IPv6 evasion, detection and attack techniques.

Phase 2: A complete literature review of related works and existing solutions for both versions of IPs and then an evaluation of those methods and solutions to find the research gaps. At the end of this phase, we achieved another of the objectives which were highlighted in § 1.3 which is to identify the limitation of existing solutions.

Phase 3: Presenting a theoretical solution that later on, led us to develop a framework to detect IPv6 evasion and insertion techniques on IPv6 network. By concluding the last two phases, and finalising this phase, we achieved the rest of our aims and objectives. The achieved aims are to develop a network-based solution to detect IPv6 techniques, improve evasion detection accuracy based on IPv6 when using fragmentation and Extension Headers and to detect Evasion, Insertion, DoS and ICMPv6 Amplification attack and improve IPv6 packet decoding accuracy. The achieved Objectives are improving IPv6 packet decoding (translating packets in correct format), proposing a new framework that helps to detect IPv6 Evasion, Insertion and DoS that use Fragmentation and Extension headers, detecting some different evasion methods which are using Fragmentation and Extension Headers to evade Network Intrusion Detection System and presenting a new framework to detect evasion attacks in IPv6 network in real time traffic and offline traffic.

A schematic diagram showing the methodological steps, summary and the contribution of this research is presented in Figure 1.1, which draw the three phases involved and how the research is going to be implemented conceptually.

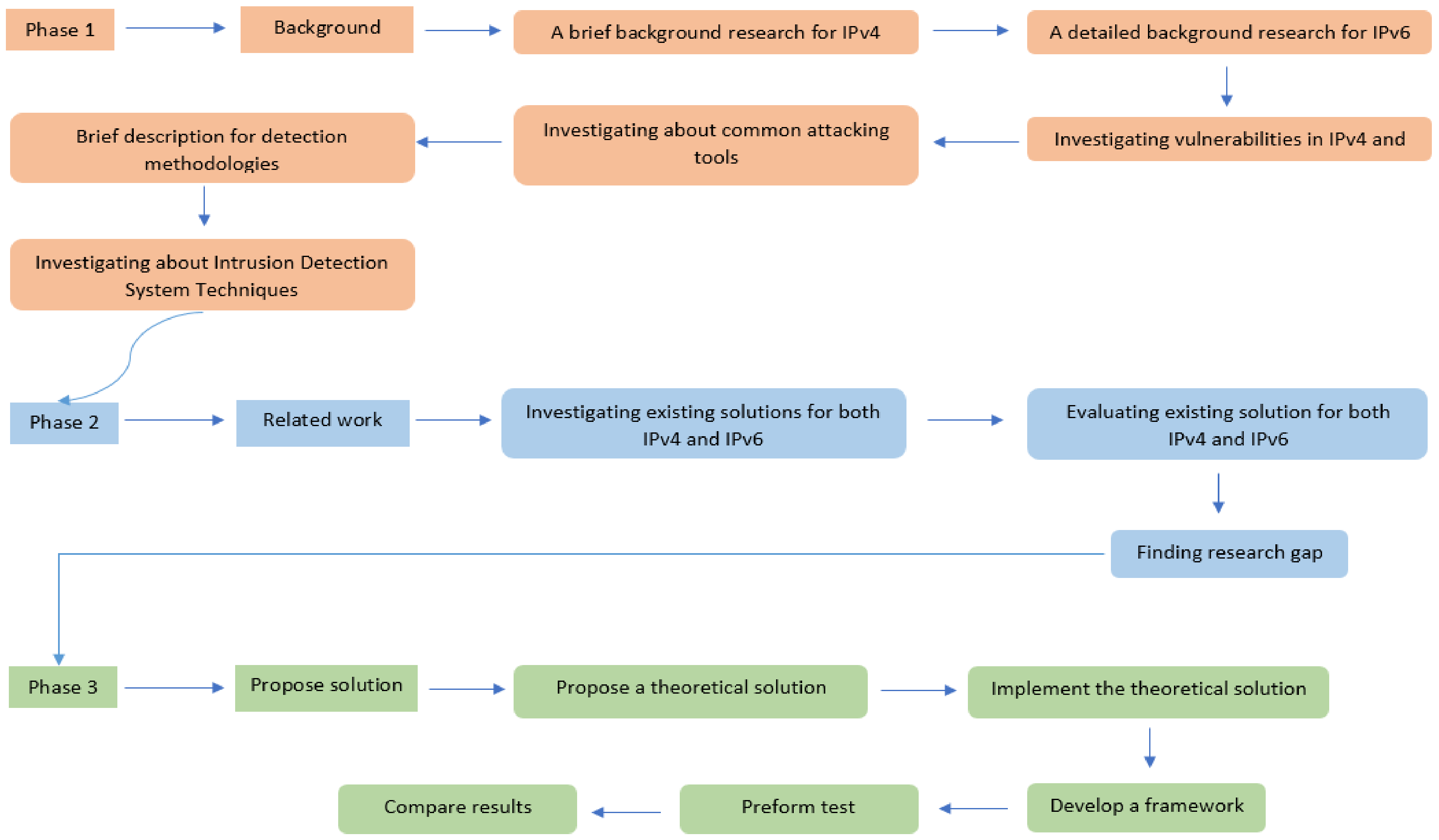

Phase 1
Background
A brief background research for IPv4
A detailed background research for IPv6
Investigating vulnerabilities in IPv4 and
Investigating about common attacking tools
Brief description for detection methodologies
Investigating about Intrusion Detection System Techniques
Phase 2
Related work
Investigating existing solutions for both IPv4 and IPv6
Evaluating existing solution for both IPv4 and IPv6
Finding research gap
Phase 3
Propose solution
Propose a theoretical solution
Implement the theoretical solution
Develop a framework
Preform test
Compare results

Figure 1.1: Research Framework

1.6. Thesis Structure

This thesis is arranged into six subsequent chapters; the order and contents of these chapters are as follows:

Chapter 1: Introduction

This chapter provided a brief summary of the whole thesis including the aims and objectives of this research and the contributions that this research will provide.

Chapter 2: Background

This chapter provides detailed background information on Internet Protocols (IP), Vulnerability in IPv4 and IPv6, The attacking tools, the detection methodologies and different types of Intrusion Detection Systems.

This gives the reader a better understanding of how the work in this thesis relates to inadequacies that currently exist.

Chapter 3: Related Work

This chapter presents a review of existing literature that focuses on the benefits and shortcomings of related solutions, which fuelled the motivation for the approach proposed in this thesis. This section focuses on existing solutions for both IPv4 and IPv6 and their applicability to detect evasion attacks.

Chapter 4: The Proposed Solution

This chapter presents the design, implementation and methodologies of the proposed framework, NOPO. The first section presents the proposed theoretical solution and techniques specifically proposed for this solution. This includes the Packet Capturing, Packet Decoding, Detection Engine and Output process.

The next section in this chapter provides an insight into the software developed as a tool for evaluating the proposed framework, showing that how the framework and techniques were implemented into the real world scenario.

Chapter 5: Tests and Results

This chapter shows the evaluation of the proposed solution, the evasion techniques, and uses some attacking tools that were demonstrated in Chapter 2 and then comparing the results with those from the other two existing solutions(Snort and Suricata).

Chapter 6: Conclusion and Feature work

This chapter summarises the findings of this research, describes the challenges that were previously identified and the extent of the success in overcoming those challenges. Also focusing on future work that could be carried out based on the results of this work or in relation to this work.

Chapter 2

Background

2.1. Introduction

This chapter provides background information on six main areas related to the work contained in this thesis, all of which is fundamental to understating the context of the challenges being addressed. Section (§) 2.2 provides basic information about IPv4 and a comparison to IPv6. §2.3 describes the IPv6 and its features and the difference between IPv4 and IPv6 header in technical terms. In §2.4 the vulnerabilities in both versions of IP's are described and explained. §2.5 explains some of the existing attacking tools, §2.6 relates to intrusion detection methodologies, §2.7 explains the differences between Intrusion Detection Systems.

2.2. IPv4 Background

Internet Protocol (IP) version 4 was developed in late 1970's - early 1980's under the project name "DARPA (Defence Advanced Research Projects Agency) Internet Program". The goal of this project was to create a "catenet" (network of networks). The motivation for this project was to develop a protocol for "*transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. The internet protocol also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet networks*" [51]. The Internet Protocol (IP) implements datagram fragmentation [48], breaking the packet into small pieces and sending it across different types of network media to arrive successfully at its intended destination. The different types of network media and protocols have different rules which involve allowing a maximum size of datagrams on their network segment. This is the reason for using fragmentation. This is known as the Maximum Transmission Unit (MTU).

The IP header is illustrated in Figure 2.2 below.

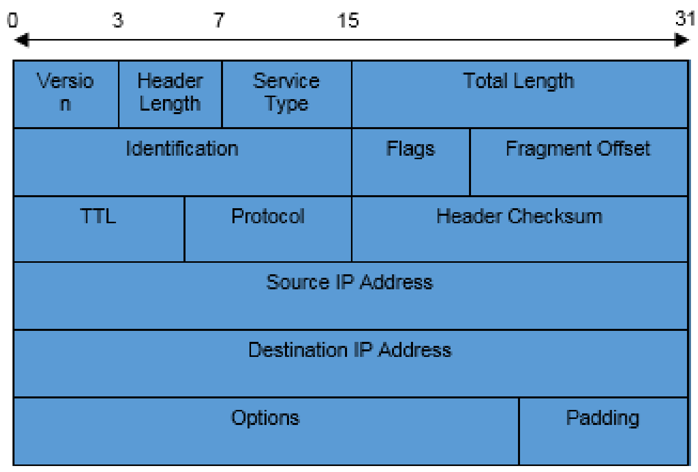


Figure 2.2: IPv4 header

- **Header length:** specifies the length of the IP header in 32-bit words. This includes the length of any options field and padding in 32bit words, if no options are used, the value of this field is usually 5 (5 x (32bit) = 20bytes) and for a maximum it can be 60 bytes (IP option).
- **Type of Service (TOS):** is used to provide quality of service features and to carry information for IP datagram and provide priority and request low-delay, highly reliable service, and high-throughput. TOS bit 1 (min delay), 2 (max throughput), 3 (max reliability), 4 (min cost)
- **Total length** specifies the length of the IP datagram in bytes. Since this field is 16 bits wide, the maximum length of an IP datagram is 216 = 65,535 bytes. Although, most are much smaller.
- **Header checksum** is a checksum of IPv4 header to protect the header against data corruption and it has a 16-bit length. The checksum is only calculated for the header bytes and if the value is zero this means there is no corruption. In the case of checksum mismatch, the packet will be discarded.
- **Identification** is a unique number to identify an IP datagram; fragments with the same identifier belong to the same IP datagram
- **Fragment** Will provide the position of the fragmented packet when the packet want to be reassembled by the end host or router. It is specified in units of 8 bytes (64bit).
- **FLAGs** are a three-bit field and are used to control or identify fragments.
    - Bit 0: reserved; must be zero

    - Bit 1: Don’t Fragment (DF)
    - Bit 2: More Fragments (MF)
- **Time to live** specifies how long the datagram is allowed to live on the network, in terms of router hops. Each router will decrement the value of the TTL field, prior to transmitting it, by one. When it reaches zero, the packet will be discarded.
- **Protocol identifies** the high-layer level protocol carried in the datagram. The values of this field were originally defined by the Internet Engineering Task Force (IETF) standard and now maintained by the Internet Assigned Numbers Authority (IANA).

2.1.1. IPv4 fragmentation

Transmitting a datagram across a network segment, which has an MTU smaller than that of the packet to be transmitted, would require fragmentation. For a successful packet reassembly at destination, each packet should follow the following rules; otherwise, the packet will be discarded:

- Each fragment must have fragment ID.
- Each fragment must have an offset.
- Each fragment must have the data length.
- Each fragment must have Flags

Fragmentation has several vulnerabilities that attackers can exploit to perform the attacks. Some of these vulnerabilities are:

- Ping of death (DoS or DDoS) [49]
- Tiny fragment attack [14]
- Teardrop attack (UDP attack) [50]
- Overlapping fragment attack

In a TCP protocol stack the most important layers for IP fragmentation attack are Transport layer and Network layer. IP fragmentation can be used to hide TCP packets from security devices.

RFC 791 [51] (the current IP protocol specification), describes a reassembly algorithm that results in new fragments overwriting any overlapped portions of previously received fragments.

As explained by Stallings [120] "a*n attacker could construct a series of packets in which the lowest (zero-offset) fragment would contain innocuous data (and thereby be passed by administrative packet filters), and in which some subsequent packet having a non-zero offset would overlap TCP header information (destination port, for instance) and cause it to be modified. The second packet would be passed through most filter implementations because it does not have a zero fragment offset."*

RFC 815 [51] outlines, "*An improved datagram reassembly algorithm, but it concerns itself primarily with filling gaps during the reassembly process. This RFC remains mute on the issue of overlapping fragments*".

The issue is when the first fragmented packet with SYN(Synchronize)=0, ACK(Acknowledge) =1 and OFFSET=0 arrives at the Firewall or Intrusion Detection System (IDS), it passes, however the second fragment which contains TCP Flags, is different from the first fragment (SYN=1 and ACK=0).

If the end-host has a reassembly algorithm that prevents new data from overwriting data received previously, attackers can send the second fragment first, followed by first fragment, and accomplish a successful overlapping attack.

In comparison to IPv4, IPv6 introduced four major changes:

- Address length to 128 bits long [15].
- The header format which has fixed length of 40 bytes and removed IP header option.
- Fragmentation, which only occurs at source and reassembly only at destinations
- Mandatory usage of IPsec [16] [17] [18] is seen as its fifth major modification before being released as optional [19].

2.3. IPv6 Background

Deployment of a new generation of IP (Internet Protocol) is on its way. On the 6th of June 2012, IPv6 was launched and since then, the traffic has grown more than 5000% and it is assumed that if the use of IPv6 continues, half of the current internet users will be using IPv6 in four years.

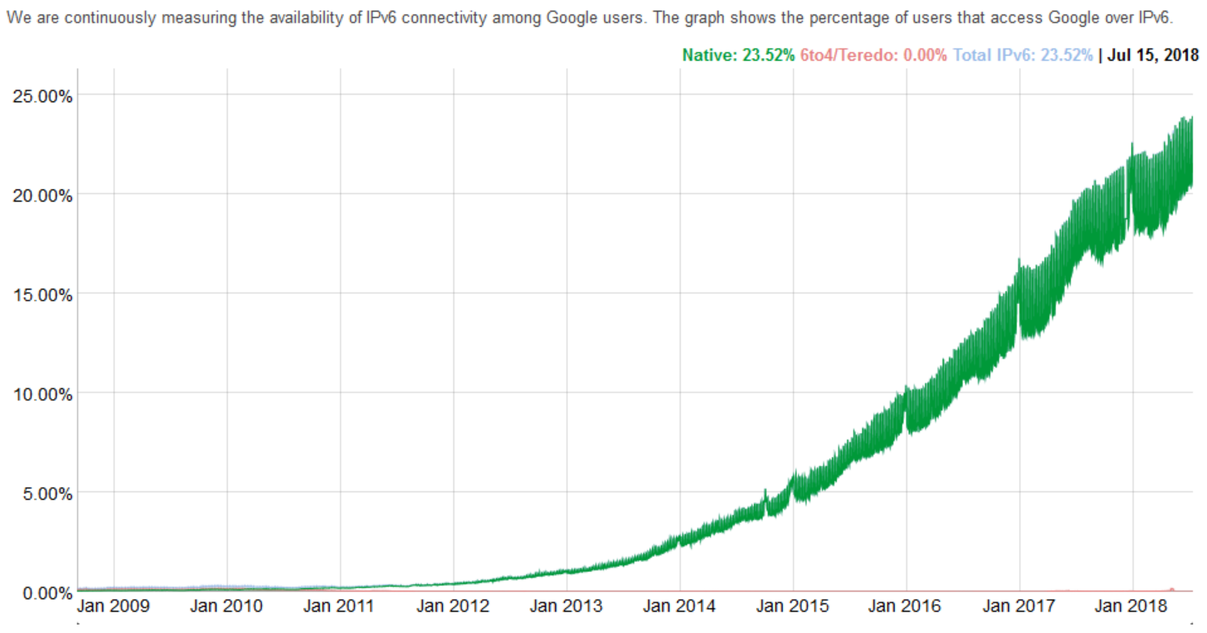


Figure 2.3: IPv6 adoption live chart by Google [94]

Since the world IPv6 launch began, as shown in Figure 2.3, IPv6 usage and connectivity started rapidly increasing as per the Google report on the 15th of July 2018 the usage was 23.52%.

Figure 2.4 shows a worldwide adoption of IPv6 based on compound of IPv6 metrics for a subset of countries provided by Cisco Lab. The countries that are shown in darker green have a higher percentage of deployment of IPv6. Based on the data provided by Cisco Lab on 17th of July 2018 Germany ranked number one in this adoption with a total of 56.57% IPv6 Deployment.

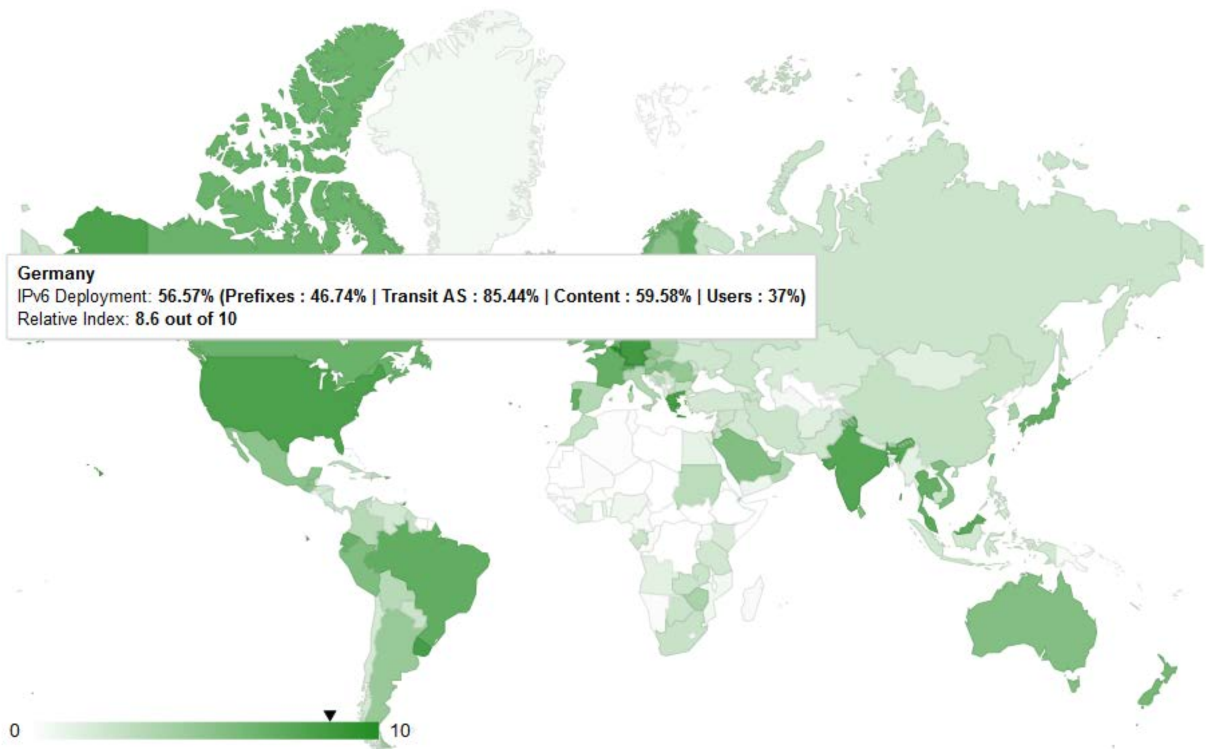

Germany
IPv6 Deployment: 56.57% (Prefixes : 46.74% | Transit AS : 85.44% | Content : 59.58% | Users : 37%)
Relative Index: 8.6 out of 10
0
10

Figure 2.4: Global IPv6 usage by Cisco lab [95]

With the rapid growth of IPv6 usage and shortage of addresses in IPv4, many internet service providers and organisation are changing to version 6 resulting in updates of related protocols.

The major changes between IPv4 and IPv6 can described as [55]:

- Extended addressing capabilities
- Header format simplification
- Improved support of Extension and Options
- Flow labelling capability
- Authentication and Privacy capabilities

Table 2.1 shows the IPv6 header format with size of each field and the field difference between both versions of IPs and Figure 2.5 shows the IPv6 header packer structure.

Table 2.1: IPv6 header format

Size(byte)	**Name**	**In IPv4**
4	Version	set to 6
8	Traffic Class	replaces Type of Services
20	Flow Label	for packet flow marking
16	Payload Length	incl. IPv6 Extension Headers
8	Next Header	
8	Hop Limit	replaces Time to Live
128	Source Address	
128	Destination Address	

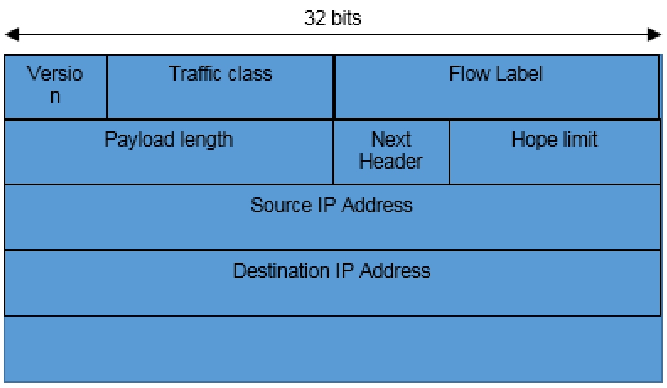


Figure 2.5: IPv6 header

One of the changes is that IPv6 Extension Headers, are encoded in separate headers that may be placed between the IPv6 header and the upper layer in a packet.

2.3.1. IPv6 Extension Headers

Apart from expanded addressing capabilities, one of most important and significant changes in IPv6 is the improvement of supporting extension headers with the options (RFC 2460 [20]). In IPv4 some of the header fields have been dropped to reduce the cost of packet handling and processing. However, this has been changed in IPv6 where the Extension Headers function has been added. The Extension Headers are placed between IPv6 header and the upper layer header in a packet and each of the Extension Headers is identified by a distinct next header value. “*As this is an optional field, each IPv6 packet may carry zero, one or more extension headers. Each Extension Header is a multiple of 8 octets long* (Figure 2.6). *Optional internet-layer information is encoded in separate headers that may be placed between the IPv6 header and the upper-layer header in a packet* [20].”

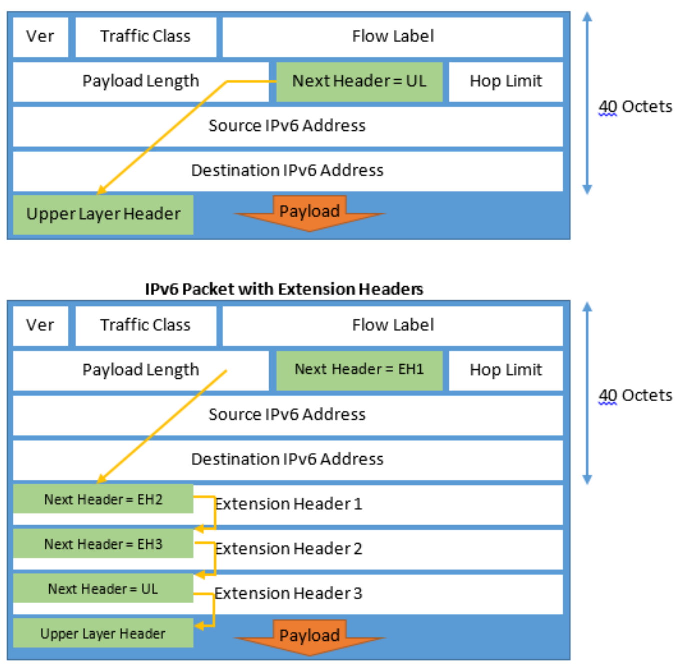


Figure 2.6: IPv6 packet with Extension Header

2.3.1.1. Hop-by-Hop Options header

The Hop-by-Hop Options header is used to carry optional information that must be examined by every node along a packet's delivery path. The Hop-by-Hop, extension header has the following format:

- Next Header: 8bit which identifies the header immediately following by Hop-by-Hop Options
- Header Extension Length: 8bit unsigned integer
- Options: The length of this field is variable and contains TLV-encoded options

2.3.1.2. Routing Header

The Routing header is used by an IPv6 source to list one or more intermediate nodes to be "visited" on the way to a packet's destination. This function is very similar to IPv4's Loose Source and Record Route option [20]. The length of Routing Header is 8bit and:

- Next Header: 8bit which identifies the header immediately following Routing Header
- Header Extension Length: 8bit unsigned integer
- Routing type: 8bit which is the identifier of a particular Routing header variant

- Segment Left: 8bit which is the number of the route segment remaining
- Type-specific data: The length of this field is variable and determined by the Routing Type, and of length such that the complete Routing header is an integer multiple of 8 octets long

2.3.1.3. The Fragment Header

The Fragment header is used by an IPv6 source address to send a packet larger than would fit in the MTU path to its destination. (Note: unlike IPv4, only source nodes perform fragmentation in IPv6, not routers along a packet's delivery path [20]. The length of Routing Header is 8bit and:

- Next Header: 8bit and identifies the initial header type of the Fragmentable Part of the original packet
- Reserved: 8bit reserve field and Initialized to zero for transmission; ignored on reception
- Fragment Offset: 13bit, the offset, in 8-octet units, of the data following this header, relative to the start of the Fragmentable Part of the original packet.
- Res: 2 bit, same as Reserved
- M flag: 1 means more fragments, 0 means last fragment
- Identification: Generated by source node for every packet that is to be fragmented

2.3.1.4. Destination Option Header

The Destination Option Header is used to carry optional information that needs to be examined only by a packet's destination node(s):

- Next Header: 8bit and identifies the header immediately following Destination Option header
- Header Extension Length: 8bit unsigned integer
- Options: The length of this field is variable and one or more TLV-encoded options

2.3.1.5. Encapsulation Header

The Encapsulation header is designed to provide a mix of security services in IPv4 and IPv6. ESP is used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service, and limited traffic flow confidentiality. ESP is similar in format and used for the IPv4 ESP header defined in RFC2406 [20].

2.3.1.6. Authentication Header

The Authentication header is used to provide connectionless integrity and data origin authentication for IP datagrams, and to provide protection against replays. This extension header is similar in format and used for the IPv4 authentication header defined in RFC2402 [79]

There are a small number of such extension headers, each identified by a distinct Next Header value.

- Hop-by-Hop Options Next Header value = 0
- Destination Options Next Header value = 60
- Routing Next Header value = 43
- Fragment Next Header value = 44
- Authentication Next Header value = 51
- Encapsulating Security Payload Next Header value = 50
- Destination Options Next Header value = 60
- upper-layer Next Header value = 135

RFC 2460 [20] saying *“Extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches the node except Hop-by-Hop Options header, which carries information that must be examined and processed by every node along a packet's delivery path, including the source and destination nodes.* [20]*”*

When more than one extension header is used in the same packet, RFC 2460 recommended that those headers appear in the following order:

- Hop-by-Hop Options header
- Destination Options header
- Routing header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header

- upper-layer header

2.3.2. IPv6 Addressing Architecture

As mentioned before, IPv6 length is 128-bit. The address fields use colons to separate each 16-bit hexadecimal number and for reducing the chance of error for the address representation, the hexadecimal numbers are not case sensitive as shown in below example.

3021:0000:120D:0000:0000:18F0:756B:125A.

In addition, in order to make the address representation of the IPv6 address easier and shorter , IPv6 uses the following conventions:

- One of the good conventions in IPv6 is that the leading zeros in the address field are optional and can be compressed:
    - Original address: 3021:0000:120D:0000:0000:18F0:756B:125A
    - Compressed address: 3021:0:120D:0:0:18F0:756B:125A

  That means if one or more blocks in the IPv6 address has “0000” it can be compressed into just one “0”.
- In addition, a pair of colons (::) represent successful compressed fields of “0”. However, this can allowed only once in an IPv6 address.
    - Original address: 3021:0:120D:0:0:18F0:756B:125A
    - Compressed address: 3021:0:120D::18F0:756B:125A
    - Original address: FF03:0:0:0:0:0:0:3
    - Compressed address: FF03::3

  So if IPv6 address has one or more “:0:0:” it can be compressed into two colons “::”.

2.3.3. IPv6 Address Prefix

The IPv6 prefix is part of the address that represents the left-most bits that have a fixed value and represent the network identifier [121]. “*IPv6 address prefixes are similar to the way that IPv4 addresses prefixes are written in Classless Inter-Domain Routing (CIDR) notation. An IPv6 prefix is represented by notation “IPv6-prefix/prefix-length”. The “/prefix-length (e.g. /64)” variable is a decimal value that indicates the number of high-order contiguous bits of the address comprising the prefix, which is the network portion of the address. For example, 1080:6809:8086:6502::/64 is an acceptable IPv6 prefix* [121]”.

The following examples are legal representations of the 60-bit IPv6 prefix:

- Example: 23BC:0000:0000:AB12:0000:0000:0000:0000/60
- Example: 23BC::AB12:0:0:0:0/60
- Example: 23BC:0:0:AB12::/60

2.3.4. IPv6 Address Types

There is a major difference in the IP address requirements between an IPv4 an IPv6 node. An IPv4 node typically use one IP address; but an IPv6 node requires more than one IP address. There are three major types of IPv6 addresses, which are [121-122]:

- **Unicast:** An address for a single interface. A packet that is sent to a unicast address is delivered to the interface identified by that address (Figure 2.7). The IPv6 packet contains both source and destination IP addresses. A host interface is equipped with an IP address, which is unique in that network segment [121].

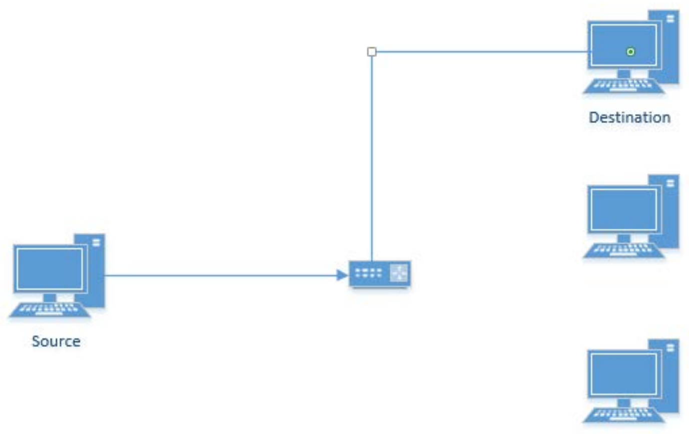


Figure 2.7: Unicast IPv6 address diagram

A unicast address is an address for a single interface. A packet that is sent to a unicast address is delivered to the interface identified by that address [122]. The following is a list of the IPv6 unicast types [121]:

- Global unicast address
- Site-local unicast address
- Link-local unicast address
- IPv4-mapped IPv6 address
- IPv4 compatible IPv6 address (nearly deprecated)

- Special IPv6 Address

- **Multicast:** An address for a set of interfaces that typically belong to different nodes. A packet sent to a multicast address is delivered to all interfaces identified by the multicast address (Figure 2.8) [121]. In order to use multicast information, an interested host needs to join that multicast group first. All the interfaces that join the group receive the multicast packet and process it, while other hosts ignore the multicast packet information [121].

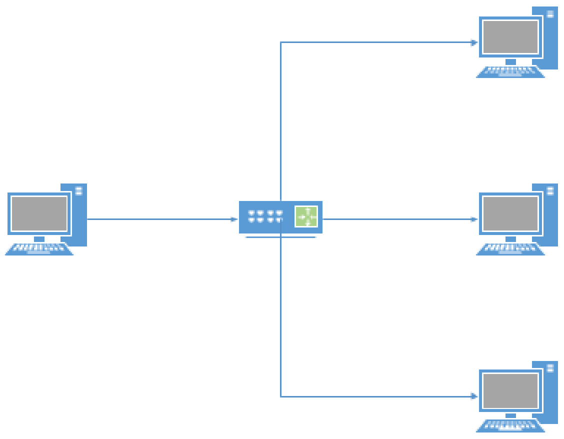


Figure 2.8: Multicast IPv6 address diagram

IPv6 multicast address is an IPv6 address that has a prefix of FF00::/8 (1111 1111). As we said before an IPv6 multicast address is an identifier for a group of interfaces. An interface may belong to any number of multicast groups on different nodes [121]. The multicast address range uses 1/256 of the total IPv6 address space (Figure 2.9) [121].

Figure 2.9: IPv6 Multicast Address format

The second octet following the initial 8bits, defines the lifetime and scope of the multicast address. The flag is a set of 4 flags, “0|R|P|T”. The higher order flag is reserved, must be initialized to “0”, and if R bit is set to “1”, then the P and T bits must set to “1”. The permanent multicast address has a lifetime parameter equal to 0 and assigned by the Internet

Assigned Numbers Authority (IANA), a temporary multicast address has a lifetime parameter equal to 1. Scope is a 4-bit multicast scope value and used to limit the scope of the multicast group. Scope parameters are [96][121]:

- 0 = reserved
- 1 = Interface-Local scope
- 2 = Link-Local scope
- 3 = reserved
- 4 = Admin-Local scope
- 5 = Site-Local scope
- 6 = (unassigned)
- 7 = (unassigned)
- 8 = Organization-Local scope
- 9 = (unassigned)
- A = (unassigned)
- B = (unassigned)
- C = (unassigned)
- D = (unassigned)
- E = Global scope
- F = reserved

The remaining 112bits is “group ID” that identifies the multicast group, either permanent or transient, within the given scope. An example of an IPv6 multicast address would be all of the NTP servers on the Internet IANA allocated addresses – FF0E:0:0:0:0:0:0:101 [96]

Within the reserved multicast address range of FF00:: to FF0F::, the following addresses are assigned to identify specific functions [96]:

- FF01::1—All Nodes within the node-local scope (that is, only for that host)
- FF02::1—All Nodes on the local link (link-local scope).
- FF01::2—All Routers within the node-local scope
- FF02::2—All Routers on the link-local scope
- FF05::2—All Routers in the site (site-local scope)
- FF02::1:FFXX:XXXX—Solicited-Node multicast address, where XX:XXXX represents the last 24 bits of the IPv6 address of node.

- **Anycast:** An address for a set of interfaces that typically belongs to different nodes. A packet sent to an anycast address is delivered to the closest interface as defined by the routing protocols in use—identified by the anycast address (Figure 2.10) [121].

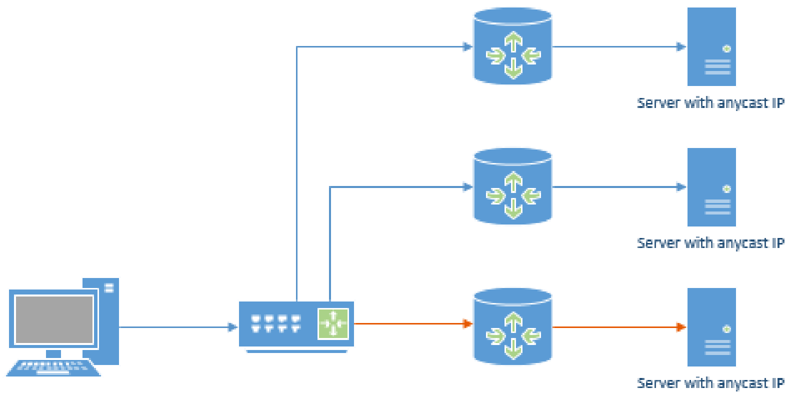


Figure 2.10: anycast IPv6 address diagram

An IPv6 address is assigned to a single interface, not a node. However, a single interface could be assigned multiple IPv6 addresses. Hence, it is easy to identify a node by any of its unicast addresses. The following are notable exceptions to these general rules [96] [121]:

- Multiple interfaces can have a single unicast address assigned to them when they are used for load sharing over multiple physical interfaces. The same is true when multiple physical interfaces are treated as a single interface at the Internet layer [96] [121].
- Routers using unnumbered interfaces on point-to-point links are not assigned IPv6 addresses, because the interfaces do not function as a source or destination for IP datagrams [96] [121].

2.3.5. Global unicast address

The IPv6 “*global unicast address is the equivalent of the IPv4 global unicast address* [121]”. A global unicast address is an IPv6 address from the global unicast prefix. The structure of global unicast addresses enables aggregation of routing prefixes that limits the number of routing table entries in the global routing table. Global unicast addresses used on links are aggregated upward through organizations and eventually to the Internet service providers (ISPs). Global unicast addresses are defined by a global routing prefix, a subnet ID, and an interface ID [121]. Except for addresses that start with binary 000, all global unicast addresses have a 64-bit interface ID. The current global unicast address allocation uses the range of addresses that start with binary value 001 (2000::/3), as shown in Figure 2.11. The Internet Engineering Task Force (IETF) has assigned binary prefix 001 (hex prefix 2000::/3) to Internet Assigned Numbers Authority (IANA) for use on the Internet. 2000::/3 is the global unicast address range and uses one-eighth of the total IPv6 address space [121]. It is the largest block of assigned block addresses. Also this means that all valid global unicast addresses begin with 2000::/3 prefix [121].

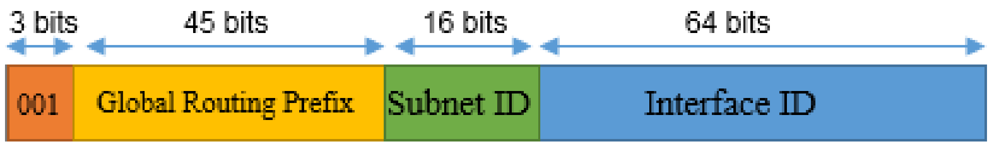


Figure 2.11: Global unicasst address format

- **Global Routing Prefix:** The prefix is assigned to a site. It will passes from IANA to the Regional Internet Registry (RIR) and then to an ISP (Internet Service Provider) or LIR (Local Internet Registry) and then to a customer or a specific customer location. Authority (IANA) is allocating the IPv6 address space in the ranges of 2001::/16 to the registries [121].
- **Subnet ID:** A 16-bit subnet field called the Subnet ID could be used by individual organizations to create their own local addressing hierarchy and to identify subnets. This field allows an organization to use up to 65,535 individual subnets [121].
- **Interface ID:** The 64-bit interface identifier in an IPv6 address is used to identify a unique interface on a link. A link is a network medium over which network nodes communicate using the link layer. The interface identifier may also be unique over a broader scope. In many cases, an interface identifier will be the same as or based on the link-layer (MAC) address of an interface. As in IPv4, a subnet prefix in IPv6 is associated with one link [121].

2.3.6. Site-local unicast address

Site-local unicast addresses are similar to the private addresses such as 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16 used in IPv4 networks. Site-Local addresses were originally designed to be used for addressing the inside of a site without the need for a global prefix. Private addresses can be used to restrict communication to a specific domain, or to assign addresses in a site that is not connected to the global Internet, without requiring a globally unique prefix [121]. IPv6 routers must not advertise routes or forward packets that have site-local source or destination addresses outside the site boundary. If the site requires global connectivity in the future, a global unicast prefix must be assigned to that site. The site-local addressing plan initially defined for site-local addressing can be directly applied using the global unicast prefix. A site-local unicast address shown in Figure 2.12 is an IPv6 unicast address that uses the prefix range FEC0::/10 (1111 1110 11) and concatenates the subnet identifier (the16-bit Subnet ID field) with the interface ID in the EUI-64 format. The site-local unicast address range uses 1/1024 of the total address space [121].

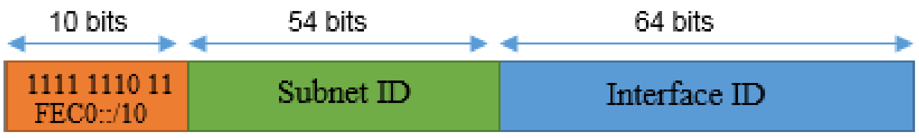


Figure 2.12: Site-local unicast address format

2.3.7. Link-Local IPv6 unicast address

A link-local unicast address is an IPv6 unicast address that is automatically configured on an IPv6 node interface by using the link-local prefix FE80::/10 (1111 1110 10) and the interface ID in the EUI-64 format. Link-Local addresses are designed to be used for addressing on a single link for purposes such as automatic address configuration, neighbour discovery, or when no routers are present. Link-local addresses are typically used to connect devices on the same local link network without the need for global addresses. Hence, link-local addresses are useful only in the context of the local link network. Nodes on a local link can use link-local addresses to communicate with each other without the need for a router. IPv6 nodes do not need site-local or globally unique addresses to communicate. IPv6 routers must not forward to other links packets that have link-local source or destination addresses. FE80::/10 is the link-local unicast address range and uses 1/1024 of the IPv6 address space [121, 122]. Figure 2.13 shows the structure of a link-local address.

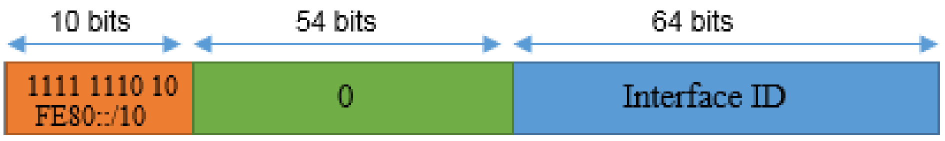


Figure 2.13: Link-local unicast address format

2.3.8. IPv4-mapped IPv6 address

The IPv4-mapped IPv6 address is another type of IPv6 unicast address that embeds an IPv4 address in the low-order 32 bits, zeros in the high-order 80 bits, and ones in bits 81 through 96 of the IPv6 address. This address type is used to represent the address of an IPv4 node as an IPv6 address. On a dual stack node, an IPv6 application sending traffic to a destination represented by an IPv4-mapped IPv6 address will send IPv4 packets to that destination. Figure 2.14 shows the structure of an IPv4-mapped IPv6 address [121].

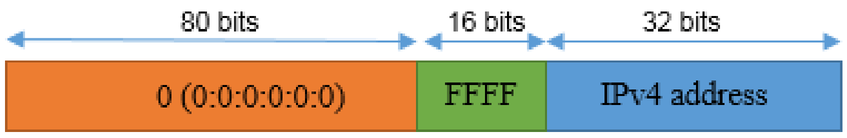


Figure 2.14: IPv4-mapped IPv6 address format

2.3.9. IPv4 compatible IPv6 address

The IPv4-compatible IPv6 address is used in IPv6 transition mechanisms to tunnel IPv6 packets dynamically over IPv4 infrastructures. The IPv4-compatible IPv6 address is a type of IPv6 unicast address that embeds an IPv4 address in the low-order 32 bits and zeros in the high-order 96 bits of the IPv6 address. The format of an IPv4-compatible IPv6 address is 0:0:0:0:0:0:A.B.C.D or ::A.B.C.D. The entire 128-bit IPv4- compatible IPv6 address is used as the IPv6 address of a node and the IPv4 address embedded in the low-order 32-bits is used as the IPv4 address of the node [121-122]. The IPv4 address used in the "IPv4-Compatible IPv6 address" must be a globally unique IPv4 unicast address. Figure 2.15 shows the structure of an IPv4-compatible IPv6 address and a few acceptable representations for the address [121-122].

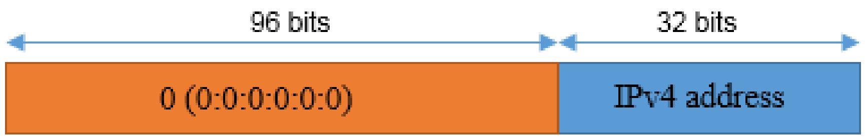


Figure 2.15: IPv4 compatible IPv6 address format

Example: 0:0:0:0:0:0:192.168.2.1 = ::192.168.2.1 = ::C0A8:0021

2.3.10. Special IPv6 address

Referring to previous sections, we can see that there are several special addresses and address groups within IPv6. Some of these will be familiar to work with IPv4 addressing, and some are new in IPv6:

- Unspecified address (::/128): “*This all-zeros address refers to the host itself when the host does not know its own address. The unspecified address is typically used in the source field by a device seeking to have its IPv6 address assigned*” [122].
- Loopback address (::1/128): “*IPv6 has a single address for the loopback function, instead of a whole block as is the case in IPv4*”[122].
- IPv4-Mapped addresses (::FFFF/96): “*A /96 prefix leaves 32 bits, exactly enough to hold an embedded IPv4 address. IPv4-Mapped IPv6 addresses are used to represent an IPv4 node’s address as an IPv6 address. This address type was defined to help with the transition from IPv4 to IPv6*” [122].
- Link-Local unicast addresses (FE80::/10): “*As the name implies, Link-Local addresses are unicast addresses to be used on a single link. Packets with a Link-Local source or destination address will not be forwarded to other links. These addresses are used for neighbour discovery, automatic address configuration and in circumstances when no routers are present*” [122].
- Unique local unicast addresses (FC00::/7): “*Commonly known as ULA, this group of addresses are for local use, within a site or group of sites[122]. Although globally unique, these addresses are not routable on the global Internet. This author looks at ULA as a kind of upgraded rfc1918 (private) address space for IPv6*” [122].

IPv6 addresses are either configured manually, state fully (such as by Dynamic Host Configuration Protocol (DHCPv6) [23]), or by the newly introduced Stateless Auto configuration (SLAAC) [24], providing plug-and-play connectivity. With SLAAC, the host first creates a link-local address on its own. After receiving a router advertisement, the node generates global addresses with the announced network prefixes. Recommended network prefix sizes for end sites are between /48 and /64 [24,26 , 123].

Due to the increasing number of mobile nodes, mobility support [28] has gained importance. It allows nodes to remain transparently reachable via the same address while wandering through the network. In case the mobile node is in a foreign network, it provides its actual address to its router by means of a binding update. This provides two possibilities for correspondent nodes to communicate with the mobile node: The communication can be passed onto the home agent, which tunnels the traffic on to the mobile node. Alternatively, route optimization allows direct communication without the home agent by using a certain routing header [123].

The transformation from version 4 to 6 takes time and is accompanied by a phase of coexistence. Some nodes are capable of both protocols, while others are limited to one or the other. Therefore, transition technologies that bridge this gap have been developed, which can be divided into two main types: (1) Tunnelling delivers a packet as another packet's payload. [29, 123]. Provides a general description on tunnelling IPv6 over IPv4, while [30] is a specification for tunnelling other protocols over IPv6. Currently, there are a high number of different technologies tunnelling IPv6 over IPv4:

- **6to4:** Referring to RFC3056 [32], is an IPv6 transition mechanism which enables encapsulation of IPv6 packets into IPv4 for transport across an IPv4 network. It allows automatic "IPv6 to IPv4" translation and dealing with packets underlying IPv4 network [31, 32].
- **IPv6 rapid deployment:** is a stateless tunnelling mechanism, which allows a Service Provider to rapidly deploy IPv6 in a lightweight and secure manner without requiring upgrades to existing IPv4 access network infrastructure [33, 34],
- **6over4**: is a tunnel, which is used to encapsulate an IPv6 packet in an IPv4 native network that allows creation of a peer-to-peer network between two machines that are communicating [35], [36]. And also a set of policies to allow isolated IPv6 hosts located on physical links with no directly connected IPv6 router to become fully functional IPv6 hosts by using an IPv4 domain that supports IPv4 multicast as their virtual local link [34].
- **ISATAP:** stands for Intra-Site Automatic Tunnel Addressing Protocol that is another transition for IPv6 packets into the IPv4 network that allows the use of IPv6 applications on the IPv4 network. ISATAP will encapsulate the IPv6 packets with IPv4 headers before sending them out to the network [37].
- **Teredo**: is an address assignment and host-to-host automatic tunnelling for unicast traffic when IPv6-IPv4 hosts are located behind IPv4 NATs [38 - 39].

"Alternatively, protocol translation, i.e., the translation of IPv4 into IPv6 headers and vice versa, can be used. Due to being tightly connected, IP translation also includes ICMP translation. The first specification Network Address Translation - Protocol Translation (NAT-PT) has been criticized by [40], [41] *for numerous reasons, e.g. lacking 1The stateless DHCP approach is technically speaking not a means of address assignment because it does not maintain a client state* [47]. *Support of DNSSEC. Its successor is standardized in* [42 - 46]. *However, tunnelling is currently preferred"* [123].

2.3.11. Internet Control Message Protocol Version 6 (ICMPv6)

Unlike ICMP for IPv4, ICMP for IPv6 [21] plays an important role in IPv6 network. ICMPv4 is not required in IPv4 but ICMPv6 is a required element and therefore it cannot be filtered completely. ICMPv6 has a next header value of 58. The main reason that ICMP was developed as a protocol was to be used for tests and diagnosis on IPv4 networks. The most important features that ICMP provides is to enable the utilities such as ping and trace route to help verify end-to-end IP communication and connectivity and provide information about any errors on the connection back to nodes [77].

ICMPv6 messages can be categorized into two categories [21]:

- Error messages
  - 1 Destination Unreachable
  - 2 Packet Too Big
  - 3 Time Exceeded
  - 4 Parameter Problem
  - 100 Private experimentation
  - 101 Private experimentation
  - 127 Reserved for expansion of ICMPv6 error messages
- Informational messages
  - 128 Echo Request
  - 129 Echo Reply

Error messages will generate a report of any errors that occur during the message delivery. Informational messages will allow sharing of required information between nodes. As in other features, attackers may use ICMP for exploitation and therefore sys-admin has no choice but to completely filter the protocol to prevent such attacks (DoS/DDoS, Evasion, Scan, Man in the Middle) [108]. However, unlike ICMPv4, ICMPv6 cannot filter/block completely due to the important rule that it plays in the IPv6 network. According to RFC 4890 filtering ICMPv6 on routers and firewalls are different from on a host.

ICMPv6 is a required protocol on every IPv6 network. ICMPv6 provides the following functions [21]:

- Neighbour Discovery Protocol (NDP), Neighbour Advertisements (NA), and Neighbour Solicitations (NS) provide the IPv6 equivalent of IPv4 Address Resolution Protocol (ARP) functionality.
- Router Advertisements (RA) and Router Solicitations (RS) help nodes determine information about their LAN, such as the network prefix, the default gateway, and other information that can help them communicate.
- Echo Request and Echo Reply supports the Ping6 utility.
- PMTUD determines the proper MTU size for communications.
- Multicast Listener Discovery (MLD) provides IGMP-like functionality for communicating IP multicast join and leave.
- Multicast Router Discovery (MRD) discovers multicast routers.
- Node Information Query (NIQ) shares information about nodes between nodes.
- Secure Neighbour Discovery (SEND) helps secure communications between neighbours.
- Mobile IPv6 is used for mobile communications.

2.3.12. Neighbour Discovery Protocol (NDP)

As defined in RFC2461, Neighbour Discovery is a protocol for IPv6. Since Address Resolution Protocol (ARP) has been removed in IPv6, both hosts and routers use Neighbour Discovery messages to determine the link-layer addresses of nodes on the local link. When a host is connected to an IPv6 network, it sends Router Solicitation messages to routers on the same link to get network information such as network prefix, default router and other network parameters). Stateless Auto-Configuration is another feature based on Neighbour Discovery Protocol which allows new hosts on the local link to get and configure their IPv6 address [24].

Table 2.2: Neighbour Discovery Protocol (NDP) messages

ICMPv6 Type	**Message name**	**Source address**	**Receiver address**	**Used for**
133	Router Solicitation (RS)	Nodes (routers link-local address)	FF02::2 (multicast address)	Sent by hosts to locate routers on attached link
134	Router Advertisement (RA)	Routers	Sender of RS or FF02::1	Routers advertise their presence and link prefixes, MTU, and hop limits.
135	Neighbour Solicitation (NS)	Nodes	Solicited node multicast address or the target node's address	To query for other nodes link-layer address and also using for duplicate address detection and to verify neighbour reachability.
136	Neighbour Advertisement (NA)	Nodes	In response to NS sender or to FF02::1	In response to NS query
137	Redirect	Routers	Link local address	To inform other nodes for better next hop routers

2.4. Vulnerabilities in IPv4 and IPv6

Despite the security improvements in IPv6, some vulnerabilities are still common between IPv4 and IPv6. Insertion, Evasion and Denial of Service are three different categories of attacks, which were proposed by *Ptacek et al.* [54] for the first time. The aim of these attacks is to make IDS or the victim host process different data or process the same data but differently. By using insertion attack, IDS will accept a packet(s) that is rejected by the host. The packet looks valid only to the IDS. The attacker can bypass the signature based IDS by inserting the traffic in such way that the signature is never matched or found. This process is different in Evasion attack; in evasion attack the IDS rejects the packet that the end host accepts. The attacker can send some or all malicious traffic into the network without being caught by the IDS.

The packet insertion attack happens when IDS accepts a packet(s) which is rejected by the end system. In this case, the IDS makes an inaccurate decision by believing that the end system has accepted and processed the packet(s) when it actually has not. An attacker, by manipulating the sending packets properly, can evade signature analysis and pass undetected through the IDS. Furthermore, an IDS evasion happen when a victim accepts a packet that an IDS has rejected. As is also explained in [54], *"an IDS that mistakenly rejects such a packet misses its content entirely, resulting in it slipping through the IDS. Evasion attacks disrupt stream reassembly by causing the IDS to miss part of it. Such attacks are exploited even more easily than insertion attacks."*

2.4.1. Evasion Layer

An evasion can occur on three levels, which are Hardware (Layer 1-2), Operating System (Layer 3-4) and application (Layer 5-7). Hardware layer evasion is out of the scope of this thesis.

In 1998 Ptacek and Newsham demonstrated the various types of techniques to evade an IDS such as overlapping fragments, wrapping sequence numbers, and packet insertion. This was possible because IDS might not process and interpret the packets in the same way the protected host behind it would [54]. Most evasion techniques at layer 3-7 are categorised as:

- Insertion: Bypass detection with invalid packets (IDS accepts the packet that the end host has rejected)
- Evasion: Bypass detection with valid packets (IDS rejects the packet that the end host has accepted)
- Denial of Service (DoS): Send too many packet that use all the end host's resources
- Pattern-matching weaknesses: bypass the signature detection by using methods like fragmentation
- Encryption and tunnelling: hiding the attack into encrypted format and use protocol such as SSL, SSH and etc. to hide from detection system
- Fragmentation: breaking the attack into multiple packets and evade detection
- Protocol violation: Targeting complex protocols such as SMB, SNMP etc.

2.4.2. Most Common Attacks in IPv4 and IPv6

This section covers the most common vulnerabilities that can be exploited and techniques that can be used by an attacker.

2.4.2.1. TTL Evasion

This technique requires the attacker to have a prior knowledge of the topology of the victim's network. To evade an IDS by TTL (Time to Live) Evasion technique, attackers should have prior knowledge of the victim’s network. Traceroute is one of the ways that can be used by an attacker to obtain this information such as the number of routers between the attacker and victim. Figure 2.16 shows an attack based on TTL evasion.

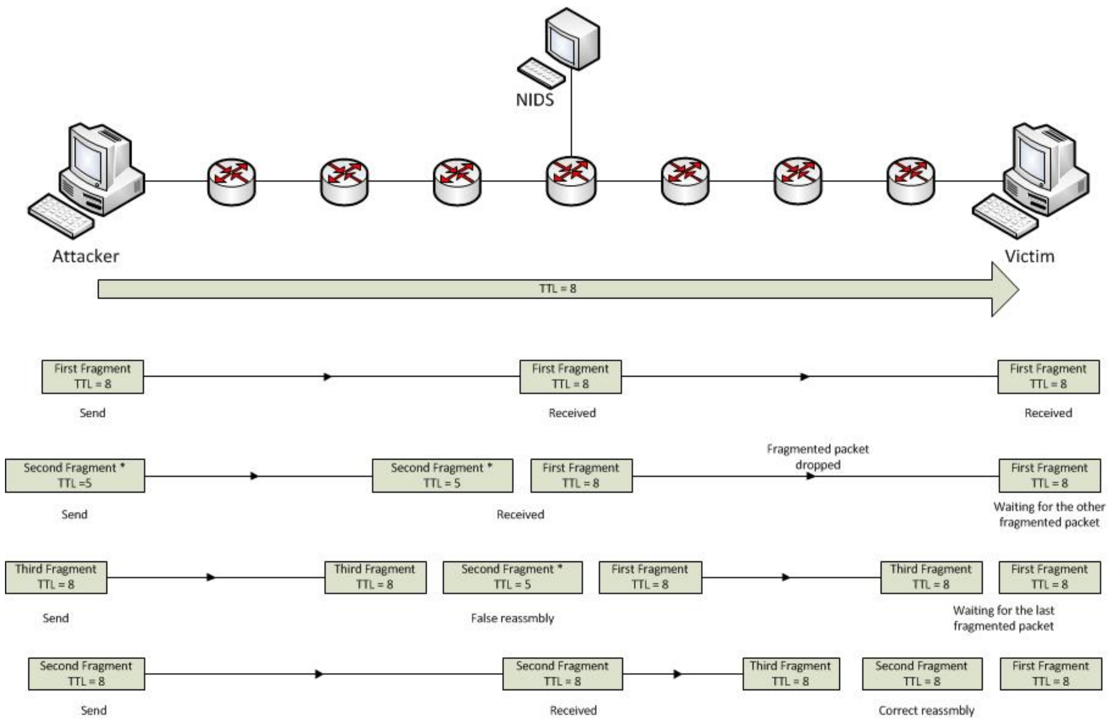


Figure 2.16: TTL Evasion

The IDS is connected to the same router which is connecting the victim to the internet to monitor the traffic. We assumed that attackers already have the information about the victim’s network. The attack begins and the attacker breaks the packet into three fragmented packets. The first fragmented packet is sent with TTL=8 where it can reach IDS and the victim host. Then the attacker sends the second fragmented packet with a TTL=5 and false payload. This fragmented packet is received by the IDS and will dropped by the next router as it has a TTL value of 0.

Afterwards the attacker sends the third fragmented packet with a TTL=8 and valid payload date. This will make a TCP reassembly by IDS while the victim hosts are still waiting for the second fragmented packet. Because of the false payload data in the second fragmented packet, IDS cannot do the reassembly correctly and as a result of that, IDS will discard all the packets and reassembly stream. Finally, the attacker sends the second fragmented packet again, but this time with a valid payload data with a TTL=8. The victim host will do the TCP-reassembly and gets the attack while IDS has only held the second fragmented packet and it has already carried out the reassembly and the stream has been discarded.

2.4.2.2. IP Fragmentation

In IPv6, only the destination node will do the packet reassembly. IPv6 needs every link to have a Maximum Transfer Unit of 1280 or greater otherwise as Request for Comment (RFC) 2460 said, the packet should be discarded. By misuse of fragmentation techniques, an attacker can bypass detection and perform attacks [14]. In a Fragmentation attack, the attacker handles fragments in exactly the same way as an IDS does. The fragmentation attacks can be summarised as follows [6] [138]:

- IDS flooding
- Overlapping fragments
- DoS and DDoS attack [56]
- Tiny fragment attack [14]
- Teardrop attack
- Overlapping fragment attack

2.4.2.3. DoS and DDoS attack

A Denial of Service (DoS) is an attempt to make a machine or network resource unavailable to its intended users by sending too many fragmented packets [56]. In this type of attack the attacker overloads the victim(s) by flooding the target machine’s resources i.e. bandwidth, RAM or CPU with too many requests (Figure 2.17). In a DDoS attack, the scenario is same, but the way in which the attacker launches their attack is different. In this attack the traffic is generated from different sources (slave) at the command of the attacker (master) and sent to the victim (Figure 2.18). There are many types of DDoS attack, but the common attacks are Traffic attack, Bandwidth attack and Application attack. In Traffic attack, the attacker floods traffic by sending a huge number of TCP, UDP and ICMP packets to the victim. In Bandwidth attack, the attacker sends a massive amount of data to the victim, which results in the loss of network bandwidth. In Application layer attack, the attacker sends data messages to the application layer, which cause service outage.

Figure 2.17:DoS attack

Figure 2.18:DDoS attack

2.4.2.4. TearDrop

Teardrop attacks exploit the reassembly of fragmented IP packets. The size of the first payload packet in N and fragment bit is on and the 2nd fragment bit is off and offset + payload is < 0. This would cause a crash. This is because the machine assumes that the packet become longer and longer.

2.4.2.5. Tiny fragment attack

If an attacker made a fragment size small enough to force some of a TCP packets and TCP header fields into the second fragment, filter rules that specify patterns for those fields will not match [124]. A disallowed packet might be passed by the IP filtering and this is because of a mismatch between a filtering implementation and the pattern of the fragmented packet size.

An example of this attack is when the first fragment contains only eight octets (referring to RFC 791all internet modules must be able to forward a datagram of 68 octets without any further fragmentations). Eight octets are enough to contain the source and destination port number, and the TCP flags will be in the second fragment. The security device which is monitoring the data attempts to drop any connection requests which have the TCP datagrams SYN=1 and ACK=0. In this instant, the device is unable to check and test these flags as they are not contained in the first fragmented packet and will typically ignore them in subsequent (Figure 2.19).

Figure 2.19: Tiny fragment attack packet

2.4.2.6. Overlapping fragment attack

RFC 791 described a reassembly algorithm that results in new fragments will overwrite any overlapped portions of previously received fragments in the current IP protocol specification.

By using such reassembly implementation, an attacker can construct a series of packets which contain innocuous data in the lowest fragment (the fragmented packet which has zero offset). Furthermore, in some subsequent, a packet with non-zero offset will overlap the TCP header information such as destination port and that will cause an issue because the second packet would pass the filtering without being detected, as it doesn’t have a zero fragment offset (Figure 2.20).

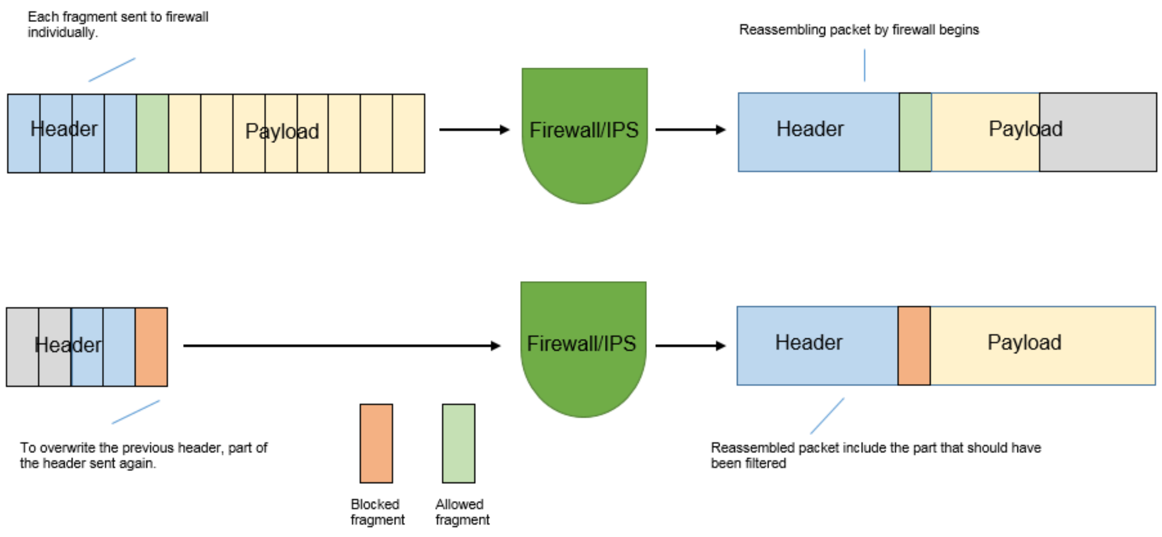

Each fragment sent to firewall individually.
Header
Payload
Firewall/IPS
Reassembling packet by firewall begins
Header
Payload
Header
Firewall/IPS
Header
Payload
To overwrite the previous header, part of the header sent again.
Blocked fragment
Allowed fragment
Reassembled packet include the part that should have been filtered

Figure 2.20: Overlapping fragment attack

TCP Overlapping is processes of segments with partial sequence numbers which overlap and are different on each Operating System. Some of them prefer to first receive the segment and the other the last. An attacker can use this as an advantage and launch insertion or evasion attacks by using segment overlaps.

2.4.2.7. TCP SYN Flooding Attack

Cheswick et al [97] discovered TCP SYN flooding vulnerability into their books “Firewalls and Internet Security: Repelling the Wily Hacker”. The first attack was publicised in 1996 with a description and exploit tool in Phrack Magazine Volume Seven, Issue Forty-Eight, File 1 of 18 [98].

The SYN flooding attack is a denial-of-service [57] method affecting hosts that run TCP server processes and the attack relies on the victim host TCP implementation behaviour. The attacker will take an advantage of the state retention TCP performs for some time after receiving a SYN segment to a port that has been put into the LISTEN state. Because of the open half-connections and the resources which are used to keep those connections alive in the host, new and legitimated connections will be dropped by the host.

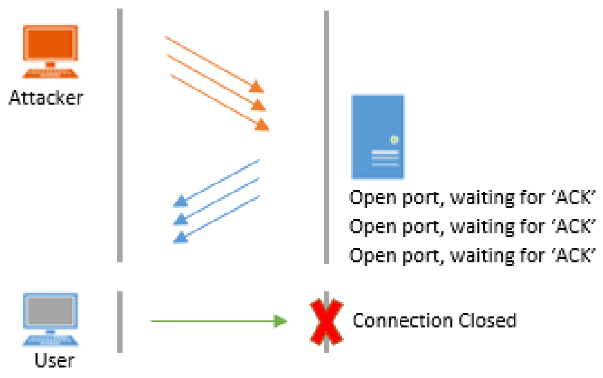

Attacker
Open port, waiting for 'ACK'
Open port, waiting for 'ACK'
Open port, waiting for 'ACK'
User
Connection Closed

Figure 2.21: TCP SYN attack

2.4.2.8. Encrypted traffic attack

According to a study by two prominent security institutes (A10 Network and Ponemon Institute) this thread exists because of using encryption at Secure Socket Layer (SSL) which hides everything, including malware from most security tools. This means that this technique as well as protecting people's sensitive information, can allow malware or attack hiding inside the encrypted traffic that passes through security hardware like IDS/IPS without being detected. An example of this technique can be the use of SSH. SSH is designed to replace the insecure remote/local login procedures such as, rLogin, rsh, rcp and telnet. Since then it became a tool for securing Internet traffic and version 2 of SSH was standardised by RFCs [107-109]. The SSH features can help attackers to run various types of attacks such as IP spoofing, IP source routing, DNS spoofing, interception of password and sensitive data, manipulation of data and spoofed connection [102].

2.4.2.9. Web-based malware

Web-based malware code is written by attackers and inserted into JavaScript. Other types of attacks that can run from the web are Unicode and Hex encoding (used to represent characters in URLs). This type of encoding has mostly been used in the security community. The famous IIS exploits "Fingerprinting Port 80 Attacks" (Cgisecurity.com, 2001) used this type of encoding. In addition, attackers could use a web browser's URL to enter a path statement in order to access a file on the web server with the intention of causing damage, or to retrieve sensitive information. The attacker could alter the URL's path statement to appear differently to evade detection and cause harm.

2.4.2.10. Maximum Transmission Unit attack

MTU (Maximum Transmission Unit) is a value, which specifies a maximum size that is allowed for datagrams on its network segment through the interface. (E.g. WLAN has an MTU of 7981bytes and Ethernet has between 1501-9198 bytes.) To find out the lowest MTU between attacker and victim, attackers can use a technique called “path MTU discovery”. In this case, if the attacker finds the lowest MTU, they can send a packet with a bigger size than the MTU and set the “Don’t Fragment” bit. Such a packet will dropped by the router at the beginning of the low-MTU line, but it will reach the IDS and it causes the insertion attack.

2.4.2.11. Source routing attack

Source routing is way that allows the sender of a packet(s) to specify the route which the packet should travel to the network and the end host. Most of the IDSs are set that the internal network has more trustworthiness than the internal network. By using source routing technique, an attacker can spoof the sender (victim host) and set, as it is one of the addresses from the victim host network. By using this technique, an attacker can perform an insertion attack.

2.4.2.12. TCP Session Teardown attack

TCP Session Teardown, in this technique, an IDS must be able to know and detect if the session between two hosts has been terminated, otherwise an attacker can convince the IDS that the session was closed and can send packets unmonitored. By sending a FIN (Final) segment and received the ACK (Acknowledges) segment corresponding by both sides, a session will be closed. An attacker can perform an evasion attack by hiding FIN/ACK segments from the IDS and make it out of sync.

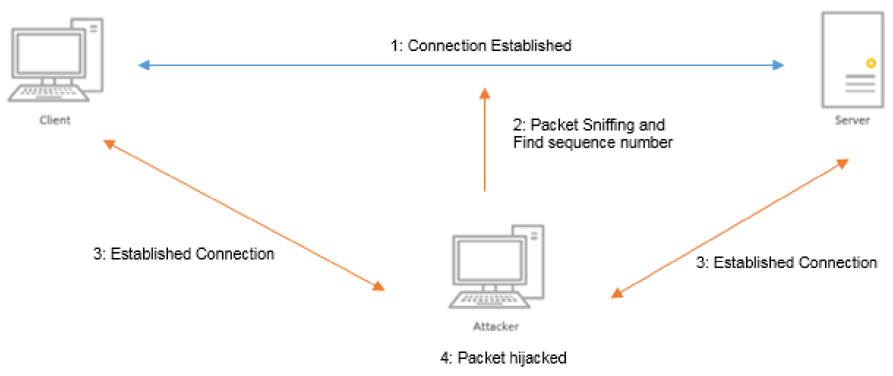


Figure 2.22: TCP Session attack

2.4.2.13. Operating System Fingerprint

This method is used for identification of the victim host. In human life, fingerprints can be used to identify a person. Similarly an OS has its unique implementation of communication protocols by which it can be identified. In order to identify the OS and its version remotely and without having a direct access to that system, the attacker uses fingerprinting to analyse certain characteristic and network behaviour communication [134]. By using such a method, the attacker can easily discover the live host on the network, identify their OS and furthermore by using this method the attacker could even reveal the victim host's missing security patches or service packs. As a result, the attacker can easily use the related vulnerability to gain access to and control the end host easily [135].

2.4.2.14. ICMPv6 Flooding Attack

ICMPv6 flooding attack is one of the most common attacks in both IP versions. The aim of using ICMPv6 attack is to use all of a victim's resources (bandwidth, CPU and RAM) by sending a large amount of traffic. The packet can contain any ICMPv6 type with source address referring to another node on the network [107].

To disturb the communications between routers and hosts, an attacker can use ICMPv6 error or informational messages such as ECHO (request and reply), router advertisement, neighbour advertisement, neighbour solicitation (NS) and multicast listener discovery messages for a successful attack [116].

2.4.2.15. ICMPv6 Amplification

The Amplification attack is considered as one of the common security challenges in IPv4 and still exists in IPv6. The amplification attack allows the attackers to generate huge numbers of packets using a small number of packets and amplify it to a large number of packets based on the multicast address feature.

Broadcast Amplification attack also known as Smurf (Figure 2.23), is the most well-known amplification attack, which is based on ICMPv6 multicast address function. The attacker uses Smurf attack to launch a DoS attack by sending an ECHO request packet to a multicast address with spoofed source address of victim machine. Once all nodes of the targeted multicast address have received a packet, all nodes start to reply to the source, which is the victim and flood it with a large number of ECHO reply attacks. The victim will be overwhelmed and cannot respond to genuine requests [116]. In addition, there is another version of Smurf attack, which is call rSmurf (Remote Smurf) attack that has stronger amplification, because each packet generated by rsmurf6 can generate a large number of packets on the remote LAN. As a result, one malicious packet will generate a storm of traffic on victim network.

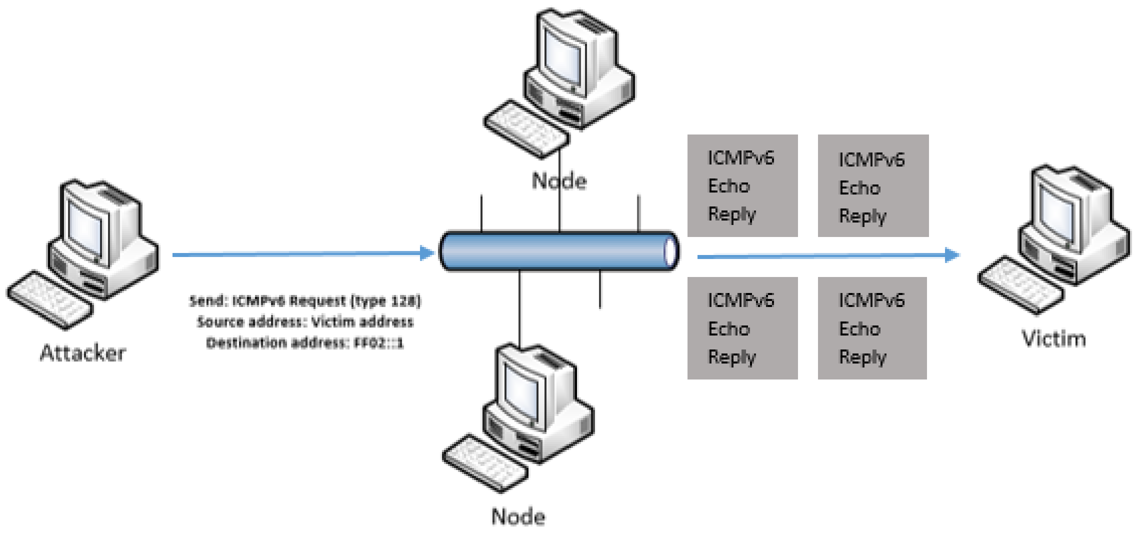


Figure 2.23: Smurf attack

2.4.2.16. ICMPv6 Protocol Exploitation

By sending a Router Advertisement (RA) packet, any node on a network can claim that they are a router. An attacker can use this feature of ICMPv6 to perform a Man in the Middle (MitM) attack by presenting themselves as a router. The first method an attacker can use to launch a Dos/DDoS attack is by using Router Discovery packets, which are Router Solicitation (ICMPv6 type 133) and Router Advertisement (ICMPv6 134). The second method will be using Neighbour Discovery (ICMPv6 type 135 and 136) packets. The third method will be using Redirect furcation (ICMPv6 type 137) packets.

The Router Discovery process is responsible for packets routing. On the IPv6 network, a host will find a router by sending a Router Solicitation packet to router multicast address (FF02::2). Once the Router Solicitation packet is received by the default router, in response to that packet the router will send Router Advertisement to the host. The Router Advertisement packet contains the information needed by the host such as router specification, on-link prefix and network parameter [118]. An attacker can misuse Router Solicitation and Router Advertisement packet and perform the following attacks:

- **Default router is "killed**: By default, every node has a router table to list all routers on the network. When a node does not have any record in the table, it will consider that all destinations are on link [22]. Now an attacker can send a Router Advertisement packet with router lifetime equal to zero and spoofed address. When the host receives the Router Advertisement packet, it will delete the router record because of the lifetime, then it will redirect all packets to the destination without a router address. If the traffic is going outside of network, all packets will be lost and therefore an attack has occurred [118].
- **Bogus address configuration prefix attack:** As mentioned earlier, one feature of IPv6 is that in absence of DHCP server, a node will generate their own IPv6 using Stateless Auto-Configuration with a subnet prefixes of Router Advertisement messages that are received from a default router [119]. The router sends Router Advertisement messages accordingly to all nodes to update their routing table information. By sending a Router Advertisement message with invalid subnet prefix to multicast address (FF02::1), an attacker can launch an attack. Now all nodes will generate an IPv6 address based on the invalid prefix that was received and all communication between hosts will be disrupted.
- **Parameter spoofing:** As mentioned earlier, Router Advertisement messages contain network parameter information and they are very useful to the host to send IPv6 packets later. An attacker can send a Router Advertisement message (for example with a small hob limit), which contains false network parameters that can disturb the packet transmission and host's communications.

2.4.2.17. Neighbour Discovery Attack

Neighbour Solicitation and Neighbour Advertisement are two ICMPv6 messages that Neighbour Discovery Protocol (non-routing one) uses. Two of most important job that NDP is responsible for are, neighbour unreachability and Duplicate Address Detection (DAD) [24]. An attacker can use these functions as an advantage and launch an attack.

Duplicate address detection DoS attack: Another feature on IPv6 network is Duplicate Address Detection (DAD). When a node need a new IPv6 address, it will send Neighbour

Solicitation to all nodes multicast address “FF02::1” to check whether that IP is in use or not. If the sender did not receive a reply that means the IPv6 address is free and the new node can use it. An attacker can use this as an advantage and send a spoofed Neighbour Advertisement packet claiming that the address is in use every time that node sends a request. By using such an attack, the new nodes will not get an IPv6 address therefore there is not any connectivity (Figure 2.24) [108].

Figure 2.24: DAD attack

Neighbour unreachability detection failure: Neighbour Unreachability Detection (NUD) process detects when a neighbour is unreachable. Once this has happened, the node starts to send a Neighbour Solicitation packet to lost node address and waits for a Neighbour Advertisement reply for a short period. If no Neighbour Advertisement is received, the node will delete the peer node from its Neighbour Cache Entry table. An attacker can send a malicious Neighbour Advertisement reply to a Neighbour Solicitation request to show that the node is still alive and on the network which it is not.

2.4.2.18. ICMPv6 Redirect message attack

IPv6 nodes use ICMPv6 “Redirect” message to find a better path to their destination. The router will send “Redirect” message to a node to optimise the packet routing process and the delivery path. The following attack types can utilise ICMPv6 redirect messages:

- The traffic can be forwarded to non-existent link
- The traffic can be redirected to an existing node. This will result in the node being overwhelmed.

2.5. Attack Tools

This section covers some of the most common attack tools to perform an attack on IP level.

Fragrouter is a network intrusion detection evasion toolkit developed by Dug Sing. It implements most of the attacks described in the Ptacek & Newsham in 1998 (It features a simple rule set language to delay, duplicate, drop, fragment etc.).

THC-IPV6-ATTACK-TOOLKIT is a collection of attacking tools that can be used to test the implementation of IPv6 network and test firewall and NIDS. This collection contain the following tools (Table 2.3) [109]:

Table 2.3: THC IPv6 Toolkit

Command Name	**Description**
parasite6	ICMPv6 neighbour solitication/advertisement spoofer, pus you as man-in-the-middle, same as ARP MITM (and parasite)
alive6	an effective alive scanning, which will detect all systems listening to this address
dnsdict6	paralysed DNS IPv6 dictionary brute forcer
fake_router6	announce yourself as a router on the network, with the highest priority
redir6	redirect traffic to you intelligently (man-in-the-middle) with a clever ICMPv6 redirect spoofer
toobig6	Maximum Transamination Unit decreaser with the same intelligence as redir6
detect-new-ip6	detect new IPv6 devices which join the network, you can run a script to automatically scan these systems etc.
dos-new-ip6	Detect new IPv6 devices and tell them that their chosen IP collides on the network (DOS).
trace6	very fast traceroute6 with supports ICMP6 echo request and TCP-SYN
flood_router6	flood a target with random router advertisements

flood_advertise6	flood a target with random neighbour advertisements
implementation6	performs various implementation checks on IPv6 and IDS/Firewall check
implementation6d	listen daemon for implementation6 to check behind a Firewall
fake_mld6	announce yourself in a multicast group of your choice on the net
fake_mipv6	steal a mobile IP to yours, if IPSEC is not needed for authentication
denial6	a collection of denial-of-service tests against a target

Havij is an automated SQL Injection tool that takes advantage of a vulnerable web application to find and exploit SQL Injection vulnerabilities. An attacker can perform back-end database fingerprint, DBMS login names and password hashes and much more like fetching data from a database. However, this tool is capable of accessing the underlying file system and executing the operating system shell commands [139].

Acunetix is a web vulnerability scanner designed to replicate a hacker's methodology to find vulnerabilities like SQL injection, DoS/DDOS attack. By using Acunetix you can use an extensive feature-set of both automated and manual penetration testing tools, security analysis and repair detected threats [140].

Mendax is a TCP de-synchronizer that injects overlapping segments in randomly generated order. An attacker can use Mendax to evade NIDS, Mendax is not a router, but is a stand-alone TCP client program which can be used by an attacker to perform an evasion from an input text file, performs a fixed set of evasion technique and sends restructured exploit to the victim host [141].

In Table 2.4, a summary of Evasion and Insertion attack tools is provided.

Table 2.4: Evasion and Insertion attack tools

Tools Name	**Developer(s)**	**Evasion Attack**
Fragrouter [54]	Dug Sing	Most techniques described by Ptacek & Newsham
thc-ipv6 [109]	van Hauser	Multiple attacking tools incduing DoS,DDoS, Evasion and insertion attack
Havij [139]	IT Sec team	SQL Injection and web app evasion
Acunetix [140]	Acunetix Team	Web app analyser and evasion test attack
Mendax [141]	Min G. Kang	TCP Overlapping

2.6. Detection Methodologies

There are three different types of detection techniques, which are Signature-based, Anomaly-based and Behaviour-based. Each of these methods has advantages and disadvantages [58 - 61]. These three methods are the most common that are used by of the Intrusion Detection System.

Signature-based

A Signature-based (knowledge-based) IDS references a database of previous attacks' signatures and known system vulnerabilities. Normally, during the attack each intruder leaves a footprint (signature) behind (e.g., nature of data packets, failed attempt to run an application, failed logins, file and folder access etc.). To identify and prevent the same attack in future, those signatures will be used. Based on these signatures the IDS identifies intrusion attempts. They usually examine the network traffic with predefined signatures and update the database each time. Signature based IDS has a performance issue of matching patterns in the large cloud environment. It means, the more signatures the IDS has, the longer it will take to perform the pattern matching, Furthermore, signature based IDS cannot detect new attacks, which are not present in its signature database. This model may not always be so practical for inside attacks involving abuse of privileges.

Anomaly-based

Anomaly-based method is the overcoming of the drawbacks of the signature-based IDSs. It identifies unusual activity associated with attacks as opposed to pattern matching, using known attack strings. One method used to determine the abnormal activity is data flow analysis [62]. Flows represent all the packets exchanged between the hosts with a “single” service. Certain statistical data is updated in the flow data record. This includes a number of bytes, packets, etc. The flow is examined to determine if it has the characteristics of a possible attack. When the flow ends, the statistical data is examined and recorded. Data flow alone may not be sufficient to detect an attack, so the recorded statistical data are correlated with other events to potentially discover attacks. The disadvantages of IDS Anomaly-based are that they are more prone to generating false positive alarms due to the ever changing nature of networks and application. Also, due to the aggregation and abstraction used in profiling, alerts generated may not contain detailed enough information for analysing the attacks and alerts [64-65].

Behaviour-based

Intrusion detection techniques assume that an intrusion can be detected by observing a deviation from the normal or expected behaviour of the system or users. The model of normal or valid behaviour is extracted from reference information collected by various means. The intrusion detection system later compares this model with the current activity. When a deviation is observed, an alarm is generated. In other words, anything that does not correspond to a previously learned behaviour is considered intrusive. Therefore, the intrusion detection system might be complete (i.e. All attacks should be caught), but its accuracy is a difficult issue (i.e. you get a lot of false alarms). The high false alarm rate is generally cited as the main drawback of behaviour-based techniques because the entire scope of the behaviour of an information system may not be covered during the learning phase. In addition, behaviour can change over time, introducing the need for periodic online retraining of the behaviour profile, resulting either in unavailability of the intrusion detection system or in additional false alarms [60].

Machine Learning

Another method that can be used as a methodology for intrusion detection system is, Machine Learning (ML). Machine Learning is an artificial intelligence system, which has ability learn and find patterns inside the data. The algorithms for Machine Learning can learn and make predictions. Those algorithms should learn with a set of dataset to be able to make predictions before it can be used. The most two common used algorithms for Machine Learning is supervised-learning and unsupervised-learning. Each of these algorithms have different classifiers. Supervised learning will train and work with labelled training dataset and unsupervised learning train and work with unlabelled training dataset

Supervised Learning Classifiers:

- Native Bayes
- Support Vector Machines
- Random Forest
- Decision Trees

Unsupervised Learning Classifiers:

- K-Means
- Fuzzy Clustering
- Hierarchical Clustering

By providing training classifier with some normal dataset, Machine Learning can categorise the difference between normal and abnormal traffic in a network [25]. By completion of training, the classifier will be evaluated by using some different datasets that contain abnormal traffic. *Barreno et al* [52] mentioned that Machine Learning techniques could detect novel differences in traffic.

Table 2.5 and Table 2.6 provides a comparison for detection methods covering their most important advantages and disadvantages.

Table 2.5: IDS type advantage

Signature-based	• Effective method to detect know attacks • Easy to maintain for user
Anomaly-based	• Effective to detect unusual activity
Behaviour-based	• Effective to detect unusual activity
Machine Learning	• Real time predication • Can be used for multi-variety of data

Table 2.6: IDS type disadvantage

Signature-based	• Performance issue in high speed networks • Not effective to detect 0-day and evasion attacks
Anomaly-based	• More prone to generating false positive alarm • Weak profiles accuracy due to changing nature of networks and application
Behaviour-based	• Cannot set on a large environment, because due to changes it will generate high rate of false alarm. • Performance issue in high speed network due to high speed data and comparison
Machine Learning	• Need a lot of training data to train • Training take time

2.7. Intrusion Detection System (IDS) Type

The job of an IDS is to detect and respond to hostile attacks in a reasonable amount of time. With IDS, an organization can monitor their network activities in real-time and get an insight into the threats to their information system. Without IDS, an organization could be attacked and comprised without any alert. IDSs have matured to the point where essentially two types of IDSs have evolved: Host-based (HIDS) and Network-based (NIDS). An IDS protects networks from attacks by searching through incoming packets for known attack signatures. These systems are typically passive, monitoring traffic as it flows into a network and alerting a system administrator when an attack is suspected [65].

Host-Based IDS

Host-based Intrusion Detection System (HIDS) are monitoring attacks at the OS, application or kernel level. HIDS are aimed at collecting information about activity on a particular single system, or host [66]. Host-based Intrusion detection systems relate to processing data that originates on the computers themselves, such as event and kernel logs, and they can also monitor which program accesses which resources and flag up in case of any malicious activity. The main disadvantage of host-based IDS is that they cannot monitor the network traffic [63].

Network-Based IDS

Network-based Intrusion Detection System (NIDS) are placed in key areas of network infrastructure and used to monitor and analyse network packets. NIDS’ popularity has grown. A NIDS is more cost effective than a HIDS, because HIDS must be installed on every system in an organisation, but NIDS can protect a large swathe of network infrastructure with one device. With NIDS, a company will have a wide-angle view of what's happening inside the network. NIDS will capture the network traffic from the wire as it travels to a host. This can be analysed for a particular sign or for unusual or abnormal behaviour. Several sensors are used to sniff the packets on the network, which are basically computer systems designed to monitor the network traffic [67]. Most network-based systems allow advanced users to define their own signatures. The disadvantage of network-based IDS is they are not capable of analysing the packets in heavy network traffic loads [68].

In Table 2.7, most known advantages and disadvantage of IDS type are listed.

Table 2.7: IDS type advantage and disadvantage

	Host-based	**Network-based**
Advantages	• Does not need any additional hardware • It can deal with encrypted traffic • Can detect Trojan horse attack	• Can be deployed into existing networks with low time distribution
Disadvantages	• Not able to detect multi host scanning • It can inflict the performance overhead on its host systems	• Cannot analyse encrypted network traffic

2.8. Summary

This chapter provided the necessary background information to understand the main concepts involved in this research. It outlined the structure of IPv4 and IPv6, the most important changes made to IPv6, the most common security vulnerabilities at IP level and showing how an attacker can use them to evade detection by presenting some tools that can be used to perform attack and/or evasion. Moreover, it outlined the methodology that most Intrusion Detection Systems are using to detect malicious activities and a brief description about different types of Intrusion Detection Systems. By concluding the Aims and Objectives of this research, better understanding of IPv6 packet header structure, examine most common security vulnerabilities and attacks on IPv6, A set of requirements drafted for the new framework. These information and requirements will be used throughout this research to gauge the suitability of existing solutions and as a means of evaluating the proposed framework.

Chapter 3
Related Work

3.1 Introduction

In this chapter, our focus is on analysing existing works and solutions, by outlining their merits and highlighting their limitations. Ultimately, it aims to provide evidence to emphasise the inadequacies of existing approaches and therefore justify the motivation behind this research. This chapter will have four subsections. We categorised the related work into three categories, which are, DoS/DDoS, Fragmentation, Malware and port-scan detection solutions and techniques. At the end, a table drafted to highlight the methods/techniques that each work used and the weakness points of the work.

3.2 Attack Type: DoS/DDoS

Satrya et al. [112] proposed a hybrid solution to detect DDoS flooding attacks in IPv6 network. As shown in Figure 3.25, the detection engine process will work with the collected data from the network to determine whether the activity on the network is normal, high traffic or it is under DDoS attack. The sub-processes of engine detection are as follows:

- Comparing incoming packets per client with a pre-set threshold and simulating a busy network. So if the number of packets exceeds the threshold, then it will show the network status as busy.
- Comparing the signature of DDoS attack. If any match or break in the threshold is found it would show the status as DDoS attack.

The decision engine will collect the results of the analysis and, based on the results, it will decide whether the traffic is normal or an attack.

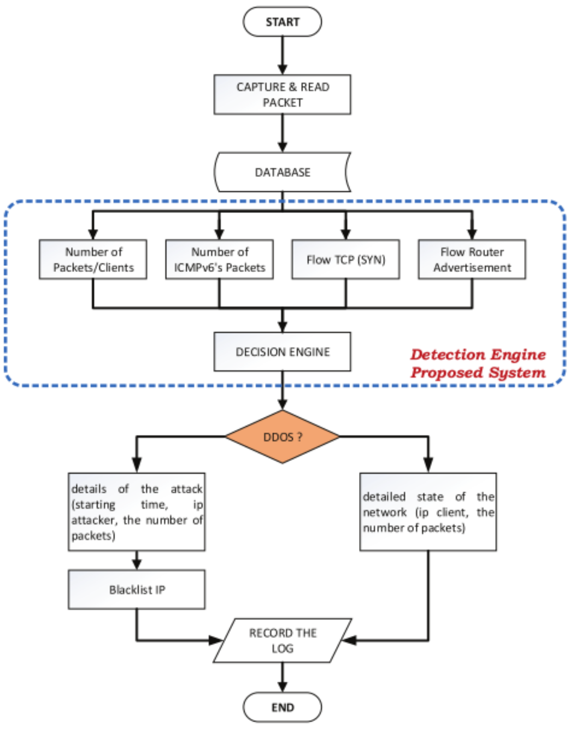

START
CAPTURE & READ PACKET
DATABASE
Number of Packets/Clients
Number of ICMPv6's Packets
Flow TCP (SYN)
Flow Router Advertisement
DECISION ENGINE
Detection Engine Proposed System
DDOS ?
details of the attack (starting time, ip attacker, the number of packets)
detailed state of the network (ip client, the number of packets)
Blacklist IP
RECORD THE LOG
END

Figure 3.25: Proposed DDOS Hybrid Detection [112]

The issue here is according to the authors, the detection accuracy is 85% and this result is based on a lab based environment scenario. However, we could not find any theoretical or technical solution to see how their system will deal with IPv6 packet.

Lee et al. [113] proposed a method to detect DDoS attack by exploiting its architecture. The architecture consists of a selection of handlers and agents, the communication and compromise, and attack. According to the author, the idea of their approach is based on the detection of each phase of the DDoS attack separately. By considering the features of the DDoS attack, they can extract several traffic variables, which give information about the occurrence of each phase of the attack. By using these variables, the authors claim that they can become aware of the DDoS attack from the initial preparation stage of the attack. The authors claim that by using such methods they can make an adaptive defence mechanism corresponding to the attack. They will observe the characteristics of the DDoS attack to select the detection parameters. Afterwards they will use a cluster to analyse the traffic. Cluster analysis is to group data so that objects in a given group are similar to each other and dissimilar from other groups. By doing this, they can separate normal traffic and each DDoS attack phase into partitioned groups.

Alnakhalny et al. [131] proposed a detection method for ICMPV6 flood attack based on Dynamic Evolving Neural Fuzzy Inference System (DENFIS). DENFIS is a system that uses online clustering to perform online and offline learning. The proposed system is based on self-machine learning. However, one important question here is if the attacker is using a mixture of methods to bypass the detection, it will take time for the machine to learn that algorithm and detect future attacks. Because of that, the attacker will change their method and therefore the detection method might not be useful for such attack.

Anbar el al. [132] An Intelligent ICMPv6 DDoS Flooding attack Detection Framework (v6IIDS) using Backpropagation Neural Network. Their aim is to detect ICMPv6 Flooding attack using an Intelligent Intrusion Detection System in an IPv6 Network (v6IIDS). The proposed system detection has four processes. These processes are Data collection and pre-processing, Traffic analysis, Anomaly-based detection and ICMPv6 flooding detection.

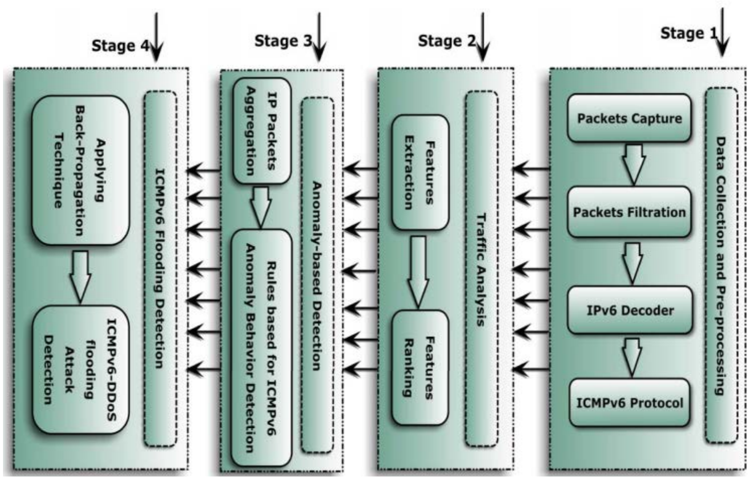


Figure 3.26: v6IIDS proposed framework architecture [132]

Song et al. [87] proposed an IP Spoofing Detection Approach (ISDA) that would keep effective parts of source IP and remove the forged part under flooding attacks and dynamically configure a flow aggregation scheme for flow-based network IDS. The proposed system was developed based on flow aggregation schemes for Flow-based Network Intrusion Detection System (FNIDS), detecting IP address spoofing level and using Fuzzy logic method. *“Flow aggregation schemes are a series of flow-aggregated methods that segregate the network traffic into a series of non-overlapping consecutive time windows and aggregate all features based on their flow”* [87]. Their system has five fixed fields of packet header, which are; Source IP, Destination IP, Source Port, Destination Port and Protocol. The main disadvantage of this proposed work is that they did not show any results based on 1Gbps network speed or more. The proposed work should only work based on the 100Mbps network.

Rafiee et al. [133] proposed a new algorithm to tackle the issue with Cryptographically Generated Addresses (CGA) [RFC3972] and Privacy Extension [RFC4941] in IPv6 state-less configuration. The proposed method uses a new way to generate Interface Identifier (IID) to reduce the computing cost and prevent security threats related to state-less configurations such as IP spoofing. However, it seems the proposed algorithm cannot detect Duplicated Address Detection attack on IPv6.

Hussain et al. [75] proposed a two-stage hybrid classification method using Support Vector Machine (SVM) as anomaly detection in the first stage, and Artificial Neural Network (ANN) as misuse detection in the second. The advantages of using SVM and ANN are better classification accuracy along with a low probability of false positive. The proposed system classifies the type of attack into four classes, Denial of Service (DOS), Remote to Local (R2L), User to Root (U2R) and Probe. The first stage is looking for any abnormal activities that could be an intrusion, while the second stage does the future analysis and if there are any known attacks it will classify them into the four categories that we already mentioned.

Figure 3.27: Two-stage hybrid classification with SVM and ANN [75]

Data Preprocess will prepare and pre-process network traffic in the data pre-process module. Once data has been received and pre-processed, it will be sent to the next process, which is “Detection and Classification”. The Detection and classification process has two stages, NIDS using SVM for anomaly and ANN for misuse detection. The data then passes to the Alarm module, which interprets events results on both stages and reports the intrusion detection activity.

Miinz et al. [11] proposed a general platform for DoS attack detection. “The system, called TOPAS - Traffic Flow and Packet Analysis System, acts as a collector for flow data exported via Cisco Netflow and/or IPFIX and provides a framework for user defined detection modules that simultaneously analyse the received data in real-time.” Several different detection modules can run in real-time according to the network administrator’s needs. A SYN flood detection module, a trace back module (allowing for the identification of the entry point of spoofed packets in the network) and a web server overloading module (focusing on DoS attacks using HTTP requests) [12].

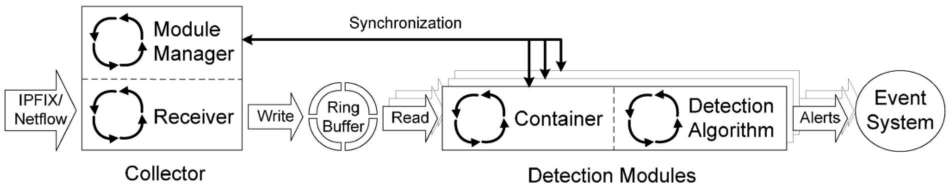


Figure 3.28: Traffic Flow and Packet Analysis System [11]

Parmar et al. [91] proposed a flow based protocol behaviour analysis to detect TCP port scan. As TCP scan is the starting point of most attacks. With TCP scan, attackers will determine the number of open ports and other information like OS type. This would help them to evade IDS/Firewall/IPS more easily. Their objectives for this system are diversity, scalability and accuracy. They construct short and long-term profiles of TCP scans with different parameters of IP flows with their statistical means and standard deviation. The flow data for evaluation is generated from a live network that is connected to the Internet. The authors compare the results with SNORT. Their proposed system detects all 13 types of scans where SNORT only detected only eight. The disadvantages of their work is that it only works with a preliminary passive network and attackers can use stealth techniques and show the traffic as legitimate network traffic and the proposed solution only works with TCP protocol and attacks on other layers cannot be detected.

Alhamaty et al. [13] tried to model the fragmentation attack and develop a new detection model for intrusion detection architecture systems to detect attacks by verifying and checking the integrity of TCP packets.

This method consists of two parts, the first part is the architecture extension in IDS, and the second part is the detection mechanism. Figure 3.29 shows the structure of the IDS proposed by the authors. IDS detectors are responsible for detecting intrusions.

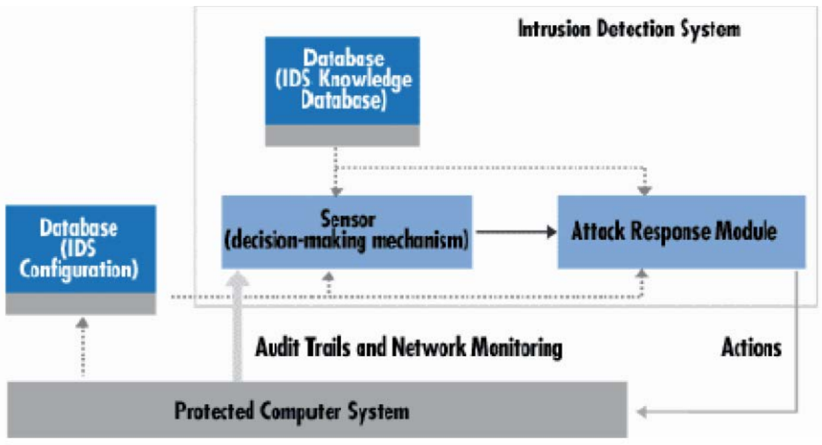

Intrusion Detection System
Database (IDS Knowledge Database)
Database (IDS Configuration)
Sensor (decision-making mechanism)
Attack Response Module
Audit Trails and Network Monitoring
Actions
Protected Computer System

Figure 3.29: IDS based on the Integrity of TCP Packet [13]

The sensor in the IDS obtains information from three sources:

- Information from existing IDS database
- Syslog file
- Tracing the traffic and controlling the network

The main point of this solution is to check TCP packets' integrity. The work focuses on the packet itself whether the packet is fragmented correctly or not. The TCP packets information processes the packets into two functions, IDS sensor and evasion detector. Whenever one of these two mechanisms detects an attack incident, it saves the signature of the packets into the IDS database and raises the alarm using "OR Gate" which is connected to the attack response model. The detection mechanism has three rules that make a logical formula, $A.L + \overline{A}.S + D.\overline{A}$ where

$A -> \ Fragment\ offset\ =\ 0.$
$L\ =\ Transport\ length\ of\ TCP\ header\ where\ in\ this\ case\ ,it$
$shouldn't\ be\ less\ than\ 4\ byte\ to\ include\ interesting\ header$
$fileds\ like\ source\ and\ destination\ port\ .$
$S\ =\ SYN\ bit.$
$D\ =\ Don't\ Fragment\ bit.$

The main rules are as the follows:

$$Rule\ (1): If\ FO = 0\ and\ Transport\ length\ (TCP)\ <\ 4byte\ [(S + D)\ port]\ then\ drop\ packet.$$

$$Rule\ (1): If\ FO =\ 1\ and\ SYN\ = 1\ then\ drop\ packet$$

$$Rule\ (1): If\ DF =\ 1\ and\ FO > 0\ then\ drop\ packet$$

The combination of rule (1) and (2) will help to detect the Tiny + Overlapping fragmentation attacks and the combination of the second section of rule (1) and the second section of rule (2) will help to detect the SYN Flood attack. The combination of the rules together in the form of A.L+¯A.S+D.¯A leads to the specification of a kind of fragmentation that is detected on TCP packet and it also mark the attacks under consideration.

The proposed solution may fail to detect a Tiny Overlapping attack when SYN=0 and ACK=1. This argument is supported by RFC 1858 [14] which talked about Security Consideration for IP fragment filtering that will pass through the detection system without being detected.

Farnaaz et al. [74] built a model for Intrusion Detection System using Random Forest classifier. Random Forest (RF) is an ensemble classifier and performs well compared to other traditional classifiers for effective classification of attacks. To find the best node for splitting, randomization is applied when constructing is in individual trees.

For the pre-processing stage, the authors used Feature Selection (FSS) that is commonly used in data mining. Feature Selection will be classified into three categories:

- **Filter method:** will only be looking at the basic of the data. The disadvantage of using this method is that it ignores feature dependencies and ignores interaction with the classifier.
- **Wrapper method:** unlike filter techniques, wrapper method embeds the model hypothesis search within the feature subnet search. The disadvantage of this method is that it has more risk of over fitting and classifier dependent selection.
- **Embedded method:** *"The search space of the feature selection algorithm is a combination of feature space and hypothesis space. The classifier itself provides the optimal feature selection."* The only disadvantage of this technique is the classifier dependent on selection.[86]

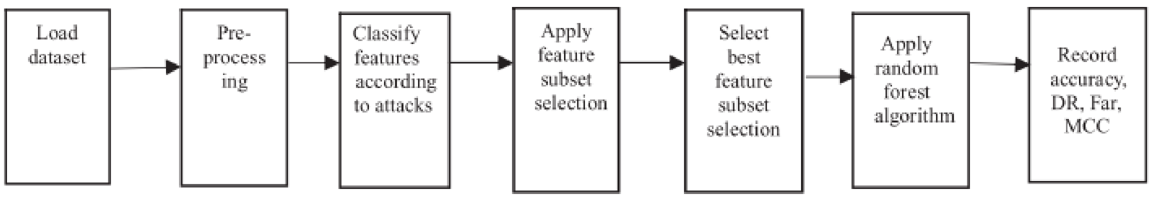

Load dataset
Pre-process ing
Classify features according to attacks
Apply feature subset selection
Select best feature subset selection
Apply random forest algorithm
Record accuracy, DR, Far, MCC

Figure 3.30: Random Forest Modelling for NIDS [74]

The proposed system should follow the above steps shown in Figure 3.30 to classify different type of attacks.

***Step 1**: Load the dataset*

***Step 2**: Apply pre-processing technique Discretization*

***Step 3**: Cluster the dataset into four datasets.*

***Step 4**: Partition the dataset into training and test*

***Step 5**: Select the best set features using feature subset selection measure Symmetrical uncertainty (SU)*

***Step 6**: Dataset is given to Random forest for training*

***Step 7**: The test dataset is then fed to random forest for classification*

***Step 8**: Calculate accuracy, Detection rate, False alarm rate, Mathew correlation coefficient* [74]

The proposed method’s disadvantage is that Random Forest will generate many noisy trees, which reduce the accuracy and detection.

Li et al. [83] solution is based on analysing information collected from the packet flows into a network. The NIDS consists of various kinds of immunity cell agents, which are dynamically generated by access via the network and each cell agent collects the information on a network. The detection works when an unauthorised intrusion (known as non-self) is detected in cooperation by exchanging information with each other cell agents. If one of the immunity agents observes network level intrusion as non-self, the agent will collaborate with other agents and then the agents will remove the process identified as non-self that constitutes the route of the intrusion and all files executed by the intrusion.

3.3 Attack Type: Fragmentation

Pinyathinun et al. [8] proposed a solution based on a NIDS (Figure 3.31). Their solution will increase the detection accuracy and reduce the noise and also reduce system workload by providing flexibility of specifying policy for individual hosts or groups. Existing problems with NIDS include a high false positive alarm, there is a lack of flexibility in identifying signature attacks, and policies can be implemented with flaws due to poor administration which makes the NIDS ineffective. Because NIDS has to deal with lots of packets, this could overload the NIDS that can cause the attacks to be missed because some packets are dropped and not analysed. Target based IDSs focus on the host target of the transmitting packet as much as on the malicious signature contained within the packet. When an alarm is generated, the log will be saved into the database. However, if there are too many false alarms, it will create a massive database. It can use a lot of resources to make a query into the database, analyse and detect. This could result in an increase in the packets' dropping rate.

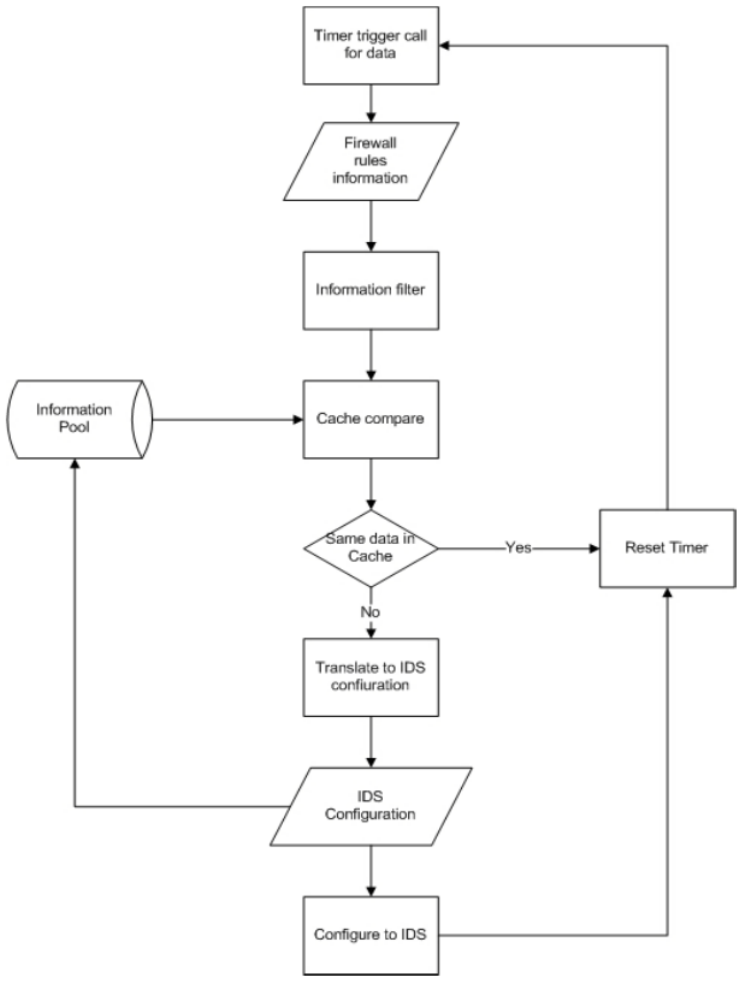


Figure 3.31: Dynamic policy data flow [8]

Weon et al. [78] propose a combined model of both signature-based and machine learning-based Intrusion Detection Systems showing that the combined system is more efficient than each individual separately. They used the DARPA Data Set in experiments in order to show the usefulness of the combined model. They used Snort for the experiment as a signature-based IDS, extended IBL (Instance-based Learner) for the principal learning algorithm for the machine learning-based IDS and C4.5 to compare the performance of their algorithms.

Chunyue et al. [88*]* proposed a pattern matching based NIDS with four modules (Figure 3.32).

- Collection: Collation module will capture the packets
- Analyse: This module has to process, pre-processing and detection engine
- Response: Handle the response and output log if any patterns match.
- Attack Rule Library: They are using “Snort Attack Rule Library” which is a disadvantage of their system because their system rules rely on a third party’s rules library.

Figure 3.32: A pattern matching based NIDS [88]

Once the incoming packets are captured, all packets will be sent to “Packet Decode” and this process starts decoding packets in a predefined format by DataPacket (Figure 3.33), formats the packets into a unifying structure in transferring the procedure, and thus simplifies for other modules

Figure 3.33: DataPacket process [88]

As previously mentioned, the authors of this paper used “Snort Rules” database for their “Attack Rule Library” process.

The Analysis module has two sub-processes, which are pre-process and Detection Engine module. Pre-process module will do further processing on the packets that are received from DataPackets. This module is a “plug-in” mechanism which includes two plugins:

- *http decoder: This plug-in is used to process Http URL string and transform it into ASCII code, which can be read by a system for detecting shady Web URL scan and vicious Unicode attack.*
- *degrag: This plug-in is used to fragmentation and TCP fragmentation*

The last module is called Response Module and deals with the intrusions behaviour such as saving attacks log, notify the system admin, reconfigure router ACP and disconnect the attacker connection. Their weakness point is their solution only works at TCP fragmentation, not IP fragmentation.

Kent et al. [16] provided Security Architecture for the Internet Protocol. In IPv6 unlike IPv4, Internet Protocol Security (IPSec) is mandatory. IPSec draws a line between protected and unprotected interfaces for host or network. If traffic wants to cross the boundary, it is subject to the access control list that is specified by the system admin who is responsible for IPSec configuration. These controls indicate whether packets cross the boundary unimpeded, are afforded security services via AH or ESP, or are discarded.

IPSec provides an end-to-end security between end hosts and all intermediate nodes. IPsec has the two following weaknesses [10, 126]:

- It does not support the upper layer
- Because it needs key exchange, it will use IKE management, which requires a valid IPv6 address. So it cannot work when a new host joins a network therefore it is not able to protect Network Discovery Protocol

Because of the complex configuration, most of the users do not implement IPsec for link-local addresses.

Zheng et al. [80] proposed algorithms to speed up the pattern matching for Network Intrusion Detection, by introducing the concept of Negative Pattern Matching (NPM) that splices flows into segments for fine-grained load balancing and optimized parallel speedup while ensuring correctness. Then they proposed the idea of exclusive pattern matching which divided the rule sets into subsets, where each subset is queried selectively, and independently given a certain input without affecting correctness.

Al-mamory et al. [76] proposed a Network Intrusion Detection System by using anomaly techniques to detect intrusion. Their approach is used to detect the known and novel attacks in traffic network. The proposed system operates in two grains levels. The first one works with basic features while the second mode works with statistical features of captured packets. The combination of both allows IDS to analyse network traffic on different granularities. The detection levels are “Coarse-Grained” IDS and “Fine-Grained” IDS. To increase IDS performance when traffic is normal the system will monitor five features and the level is set to Coarse-Grained. When an intrusion is detected by Coarse-grained, the other level Fine-Grained will activate with monitoring of twenty features to be able to detect attacks. Their system has four processes that are, data collection, pre-processing, classification and response.

Kempf et al. [128] proposed SEcure Neighbour Discovery (SEND) protocol to mitigate the issue of IPsec for link-local communication. SEND is an extension of NDP that adds several options such as Cryptographically Generated Addresses (CGA), RSA Signature and Timestamp and Nonce Options. In addition, they introduce four new Authorization Delegation Discovery, Certification Path Solicitation Message Format, Certification Path Advertisement Message Format, Router Authorization Certificate Profile and Suitability of Standard Identity Certificates [127, 128].

A review of SEND done by *Meinel et al.* [129] challenges SEND as it does not provide link layer security and cover NDP communication confidentiality. The Cryptographically Generated Addresses cannot assure the real node identity. Because of the structure of SEND, it will use more CPU of nodes and bandwidth to process. In addition, if Router Authorization and Standard Identity Certificates are implemented into routers, it will put an extra workload on them.

Gilad et al. [55] proposed model is for blind spoofing, hijacking and data insertion into TCP/IP sessions. Their attacks are based on predicting the IP-ID value used in packets between the end-host source and destination, and then exploiting the predicted IP-ID to cause packet loss, interception or modification (for fragmented packets). To carry out wrong reassembly and packet losses, attackers will match some of the legitimate fragments that have arrived.

Modification (of only part of the packet) works similarly, except that the attacker has to carefully construct his fragment so that its content will replace the desired parts of the original payload. In most scenarios, interception is also easy, when the attacker controls a zombie machine behind the same Network Address Translator (NAT) as the destination, as illustrated in Figure 3.34 and the attacker can intercept a packet by changing the destination port specified in the first fragment.

Figure 3.34: Showing obtaining IP ID by attacker [55]

Due to of the changes made, IPv6 supports only source fragmentation. The initial work done by Kent and Mogul [105] mainly presented performance and reliability concerns. The fragmentation is still widely used, especially for UDP and Virtual Private Network. They mention that the fragmentation which is used in IPv6 is still vulnerable to their attacks.

An attacker can usually learn the value of a global IPID counter by simply receiving a packet from the sender. Receiving such packets is often possible, e.g., as a response to a packet sent by the attacker (e.g., SYN to public web server), or by causing the client to open a connection to the attacker. The attacks can be deployed by a blind (spoofing) attacker, and result in packet interception, modification and loss for both versions of IPs, IPv4 and IPv6. Their main conclusion is the need to improve the specifications and validation of common networking protocols such as IP, following (and motivating) [32]. The implementations of IPv4, IPv6, IPsec and other tunnels, should be carefully tested against the vulnerabilities described within and in particular modify their 'IP-ID choosing' paradigm. Furthermore, it is advisable that errata be issued especially for the relevant specifications. Considering that recently, with the adoption of mobile TCP/IP devices, there may be many new implementations. Finally, since many implementations may be impractical to fix in a timely fashion, appropriate defences should be added to firewalls and IDS/IPS devices.

3.4 Attack Type: Malware and Port scan

Rodrigues et al. [73] proposed a policy and network-based Intrusion Detection System for IPv6-enabled wireless sensor networks. They proposed a system using selected network nodes acting as watchdogs with the purpose of identifying the possibility of intrusions by using eavesdropping on the exchanged packets in the neighbourhood. These nodes perform local packet monitoring, and the main task is to eavesdrop on the exchanged messages between the nodes in their neighbourhood, which act as host-based IDS. As Figure 3.35 shows, the proposed system has two main application components, NIDS and EMS application. Each NIDS agent is configured with a set of rules. If any match is found between the rules and the packets, the system will generate an alarm and send it to the Event Management System (EMS).

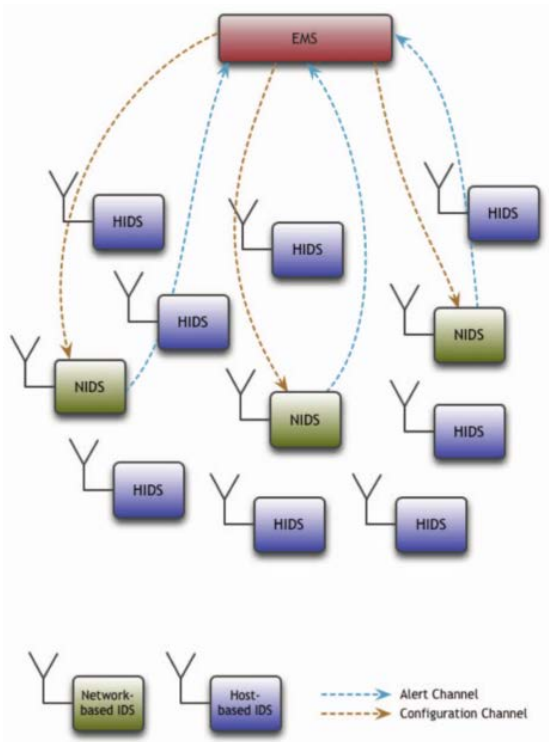


Figure 3.35: The proposed solution architecture [73]

A wireless sensor network is a diverse type of network and each NIDS is installed in a particular location of the network and cannot be configured with the exact same set of rules. Therefore, the rules in each of the watchdogs should be set and match as closely as possible to its neighbourhood traffic patterns. A policy programing approach is adopted into each NIDS to be able to dynamically configure each NIDS with a set of rules. The Packet Monitoring module collects audited data and activities within the radio range of each NIDS, which is sent to the Detection Module. The Detection module is responsible for saving and matching the rules with the network traffic. All these rules are added by the network administrator and are occasionally sent to watchdogs to improve the detection performance. The weakness of this work is if one of a NIDS agents goes down it will affect the whole process, because as the authors explained all agents are rely on each other.

Mali et al. [111] proposed solutions to provide a benchmark for organizations who want to test their network against IPv6 vulnerabilities. They performed a qualitative analysis of some of the common security aspects of IPv6 protocol. They developed an automation script which they claimed can detect such vulnerabilities. The developed detection method is based on Request for Comments (RFCs) recommendations. The solution will analyse the incoming packets and compare them with RFCs. If any of the packets didn't match with RFCs it will mark them as "Malformed packet". The authors claim that their system will inspect the packets just based on the RFCs, so the proposed solution is not an Intrusion Detection System and it is just a packet analyser. We ran a few tests on this solution and we could bypass their "Analyser" without triggering the alarm mechanism.

Yangui et al. [136] proposed a worm detection and containment system for IPv6 network. The proposed system detection is based on the user that is sending the DNS queries. Their system architecture (Figure 3.36) has three components, which are, Monitoring Server, DNS Server and Warning Server. The monitoring server runs the DNS analysis engine and The Monitoring Server runs the DNS analysis engine and continuously monitors all DNS traffic, which is generated or received by any local host. The Monitoring Server can identify anomalous activities that are generated by a worm to their DNS server by applying DNS traffic data mining and comparing them with their history profile. The DNS Server is responsible for providing all DNS services and taking advantage of DNS hijacking to lead victim hosts' traffic to the Warning Server upon receipt of alerts. The Warning Server persuades and helps the users of the compromised hosts, after confirmation, to eradicate the intrusion and/or vulnerability as soon as possible. The detection of a worm-infected host is achieving by analysis and extracting all DNS query packets by Monitoring Servers.

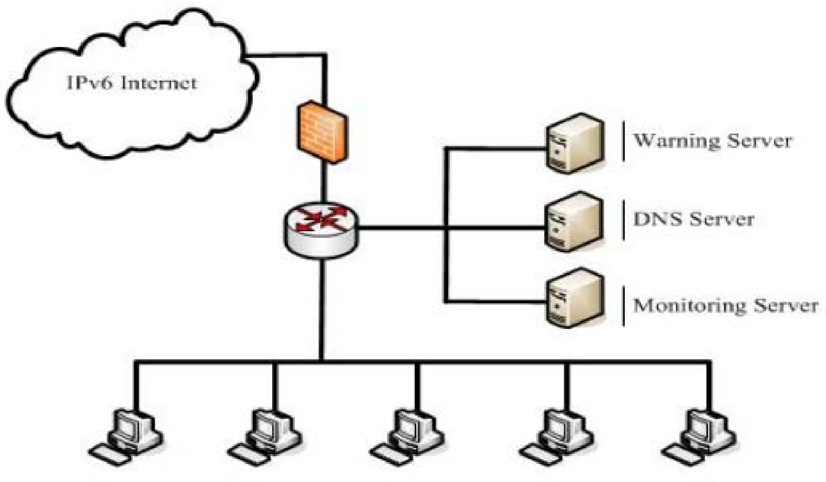


Figure 3.36: DNS-based worm detection [136]

Garfinkel et al. [7] presented an architecture that retains the visibility of a host-based IDS (Figure 3.37), but pulls the IDS outside of the host for greater attack resistance, by using a virtual machine monitor (VMM). This approach can isolate the IDS from the monitored host, but at the same time provides excellent visibility into the host's state.

Visibility of IDS makes evasion more difficult by increasing the range of analysable events, decreasing the risk of having an incorrect view of system state, and reducing the number of unmonitored avenues of attack. This means the more the IDS is visible, there will be a weaker isolation between the IDS and the attacker. And it will increase the risk of a direct attack on the IDS.

Network-Based Intrusion Detection System (NIDS) offers high attack resistance at the cost of visibility, and Host-Based Detection System (HIDS) offers high visibility but sacrifices attack resistance. Virtual Machine Monitor is the technique used in the proposed approach.

This technique allows pulling the IDS out of the host it is monitoring, into a completely different hardware protection domain, providing a high confidence barrier between the IDS and an attacker. "The VMM provides the ability to interpose at the architecture interface of the monitored host and better visibility with a motioning mechanism which can monitor both hardware and software level events". Only the IDS that is running outside of a virtual machine has access to hardware level state events. This will allow writing IDS polices as high-level statements in the OS, and thus retain the simplicity of a normal HIDS policy model. An architecture that allows more properties to be observed offers better visibility to the IDS. This allows an IDS's policy to consider more aspects of normative host behaviour, making it more difficult for a malicious party to mimic normal host behaviour and evade the IDS.

The author claims that it is very difficult for attackers to compromise the VMM, because everything in the VMM is unprivileged, and the VMM has only to provide isolation, with no concerns about providing controlled sharing. Isolation, Inspection and Interposition are three advantages for this approach. The VMM supports virtual I/O devices that are capable of doing Direct Memory Access (DMA). These virtual devices can use DMA to read any memory location in the virtual machine. The authors used this virtual DMA capability to support direct physical memory access in the VMM interface. The VMM allows the monitoring to enforce more restrictive protection of certain memory pages. VMMs may provide interfaces accessible from outside of a VM that provide an avenue of attack. A hosted VMM might be running on a host OS with a remotely exploitable network stack, or application-level network service.

Sujatha et al. [9] proposed an efficient NIDS. The proposed solution achieves pre-processing by using the dataset Support Vector Machine algorithm. This will also make a new data model which has been used for creating rules for misuse detection. The dataset can be classified into two datasets; namely Positive Kernel and Negative Kernel. Positive Kernel is used for creating the rules. After classifying the dataset, fuzzification (fuzzification comprises the process of transforming crisp values into grades of membership for linguistic terms of fuzzy sets; the membership function is used to associate a grade to each linguistic term) is applied to that dataset and then the rules have been created by Genetic Network Programming which is based on direct graph structure. In the testing phase, the system has been used to detect the misuse activities.

By combining SVM with Genetic Network Programming, this increases the performance of the detection rate of the Network Intrusion Detection Model and reduces the false positive rate.

The intrusions affect system performance as well as the security policy of the system. To detect intrusion and block malicious activities on the system, the author used a data mining technique. These data mining techniques detect the misuse activities as well as anomalous activities. That technique is classified in two ways. One is supervised learning and the other is unsupervised learning. Supervised learning is used to detect the misbehaviour activities and unsupervised learning is used to detect anomalous activities.

The proposed system architecture has two major functions, namely, Incremental Support Vector Machine (I-SVM) and Genetic Network Programming. The I-SVM mainly does the sample selection according to the RBF- kernel function.

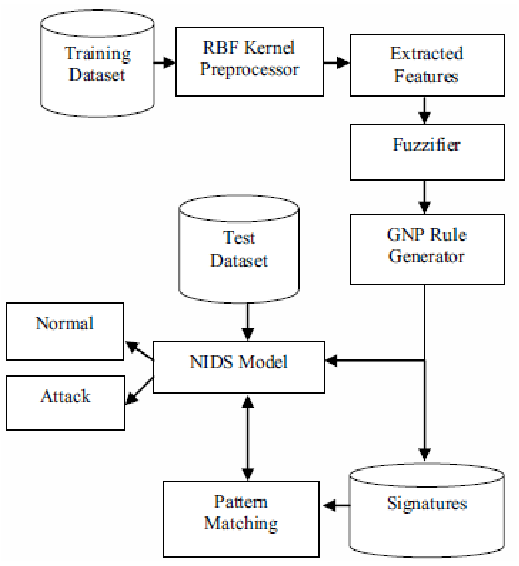


Figure 3.37: NIDS using Genetic Network Programming with Support Vector Machine [7]

The training dataset is used to create the rules and generated signatures are stored for pattern matching. In the testing phase the test dataset has been pattern matched with the NIDS model and finally the data are classified into normal attack or abnormal attack.

There are two key points to be taken into consideration. One is how to select samples to construct the reserved set, and the other is how to assign weights to each sample. The decision-function of SVM is determined by the support vectors and then determines which samples are most likely to be the support vectors. The pre-processed dataset is produced by the incremental SVM method. The training time increases accordingly as the size of the dataset increases.

For this work, the author picked nine attributes instead of forty-one, which reduce the false alarm and training time. The selected attributes are “server count, count, source bytes, destination bytes, diff_host_srv_rate, dst_host_port_rate, service, flag, proto_type”. The proposed solution didn’t analyse any specific attacks and it may not work against fragmentation attack.

Abuadlla et al. [89] proposed a two-stage intrusion detection system using two neural network stages by using flow data (Figure 3.38). The main objective of their work is that the system classifies and detects the attacks. The first stage of this work will detect attacks by monitoring significant changes in the network traffic while the second stage defines if there is a known attack and classifies the attack. The first stage uses a multi-layer feed forward neural network for attack detection and the second stage uses radial basis function networks for attack classification. The first stage uses six flow features and the second stage uses eleven flow features. The Netflow records used in this experiment are generated from the DARPA dataset by a tool called "softflowd tool". The authors of this paper promised a better detection accuracy with low probability of false alarm, however their evaluations are not guaranteed because the results with datasets will be different from the results in the real world.

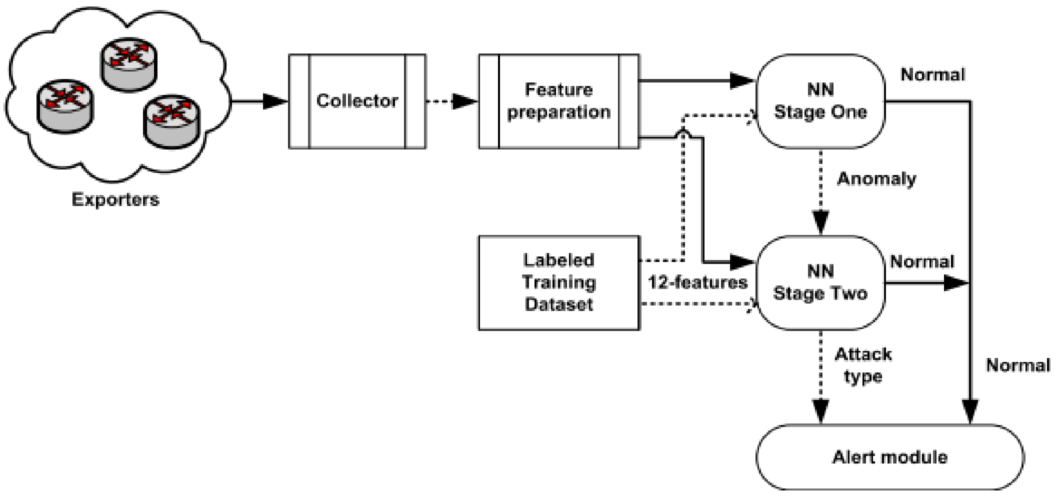


Figure 3.38: Implemented two stages NN based system Alert [89]

Kapravelos et al. [69] proposed an automated approach to automatically detect evasive behaviour in JavaScript called, Revolver. With efficient techniques, Revolver will able to identify similarities between large numbers of JavaScript programs (despite their use of obfuscation techniques, such as packing, polymorphism and dynamic code generation), and to automatically interpret their differences to detect evasion.

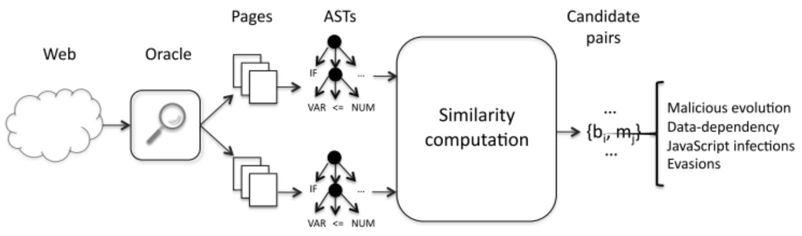


Figure 3.39: Revolver architecture [69]

The attacker can evade this method by breaking the code into two different scripts. As long as each script has got an incomplete code, it is impossible for the method to detect the attack and evasion. On the other hand, it can launch both scripts in the array in the background of the webpage to infect the victim by downloading or installing malicious software.

Shingo et al. [70] proposed a novel fuzzy class-association rule mining method by combining fuzzy set theory with Genetic Network Programming, based on GNP for detecting network intrusions. GNP is an evolutionary optimization technique, which uses directed graph structures instead of strings in genetic algorithm or trees in genetic programming, which leads to enhancing the representation ability with compact programs derived from the reusability of nodes in a graph structure. The proposed solution can be flexibly applied to both misuse and anomaly detection in network-intrusion-detection problems.

Khayam et al. [71] proposed two joint network-host based anomaly detection techniques that detect self-propagating malware in real-time by observing deviations from a behavioural model derived from a benign data profile. The three detectors are named as maximum entropy detector, rate limiting detector and credit-based threshold detector. The entropy threshold was used to detect the intruders. As the authors mentioned in their paper, their work is based on the theoretical solution and results would be different when they develop it and test it in a real world scenario.

Govindarajan et al. [72] proposed a hybrid architecture involving ensemble and base classifiers for intrusion detection systems based on two classification methods involving multilayer perceptron, radial basis function, and an ensemble of multilayer perceptron and radial basis function. They proposed a hybrid architecture involving ensemble and base classifiers for intrusion detection systems. It has found that the performance of the proposed method is superior to that of single usage of existing classification methods such as multilayer perceptron and radial basis function. In addition, it has been found that an ensemble of multilayer perceptron is superior to ensemble of radial basis function classifier for normal behaviour and the reverse is the case for abnormal behaviour.

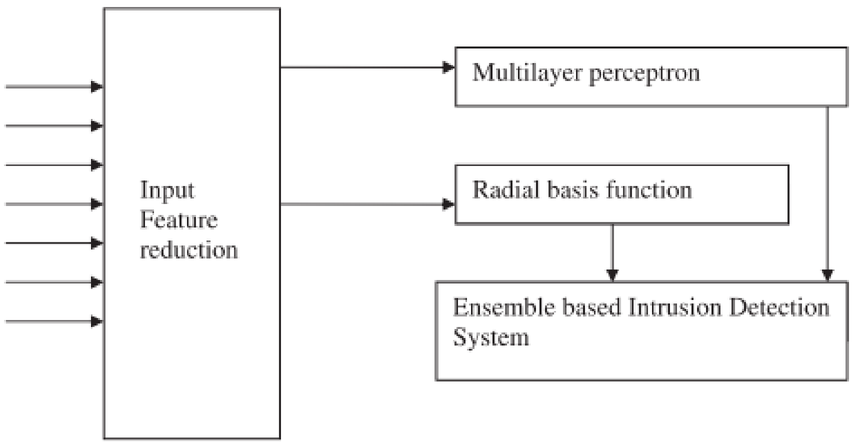


Figure 3.40: Ensemble approach for IDS [72]

Vykopal et al. [90] presented a detection method based on network for Remote Desktop Protocol (RDP), in which the authors claim that the current approach is only applicable in environments where the devices are under the control of the network administrator. The data analysis is based on network flow data collected from the Masaryk University (in the Czech Republic) network and host based data from open RDP server logs. They implement their detection with NfSen plug-in with derived NetFlow signature. The detection reported that approx. 45% of all RDP which is related to the traffic in the university campus, is malicious. As Flow-based techniques do not have access to the packet payload, the IDS cannot detect attacks that are embedded in the packet payload, (i.e. SQL Injection or XSS attack) and also the value of the flow export interval critically affects the performance of flow-based IDS.

Sperotto et al. [92] proposed an extension for Netflow and IPFIX flow exporter to detect timely intrusion detection and mitigation of large flooding attacks. The first stage was to implement a lightweight intrusion detection module into flow exporter of NetFlow/IPFIX, which moved the detection closer to the traffic observation point. The intrusion detection module uses a time series forecasting based on exponentially weighted moving average (EWMA) where the DDoS attack related metrics were measured and compared with forecast value. If the measured value of traffic does not lie within the range of the forecasted value, it will consider it as malicious. The characteristics of malicious flows will add to a blacklist, which is used to filter out the malicious traffic from clean traffic. If multiple attacks arise, it will reduce the performance and IDS would not be able to inspect the flow and detect an attack at the speed at which they have been collated. The detection accuracy was 92% with 0.01% of false positive rate for a span ("*represents the length in seconds of the history considered by the detection algorithm*") of

900s. The data used for evaluating this method was captured from the backbone network of Czech national research and education network.

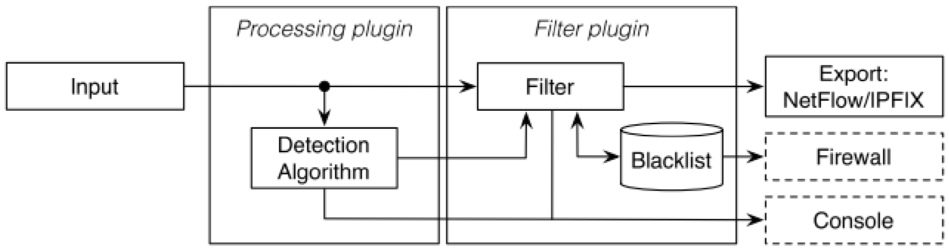


Figure 3.41: Prototype Architecture [92]

Hellemons et al. [93] proposed a flow based solution to detect SSH attacks called SSHCure. The proposed system has two phases of detection algorithm for attack detection. The first phase is call brute-force. This phase detects an attack by checking the same number of packets per flow. Two unsuccessful connection attempts from an attacker will have the same number of packets per flow. The second phase is called the compromised phase where it has the sixattack scenario. If the SSH traffic flow matches with a particular attack scenario, an attack is detected and the system will close the connection. Each attack phase uses a different threshold value for the two flow metrics. The proposed work obtains an accuracy of 84%. The weakness of this method is that the algorithm has various shortcomings that ultimately raise false alarms or cannot detect the remaining attacks.

3.5 Conclusion

This chapter has provided a summary of related work from many different areas of research that could be applied in detecting malicious traffic within the IP network. It has reviewed the plausible identified techniques, providing an overview and outlining both the benefits and shortcoming with respect to use in an IPv6 network. These techniques were also analysed for both their benefits and shortcomings.

Table 3.8 provides a short summary of the detection methodology techniques that are used by each work and highlighting their disadvantage(s).

Table 3.8: Summery of related work chapter

Detection Method	**Disadvantage**
Signature-based [112]	• Low detection accuracy • No Implemented solution • Inefficient against using fragmentation
Cluster Analysis [113]	• Working based packet volumes • No packet decoder or analyser
Dynamic Evolving Neural Fuzzy Inference System (DENFIS) [131]	• No packet decoder or analyser • Vulnerable to methods proposed in this thesis
back-propagation neural network [132]	• Can evade detection by hiding traffic into extension headers as the method filters the traffic just to ICMPv6 packets.
Flow-based [87]	• Not able to detect packet at high speed • Not able to decode IPv6 packet
unknown [133]	• Can be easily evaded by using a mixture of Extension Headers
Anomaly Detection [75]	• It's anomaly detection and that could take time to find the difference between the normal and abnormal traffic • Not supporting IPv6
Anomaly [11]	• It cannot handle the fragmented packet • Not supporting IPv6 • Only detects the attack on application level (i.e. DoS attack using HTTP request)
Flow-based Anomaly [91]	• Just work on TCP • Not supporting IPv6
Signature-based [13]	• Fail to detect Tiny Overlapping attack when SYN=0 and ACK=1
Anomaly [74]	• Generate too muchnoise that results in lower detection accuracy
Anomaly [83]	• Not able to detect fragmentation attack • Low detection rate when using "unknown attack" • Only able to detect pre-defined attacks
Signature-based [8]	• Low performance • Does not decode or analyse packets
Signature-based Anomaly [78]	• Relay on Snort • Uses machine learning, therefore the detection is based on the learning algorithm and do not early detection
Signature-based [88]	• Only works with TCP packet and TCP reassembly • The detection relies on algorithm
unknown [16]	• Not supporting Upper layer • Not able to protect Network Discovery Protocols
Signature-based [80]	• Not able to detect fragmentation attack
Anomaly [76]	• Vulnerable to fragmentation attack • Does not support IPv6 packet
Protocol [128]	• Not able to use on all OS

	• Too Complex • Extra workload on nodes
unknown[55]	• Relies only on IP-ID • Does not have any packet analyser or decoder • Can bypass using fragmentation
Signature-based Anomaly [73]	• Difficult to implement • It does not generate signature for all network traffics
Signature-based [111]	• Can be evaded by the proposed solution in this thesis • The detection is just based on RFC recommendations
Anomaly [136]	• Vulnerable to DoS/DDoS attack as can bring the detection down • It is only detecting worms
Host-based [7]	• A hosted VMM might be running on a host OS with a remotely exploitable network stack or application-level network service
Signature-based [9]	• Does not detect fragmentation attack • Does not support IPv6
Anomaly [89]	• Does not decode packets, therefore it cannot see the packet contents • Does not support IPv6
Signature-based [69]	• It's a web-based solution to detect malware
Anomaly [70]	• Low detection rate • Does not support IPv6 • Does not have packet decoder
Signature-based [71]	• Consumes too much resources • It works only on malware • Does not have any packet analyser
Anomaly [72]	• Low rate detection • Does not detect attacks based on IP,TCP or UDP
Flow-based [90]	• Cannot decode packet correctly, therefore cannot detect attacks that embedded with payload
Anomaly [92]	• Inefficient against using fragmentation • Only detects DDoS attack
Anomaly [93]	• False alarm • Low detection accuracy
Signature-based [53]	• Bypass detection using the proposed methods on this Thesis • Weak packet decoder compared to Suricata [3] and the proposed framework on this Thesis • Cannot detect DoS attack • Can be evaded using Extension Headers
Signature-based [115]	• Bypass detection using the proposed methods on this Thesis • Uses more resources on busy network • Cannot Detect DoS attack • Can be evaded using Extension Headers

Lastly, a short and brief finding of our literature review, we can categorise the gaps of existing solutions and research into the following:

- Limited technical research for IPv6 evasion attacks
- Limited in-depth research for IPv6 security challenges
- Limited technical research for IPv6 evasion detection methods
- Limited number of NIDS for IPv6
- Limited number of IPv6 packet decoders

Chapter 4

Proposed Solution

4.1 Introduction

This chapter proposes a framework for detecting Evasion, Insertion and DoS attack in IPv6 when using fragmentation and extension headers. This chapter starts by explaining the requirement of the new detection system in §4.2 and then the design and implementation of the proposed framework in §4.3.

4.2 Requirements of Proposed Solution

By reviewing, the outcome from previous sections in this chapter and considering the aims and objectives of this research from Chapter 1, it is possible to create a set of requirements, which define the characteristics that potential NIDS solutions must possess. These requirements can therefore be used to criticise the suitability of existing solutions and can help to ensure the success of the proposed framework and provide a useful mechanism to evaluate the proposed framework's accuracy at a high-level. These requirements were devised by examining the attributes of IPv6, evasion methods and detection needs, they are as follows:

- **Detection Accuracy**: It must provide a high detection accuracy and low-level of false positive and false negative alarms.
- **Both Real-time and Offline Detection**: It must able to provide detection for both Real-time traffic and offline (saved traffic) traffic
- **Scalable**: It should be developed in a way that is expandable easily.
- **Detection Type**: It should be able to detect attack based on packet type or packet signature.
- **Lightweight**: It should be using resources (CPU, RAM) as little as possible.

- **Easy to Use**: To use the solution, should not depend on any prior knowledge where the other existing solutions needed an expert to run them.
- **Efficiency**: It must provide a satisfactory level of protection in detecting threats on the network.

4.3 Design and Implementation

The new features in IPv6 present numerous security challenges to existing detection techniques, which were developed to detect intrusion and evasion in a network. Existing approaches lack the capability to protect the network when some of these features such as an extension header are used by an attacker to bypass the detection.

The aim of this thesis is to create a solution capable of detecting Evasion, Insertion and Denial of Service (DoS) attack. So far, this thesis has discussed in detail the structure of Internet Protocol (IP), the security challenges and vulnerabilities that IPs are facing and the limitations of existing solutions. In order to combat those, a new solution has been created called NOPO framework.

It is obvious from the previous chapters that there is a need for a new and capable detection solution to cope with the challenging structure and complexity of IPv6 network. The literature review has shown there are no entirely suitable solutions, and it highlighted the shortcomings of many existing popular techniques. Therefore, a novel approach is required to be able to combat against IPv6 evasion attacks and techniques. Additionally, a new viable approach is required to decode the packet correctly and extract all information that is needed to analyse to detect the intrusions or malicious activities. This approach also needs to adhere to the aims and objectives set out in §1.3 and the requirements outlined in §4.2.

Referring to all the aforementioned (RFC2460, 1998; RFC 3964, 2004; RFC 7123, 2014) recommendations of corresponding IPv6 Requests for Comment (RFCs) and previous sections, when using IPv6 Extension Headers and IPv6 Fragmentation there are potential attacks against the Operating System (OS). In case of discrepancies between the behaviour of several OS, this can lead to OS fingerprinting, Intrusion Detection System (IDS) insertion and IDS evasion, and Firewall evasion. Furthermore, there are still some issues regarding the handling of the IPv6 fragments [6]. One of the simplest examples of the one of the most common attacks can be fragmentation attack, which is common between IPv4 and IPv6.

Several IPv6 fragmentation and overlapping methods were used to test the effectiveness of some of the most popular OS and it found none of them is fully RFC compliant while most of them seem to have significant issues. Figure 4.42 shows a fragmentable IPv6 packet header followed by showing an IPv6 fragmented packet structure. A fragmented IPv6 packet has the following structure:

1- IPv6 Header [Version, Traffic Class, Flow Label, Payload Length, Next Header, Hop Limit, Source address, Destination Address] and Next Header set to 44 (Identification number for Fragment Extension Header)
2- Fragment Header [Next Header, Reserved, Fragment Offset, Res, More Fragment, Identification number]
3- The Fragment part of the packet (data or payload)

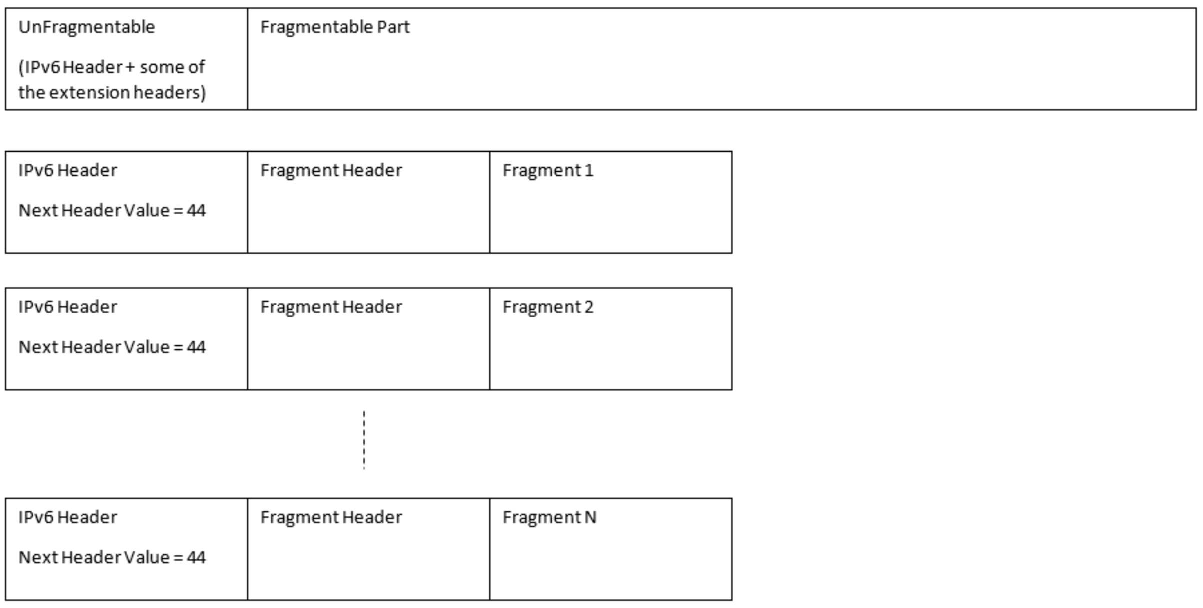


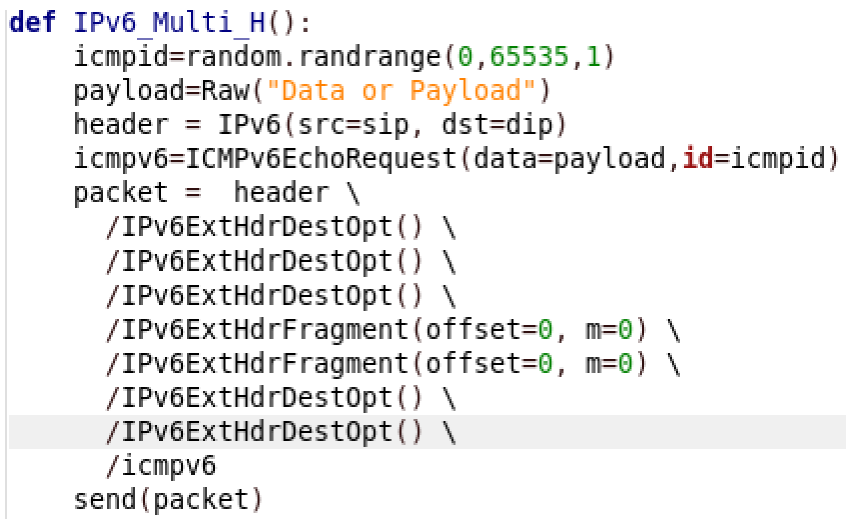
Figure 4.42: IPv6 Fragmentation packet header

In this chapter, a solution has been proposed which leads to developing a framework that can some different methods, which can be used by attackers to bypass Intrusion Detection System without being detected.

The code shown in Figure 4.43 is the simple code which is used to send an ICMPv6 (Ping Version 6 request) to the Virtual Machines. This code has the following variables:

- **icmpid** = generate random identification numbers from 0-65535 by increment or decrement of 1
- **payload** = it is a simple data or payload
- **header** = IPv6 source and destination address
- **icmpv6** = ICMPv6 Echo Request (data is the payload value and id is the random id generated on “icmpid”)
- **packet** = Here we placed previous variables to create an IPv6 packet. The variables are: header + 3 Destination Option Headers + 2 Fragment Extension Headers + 2 Destination Option Headers + icmpv6
- **send** = send the packet

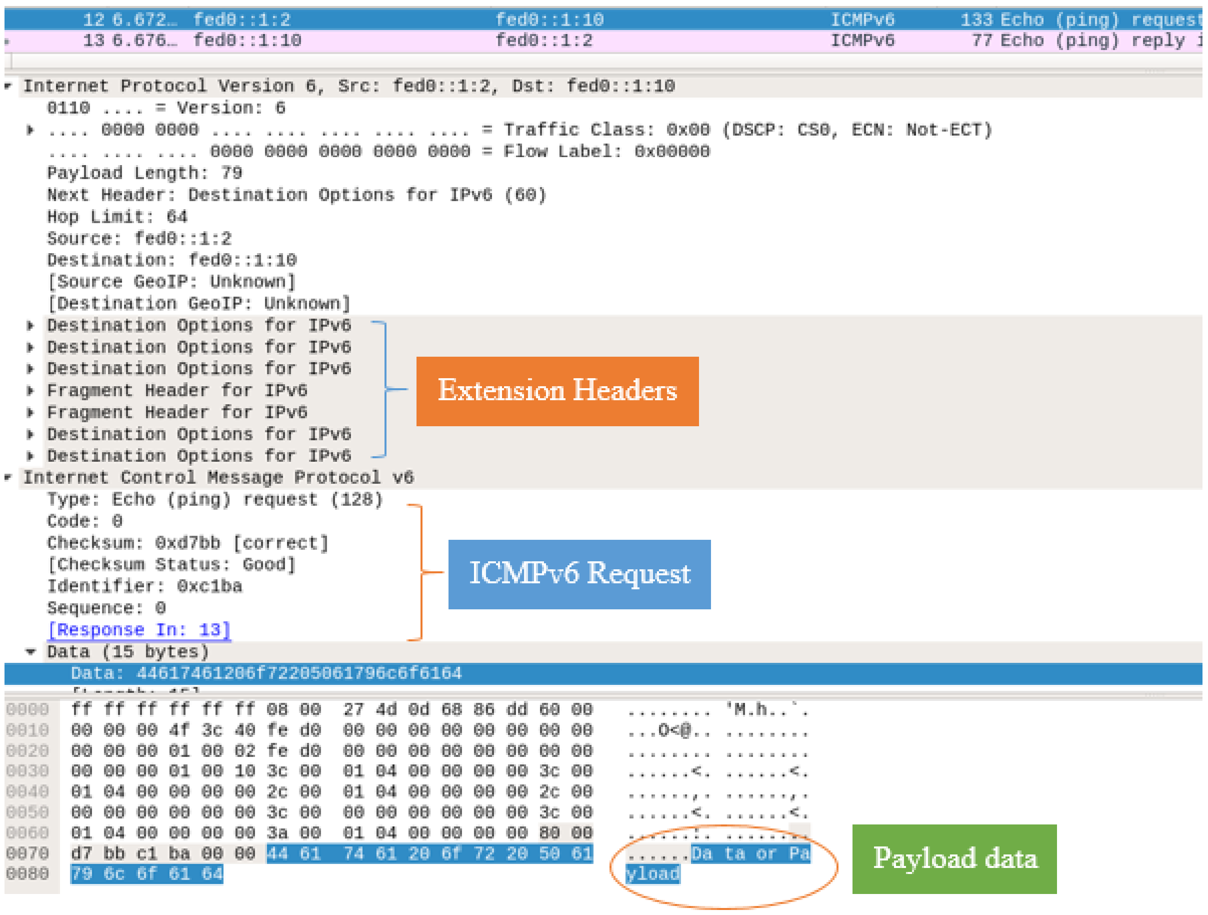
In this code we used 7 Extension Headers, 5 Destination Option Headers and 2 Fragment Extension Headers and also an ICMPv6 (Ping Version 6) request to send to an end-hosts. Wireshark were used to capture the packets.



```
def IPv6_Multi_H():
    icmpid=random.randrange(0,65535,1)
    payload=Raw("Data or Payload")
    header = IPv6(src=sip, dst=dip)
    icmpv6=ICMPv6EchoRequest(data=payload,id=icmpid)
    packet =  header \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrFragment(offset=0, m=0) \
      /IPv6ExtHdrFragment(offset=0, m=0) \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrDestOpt() \
      /icmpv6
    send(packet)
```

Figure 4.43: Multiple Extension Headers with payload

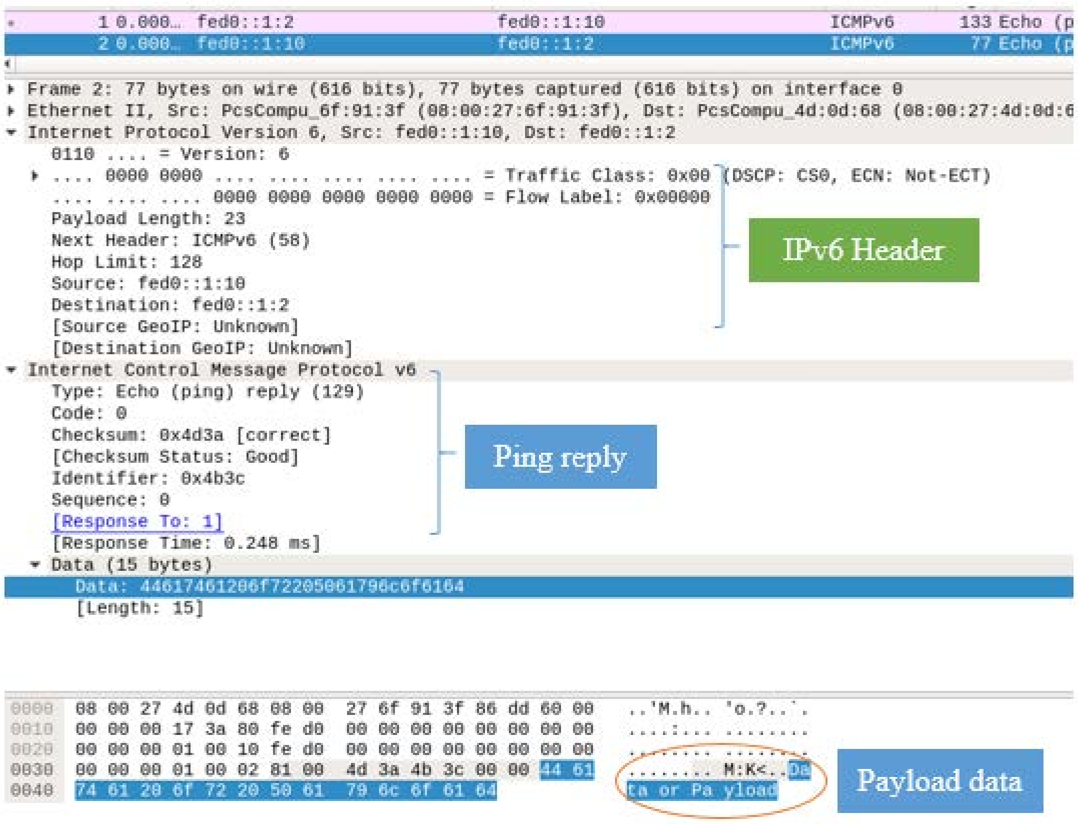
Figure 4.44 shows the captured packet that was sent to our end-host (Windows 7) that contained the data, seven extension headers and a ping request at the end with payload data “Data or Payload”.



```
   12 6.672…  fed0::1:2              fed0::1:10             ICMPv6      133 Echo (ping) request
   13 6.676…  fed0::1:10             fed0::1:2              ICMPv6       77 Echo (ping) reply i

▾ Internet Protocol Version 6, Src: fed0::1:2, Dst: fed0::1:10
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 79
    Next Header: Destination Options for IPv6 (60)
    Hop Limit: 64
    Source: fed0::1:2
    Destination: fed0::1:10
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
  ▸ Destination Options for IPv6
  ▸ Destination Options for IPv6
  ▸ Destination Options for IPv6
  ▸ Fragment Header for IPv6
  ▸ Fragment Header for IPv6
  ▸ Destination Options for IPv6
  ▸ Destination Options for IPv6
▾ Internet Control Message Protocol v6
    Type: Echo (ping) request (128)
    Code: 0
    Checksum: 0xd7bb [correct]
    [Checksum Status: Good]
    Identifier: 0xc1ba
    Sequence: 0
    [Response In: 13]
  ▾ Data (15 bytes)
      Data: 446174612006f72205061796c6f6164

0000  ff ff ff ff ff ff 08 00  27 4d 0d 68 86 dd 60 00   ........ 'M.h..`.
0010  00 00 00 4f 3c 40 fe d0  00 00 00 00 00 00 00 00   ...O<@.. ........
0020  00 00 00 01 00 02 fe d0  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 01 00 10 3c 00  01 04 00 00 00 00 3c 00   ......<. ......<.
0040  01 04 00 00 00 00 2c 00  01 04 00 00 00 00 2c 00   ......,. ......,.
0050  00 00 00 00 00 00 3c 00  00 00 00 00 00 00 3c 00   ......<. ......<.
0060  01 04 00 00 00 00 3a 00  01 04 00 00 00 00 80 00   ......:. ........
0070  d7 bb c1 ba 00 00 44 61  74 61 20 6f 72 20 50 61   ......Da ta or Pa
0080  79 6c 6f 61 64                                     yload
```

Figure 4.44: Sent packet to windows 7

We sent a packet with payload, which in this case, is “Data or Payload” to end-host. As is visible in Figure 4.45 the end-host replied to our ping request in Figure 4.44 and included the same payload data sent previously.



```
      1 0.000…  fed0::1:2                fed0::1:10              ICMPv6      133 Echo (p
      2 0.000…  fed0::1:10               fed0::1:2               ICMPv6       77 Echo (p

▸ Frame 2: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_6f:91:3f (08:00:27:6f:91:3f), Dst: PcsCompu_4d:0d:68 (08:00:27:4d:0d:6
▾ Internet Protocol Version 6, Src: fed0::1:10, Dst: fed0::1:2
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 23
    Next Header: ICMPv6 (58)
    Hop Limit: 128
    Source: fed0::1:10
    Destination: fed0::1:2
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▾ Internet Control Message Protocol v6
    Type: Echo (ping) reply (129)
    Code: 0
    Checksum: 0x4d3a [correct]
    [Checksum Status: Good]
    Identifier: 0x4b3c
    Sequence: 0
    [Response To: 1]
    [Response Time: 0.248 ms]
  ▾ Data (15 bytes)
      Data: 446174612066f72205061796c6f6164
      [Length: 15]

0000  08 00 27 4d 0d 68 08 00  27 6f 91 3f 86 dd 60 00   ..'M.h.. 'o.?..`.
0010  00 00 00 17 3a 80 fe d0  00 00 00 00 00 00 00 00   ....:... ........
0020  00 00 00 01 00 10 fe d0  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 01 00 02 81 00  4d 3a 4b 3c 00 00 44 61   ........ M:K<..Da
0040  74 61 20 6f 72 20 50 61  79 6c 6f 61 64            ta or Pa yload
```

Figure 4.45: Windows 7 reply

For IDS to be able to detect the packets, a rule is added to the rules database (Figure 4.46) to trigger the alert if any of the incoming packets to the end host(s) contain a ping request.

- **Alert**: generate an alert when rule matched
- **icmp**: means the rule will apply to “icmp” packet
- **any**: used for source IP address. In our case it shows that the rule will be applied to all packets.
- **any**: used for the source port number. Since port numbers are irrelevant at the IP layer, the rule will be applied to all packets.
- **->**: sign showing the direction of the packet. In our example it will be applied to any incoming packet
- **any**: used for destination IP address and shows that the rule will be applied to all packets.

- **any**: used for the destination port number. Since port numbers are irrelevant at the IP layer, the rule will be applied to all packets.
- **msg**: if any match found the log will contain the message
- **itype**: the ICMP type number
- **sid**: stands for Snort ID and it is uniquely identify Snort rules
- **rev**: stands for "Revision" is used to uniquely identify revisions of Snort rules.

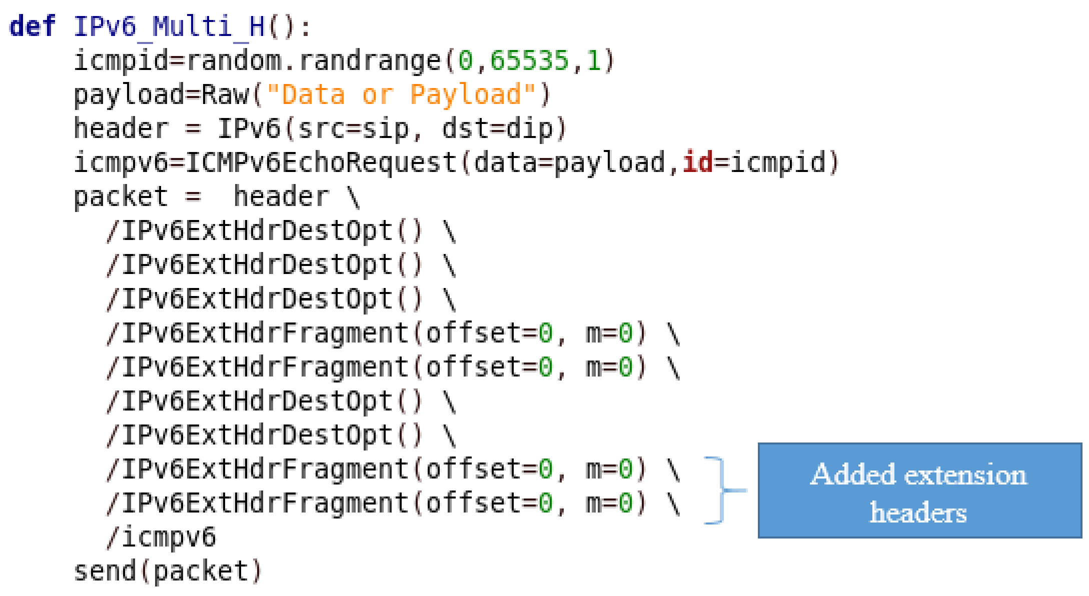
The result of the following rules will be if any incoming IPv6 packet has an ICMP type number 128 it will generate an alarm with message "Ping6 Detected".

Figure 4.46: Local rule (Snort)

By sending the packet shown in Figure 4.43 to end-hosts where there are two NIDSs on the network to detect any incoming packet with ping request (itype 128), the result shown both NIDSs detect the packet

By adding 2 extension headers to the packet (Figure 4.47), we successfully bypassed Snort. This means we evaded snort, but SURICATA detected the packet. This packet contains nine extension headers, five Destination Option Headers, four Fragment extension header and one ICMPv6 request header. The new packet has the following structure:

- **icmpid =** generate random identification numbers from 0-75498
- **payload =** it is a simple data or payload
- **header =** IPv6 source and destination address
- **icmpv6 =** ICMPv6 Echo Request (data is the payload value and id is the random id generated on "icmpid")
- **packet =** Here we place previous variables to create an IPv6 packet. The variables are: header + 3 Destination Option Headers + 2 Fragment Extension Headers + 2 Destination Option Headers + icmpv6
- **send =** send the packet



```
def IPv6_Multi_H():
    icmpid=random.randrange(0,65535,1)
    payload=Raw("Data or Payload")
    header = IPv6(src=sip, dst=dip)
    icmpv6=ICMPv6EchoRequest(data=payload,id=icmpid)
    packet =  header \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrFragment(offset=0, m=0) \
      /IPv6ExtHdrFragment(offset=0, m=0) \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrDestOpt() \
      /IPv6ExtHdrFragment(offset=0, m=0) \
      /IPv6ExtHdrFragment(offset=0, m=0) \
      /icmpv6
    send(packet)
```

Figure 4.47: Multiple Various Extension Headers in Atomic Fragments with two extra EH

A solution has been proposed to overcome the limitation of existing solutions and covering the gaps. The proposed solution is based on four elements and these are:

- Time: It describes that our solution detection is based on a real time monitoring of a network, the granularity of the system is continuous and the response method is notifications.
- Detection Strategy: describing that our processing strategy is centralised and detection methodology that we are using in our solution is signature.
- Data Source: Shows what our data source is and how it will be collected. The data will be captured from network traffic by sensor(s) and the data collocation is distributed.
- System Architecture: showing the architecture of our solution which is based on a wired/wireless network(s) and our IDS is network based

Before going into the details of the techniques and processes involved in NOPO, it is imperative to have an understanding of the system itself. Figure 4.48 shows the structure of the proposed system with an illustrative overview.

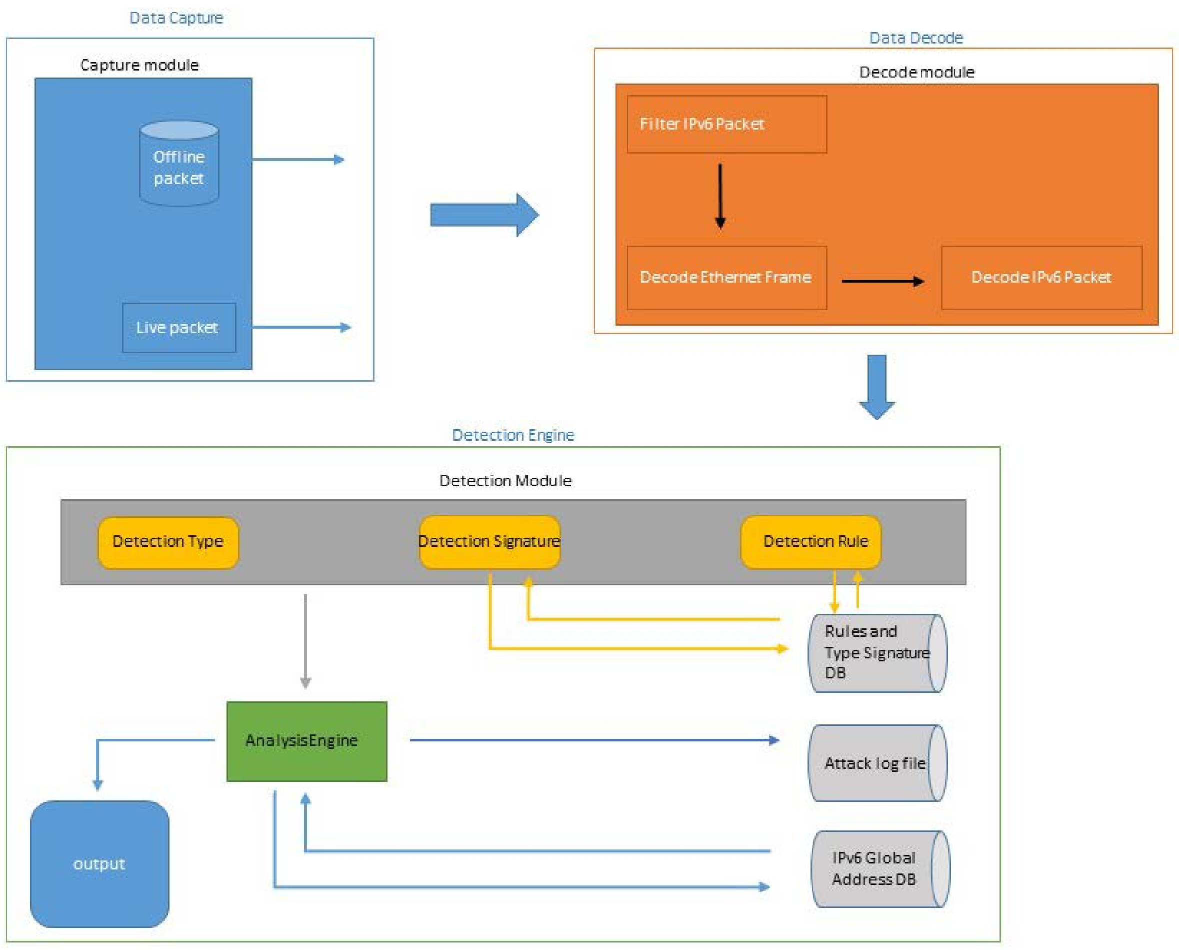


Figure 4.48: Illustrated overview of NOPO

The framework proposes a novel method by using a better way to decode the incoming traffic in the IDS and by defining roles and responsibilities to deal with packet decoding and detecting threads.

The proposed scheme has four main components, which are

- Capturing Packet
- Packet Decoding
- Detection Engine
- Output

An Incoming packet will go through the filtering process to filter out any unnecessary packets. Once packets reach the Packet Decoding process, the system will start to decode the packet. Then packets are sent to the Detection Engine where packets are evaluated using our Detection Engine process. A decision engine is the brain of this scheme incorporating and collaborating with other components to analyse the network traffic performance to detect network evasion and inserting into the log the details of the malicious activities and sending the notification to the sys-admin.

Our solution has three main processes, Packet capture (Figure 4.50), Packet Decoder (Figure 4.51) and Detection Engine (Figure 4.60). Packet decoder has seven sub-processes and Detection Engine has eight sub-processes.

4.3.1 Packet Capturing

The system starts by choosing between the live or offline packet capturing option. If "Live" streaming is selected then the user should choose on which interface the framework wants to collect the packets (e.g. eth0) and then packet capturing starts. If the "Offline" option is selected then the user should define the path to a PCAP file that they want to analyse.

Our system gives an option of analysing the captured packet by simply entering the path of the PCAP file. If this option is selected, the PCAP file will be directly sent to our Packet Decoding process. This built version of the solution only works with IPv6. On "Live" option, the system automatically filters out the packets after they have been captured and only sends "IPv6 packets" to the packet decoder.

Figure 4.49: Packet capturing code

Figure 4.49 shows the code for live sniffing and reading the packet from "PCAP" file which. On the "Live" part, we have three sub-process if "Live" selected:

- Open Interface: This will be the interface from which the framework will sniff all packets. On Linux machines, the user needs to run the framework as "root". Because normal users do not have the permission to sniff packets from interfaces.
- Start capture and Filter Packets: Because the proposed system only works for IPv6, a filtering function place to monitor only IPv6 packet.

- Send to Packet Decoder: This will send all captured packets from the selected interface to the Packet Decoder part.

On the “PCAP” part, we have three sub-process as well.

- Open PCAP file: this process will open the file that the user has chosen.
- Read PCAP file: in this process the PCAPfile content will be read by this process
- Send to Packet Decode: will send the read data to the packet decoder for the decoding process.

Figure 4.50 is an illustrated overview of the packet capturing process for a better understating of what will happen in practice with this process. The framework will start to work by selecting between “*Live*” sniffing and “*offline*” PCAP reading. Once one of the options is selected then depending on the selection, the sub process starts to work. If “Live” sniffing is selected then the user needs to choose an interface (e.g. eth0 or enp0s3) and then packet sniffing starts. The next process will do the packet filtering and allow only “IPv6” packet. The final sub process for this option is to send the captured packet to the Packet Decoder engine. If the “*Offline*” option is selected then the user needs to define a path to a “PCAP” file. Then the process opens and reads the PCAP file and sends it to the Packet Decoder.

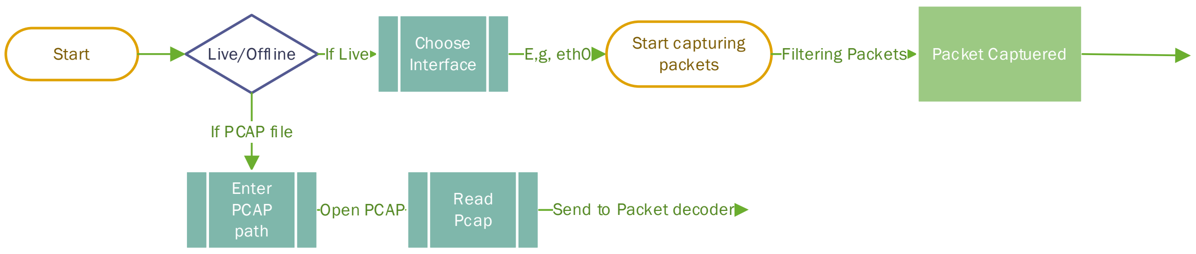


Figure 4.50: Packet capturing process

4.3.2 Packet Decoder

This process is one of the most important processes, which are not fully functional on other NIDS’s because, if an NIDS can translate (decode) a packet in a correct format, it can easily identify an attacks or threads. As shown in Figure 4.51, the Packet Decoder has two sub processes to extract the necessary information from a packet.

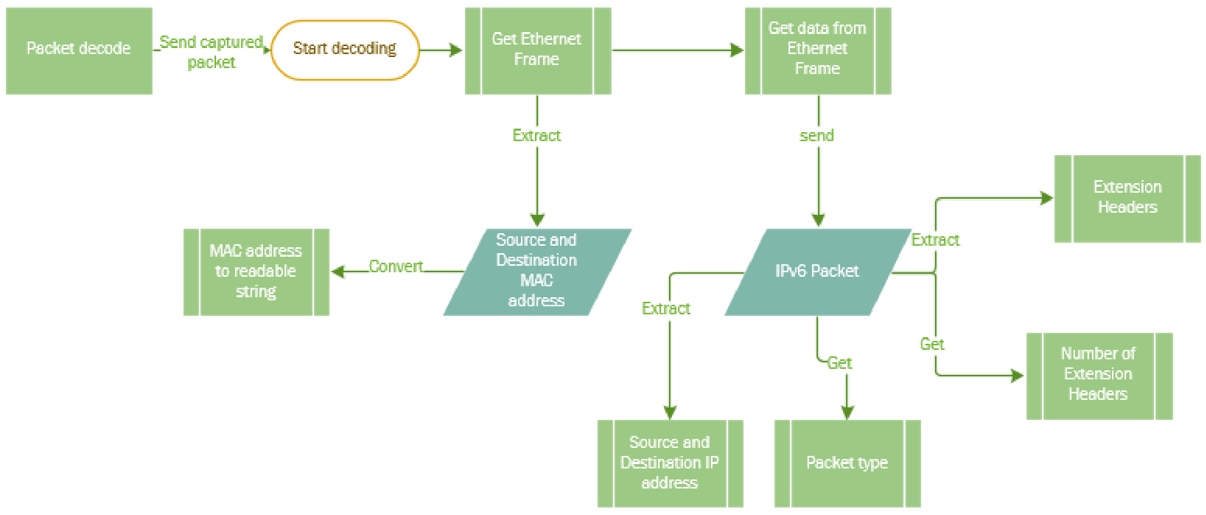

Packet decode
Send captured packet
Start decoding
Get Ethernet Frame
Get data from Ethernet Frame
Extract
send
Extension Headers
MAC address to readable string
Convert
Source and Destination MAC address
IPv6 Packet
Extract
Extract
Get
Get
Number of Extension Headers
Source and Destination IP address
Packet type

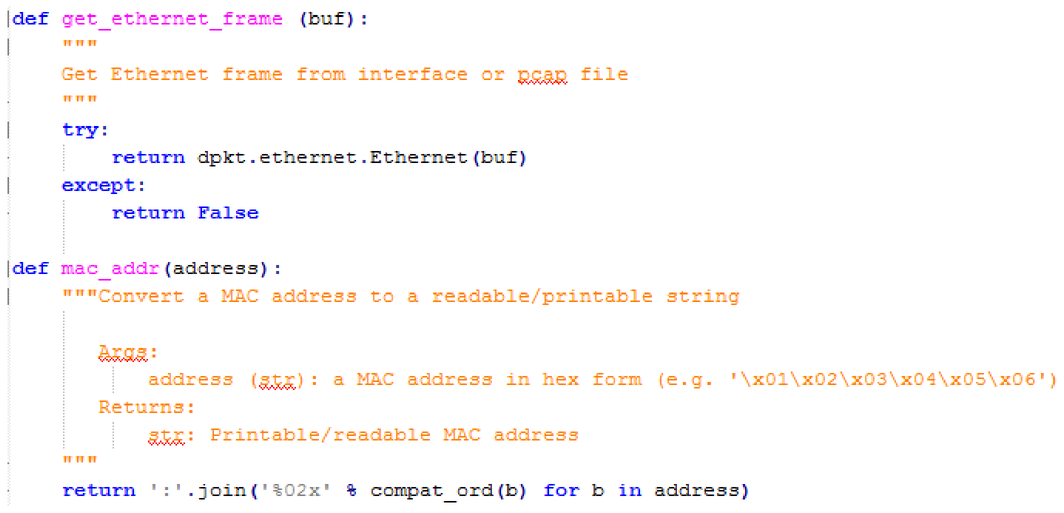
Figure 4.51: Packer decoder

4.3.2.1. Get Ethernet frame

This sub process (Figure 4.52) gets the packet Ethernet frame from the "*interface*" or "*pcap*" file and will try to extract the data from it. The Ethernet frame contains the following variables:

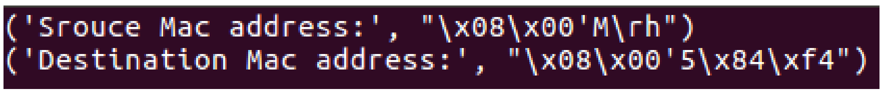
- Preamble: informs the receiving system that the frame is starting and enables synchronisation.
- SFD: Start Frame Delimiter signifies frame flag and indicates the start of a frame
- Destination MAC: Destination Media Access Control (MAC) address
- Source MAC: Source MAC address
- Type: Ethernet frame type
- Data: Payload data
- FCS: Frame Check Sequence will detect the corrupted data

To extract data from an Ethernet frame, the function will get the "*buffer*" from the captured packet from interface or read from the PCAP file. If any error occurred during this process, the function will "*return false*". Once the buffer data is received and extracted the next function will start to get and convert the MAC address from the Ethernet buffer. As shown in Figure 4.53, the MAC address is in hex format. We need to convert this to readable decimal address (Figure 4.54).



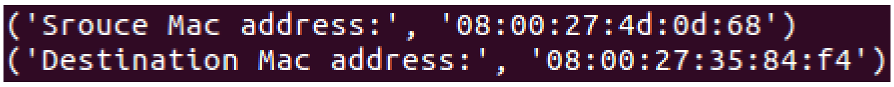
```
def get_ethernet_frame (buf):
    """
    Get Ethernet frame from interface or pcap file
    """
    try:
        return dpkt.ethernet.Ethernet(buf)
    except:
        return False

def mac_addr(address):
    """Convert a MAC address to a readable/printable string

       Args:
           address (str): a MAC address in hex form (e.g. '\x01\x02\x03\x04\x05\x06')
       Returns:
           str: Printable/readable MAC address
    """
    return ':'.join('%02x' % compat_ord(b) for b in address)
```

Figure 4.52: Get Ethernet frame code



```
('Srouce Mac address:', "\x08\x00'M\rh")
('Destination Mac address:', "\x08\x00'5\x84\xf4")
```

Figure 4.53: Un-converted MAC address



```
('Srouce Mac address:', '08:00:27:4d:0d:68')
('Destination Mac address:', '08:00:27:35:84:f4')
```

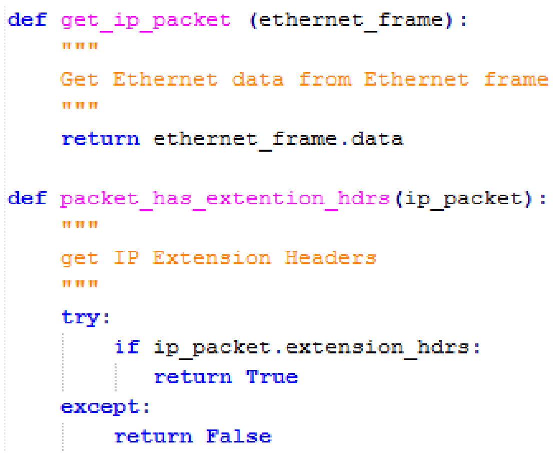
Figure 4.54: Converted MAC address

4.3.2.2. Get data from Ethernet frame

This sub-process (Figure 4.55) will get the data from the "*Ethernet frame*". The extracted data contain IPv6 header information. As mentioned before the IPv6 header has the following structure:

- Version
- Traffic Class
- Flow label
- Payload length
- Next Header
- Hop Limit
- Source Address
- Destination Address
- Extension Header (if there is any)

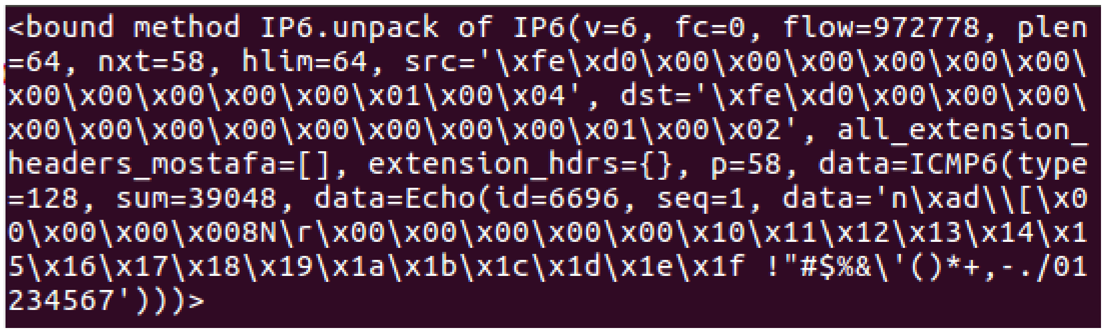
To get data from an Ethernet frame, a function name "*get_ip_packet*" created. This function will get the data from an Ethernet frame by using "*return ethernet_frame,data*". Once the data is collected from an Ethernet frame, then the system will be looking for any extension headers. As mentioned before an IPv6 packet can have zero, one or more extension headers.



```python
def get_ip_packet (ethernet_frame):
    """
    Get Ethernet data from Ethernet frame
    """
    return ethernet_frame.data

def packet_has_extention_hdrs(ip_packet):
    """
    get IP Extension Headers
    """
    try:
        if ip_packet.extension_hdrs:
            return True
    except:
        return False
```

Figure 4.55: Get Packet and Extension Headers

For better understanding of how a decoded IPv6 packet will look, a sample IPv6 packet is decoded in Figure 4.56. We generated this packet by using "ping6" command. This command is a Linux based command to perform a ping request on IPv6 network.

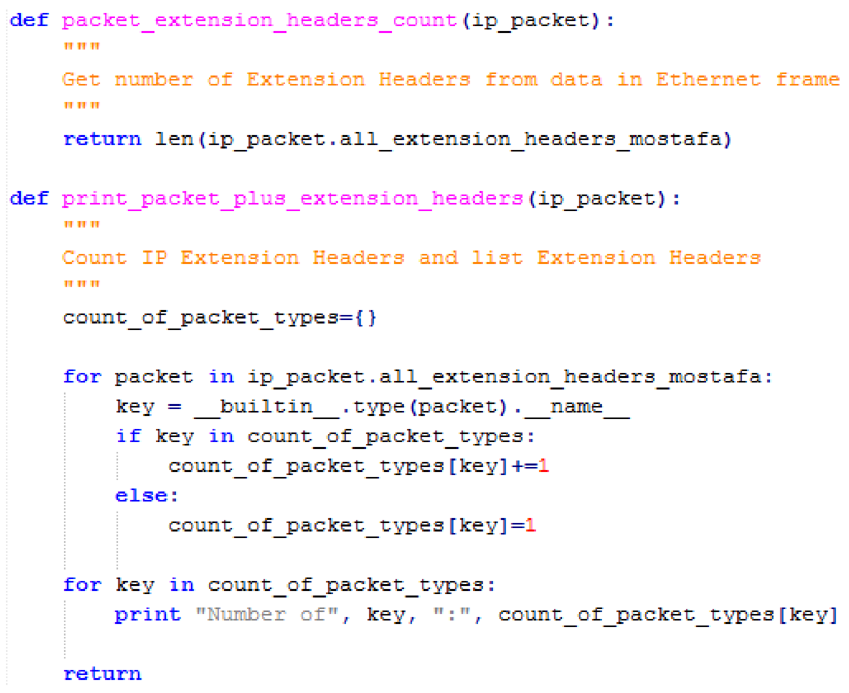


```
<bound method IP6.unpack of IP6(v=6, fc=0, flow=972778, plen
=64, nxt=58, hlim=64, src='\xfe\xd0\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x01\x00\x04', dst='\xfe\xd0\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x02', all_extension_
headers_mostafa=[], extension_hdrs={}, p=58, data=ICMP6(type
=128, sum=39048, data=Echo(id=6696, seq=1, data='n\xad\\[\x0
0\x00\x00\x008N\r\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x1
5\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01
234567')))>
```

Figure 4.56: Simple IPv6 packet decoded

One of our main objectives was to decode the packet and extract data correctly. To achieve this objective we need to have a reliable packet decoder. For the packet decoding process, we used a python module name "dpkt". Dpkt is a python module for packet parsing / creation with definitions for the basic TCP/IP protocol. During the development phase by debugging the module we founded some limitations and bugs, the standard data unpack function included in the module follows extension headers which means following its parsing, one has no access to all the extension headers in their original order. By defining, a new field called "all_extension_headers" and adding each header to it before it is moved along, will allows us to have access to all the extension headers while keeping the original parse speed of the framework virtually untouched. In addition, the extra memory foot print from this is also negligible as it will be a linear fraction of the size of the whole set of packet. The patch submitted the into github repository [110]. In that version, users are able to extract all extension headers in the packet in a correct format, which in the previous version users were not able to do. Therefore for extracting the packets with high numbers or different combinations of extension headers, the module will crash. An example of that could be, when a packet with more than 2000 extension headers is sent to one of end-hosts, the previous version of "dpkt" was not able to decode the packet or even see the packet.

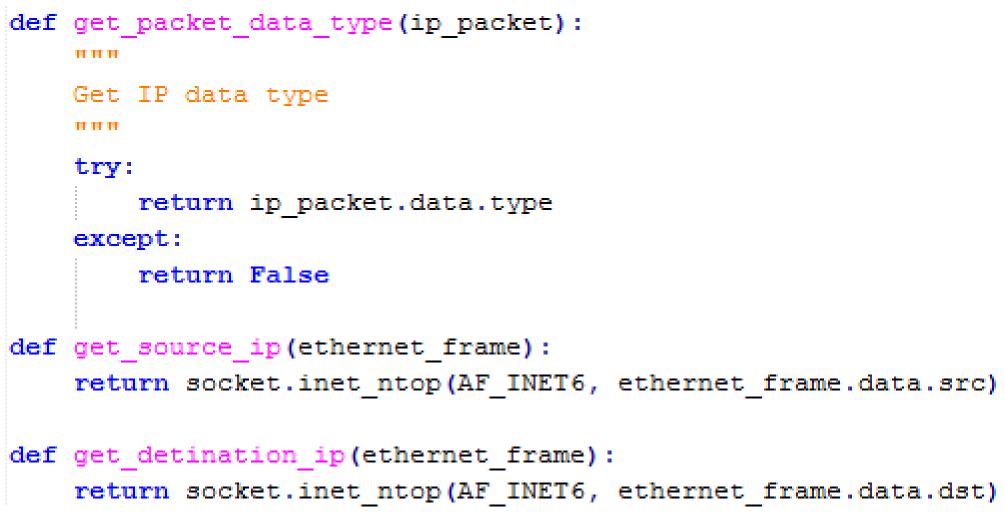
Figure 4.57 shows how the framework will get the extension headers from IPv6 header and count them. To get the extension headers a function was created called "*packet_extension_headers_count*". This function will get the extension headers from IPv6 by calling "*all_extension_headers_mostafa*" which the definition of edited part in original dpkt module (the patch provided to developer for the bug that founded in this module). The next function will start to count the number of extension header(s) in each packet.



```
def packet_extension_headers_count(ip_packet):
    """
    Get number of Extension Headers from data in Ethernet frame
    """
    return len(ip_packet.all_extension_headers_mostafa)

def print_packet_plus_extension_headers(ip_packet):
    """
    Count IP Extension Headers and list Extension Headers
    """
    count_of_packet_types={}

    for packet in ip_packet.all_extension_headers_mostafa:
        key = __builtin__.type(packet).__name__
        if key in count_of_packet_types:
            count_of_packet_types[key]+=1
        else:
            count_of_packet_types[key]=1

    for key in count_of_packet_types:
        print "Number of", key, ":", count_of_packet_types[key]

    return
```

Figure 4.57: Get number of extension header

The final part for this process is to get the IP address source, destination and packet type. As shown in Figure 4.58, the function called "*get_packet_data_type*" will get the IP packet data type if the packet is ICMPv6 otherwise return nothing. This will be achieved by calling "*ip_packet.data.type*" where "*ip_packet*" is IPv6 header and "data" is the IPv6 data.

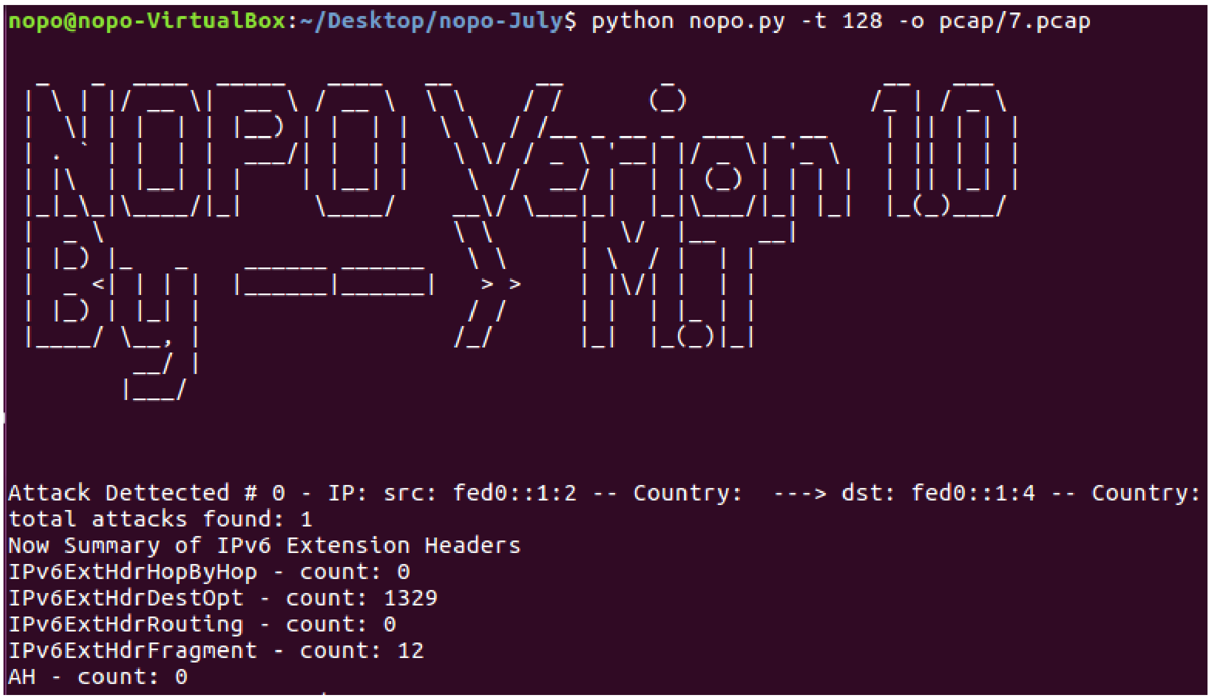


```
def get_packet_data_type(ip_packet):
    """
    Get IP data type
    """
    try:
        return ip_packet.data.type
    except:
        return False

def get_source_ip(ethernet_frame):
    return socket.inet_ntop(AF_INET6, ethernet_frame.data.src)

def get_detination_ip(ethernet_frame):
    return socket.inet_ntop(AF_INET6, ethernet_frame.data.dst)
```

Figure 4.58: Get packet type, source, and destination IP code

As has been shown in Figure 4.59, by improvement made to the packet decoder we could easily extract the information from each packet. In this example, we extract the number of extension headers that this packet contains. The result shows packet "*7.pcap*" matched with the type defined as a rule. The output of this packet analysing is as follows:

- **Attack detected**: showing source and destination address with the country that the IP belongs to. As we are doing the test on a local network, the country field is empty.
- **Total attacks found**: indicating how many match in total found on this pcap file
- **Now Summary of IPv6 extension headers**: showing how many extension headers the packet has and listing them by name and showing how many of each of them.



```
nopo@nopo-VirtualBox:~/Desktop/nopo-July$ python nopo.py -t 128 -o pcap/7.pcap

Attack Dettected # 0 - IP: src: fed0::1:2 -- Country:  ---> dst: fed0::1:4 -- Country:
total attacks found: 1
Now Summary of IPv6 Extension Headers
IPv6ExtHdrHopByHop - count: 0
IPv6ExtHdrDestOpt - count: 1329
IPv6ExtHdrRouting - count: 0
IPv6ExtHdrFragment - count: 12
AH - count: 0
```

Figure 4.59: Improved Version of Packet Decoder

4.3.3 Detection Engine

For a better understanding of how the Detection Engine works, an illustrated overview is drafted (Figure 4.60) to show how the sub-processes work together to analyse the packet. Once all data is extracted from packet in the previous process, then it is time for comparison between user-defined variables (Rule, Signature or Type). As shown in Figure 4.60 the packet data will be sent for decision making by one of the options below.

Rule: If sys-admin chooses "rule" then the user should define a path to the rule file (e.g.' /home/NIDS/rule.txt'). This file contains a local rule that sys-admin had already written into that file. These rules are based on the "HEX" value. For example, if sys-admin wants to set an alert for any "ping6" request, the rule will be "80" in hex language (128 in decimal value).

Signature: If sys-admin chooses signature, the user can add the signature that he/she is looking for. This is also in a 'HEX' format. This will help to identify a specific attack that sys-admin are looking for. If the incoming packets or PCAP file match with this signature, it will, send a match signal to the decision maker, if not it will continue inspecting incoming packets or PCAP files.

Type: Type is another option. With this option, NIDS will look for an attack type. For example, we blocked any ping request on our firewall. An attacker can use IPv6 Extension Headers to bypass this policy and find the live hosts on the network. Using this option is simple; sys-admin should just add the type here, which is "128". Be aware that the test we have done on other NIDSs (Snort and SURICATA) showed that they could not detect all the evasions using these methods.

Once a packet has been inspected and examined against the "Rule, Signature or Type", the result will be sent to the decision maker. If one of the above results was positive, it will send the decoded packet to the Global IPv6 Address database to find out the city and country of the packet.

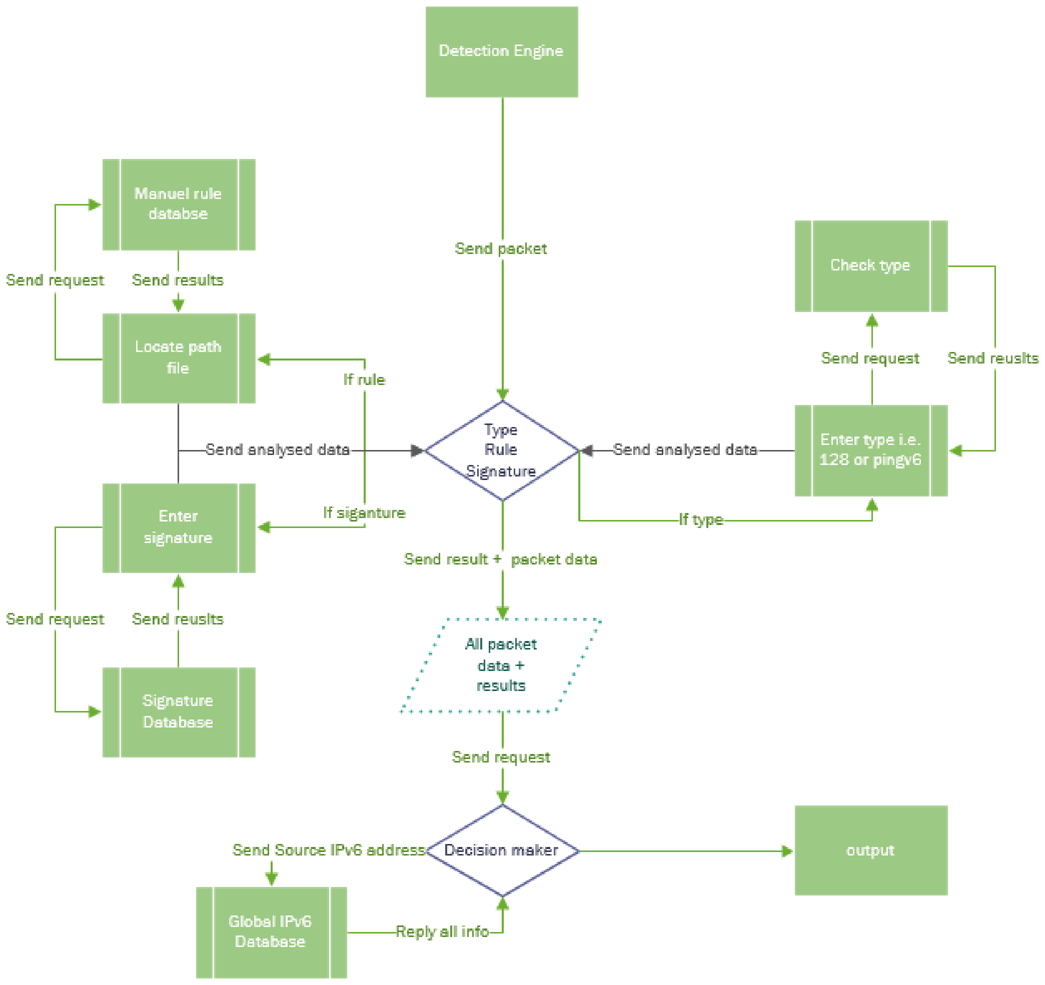


Figure 4.60: Detection Engine

For analysing packets, a function named “*inspect_packet*” is created (Figure 4.61). This function starts to work by looking for IPv6 Ethernet frame type to make sure that the packet is an IPv6 packet. To do that we need to define some variables. First, value “*IPV6_TYPE_ETHERNET*” set to “*dpkt.ethernet.ETH_TYPE_IP6*” which is a value of an IPv6 Ethernet frame type. Then we set “*ethernet_frame*” and called a previous function from §4 which is “*get_ethernet_frame(buf)*” and pass the result and assign the variable to “*ethernet_frame*”. The next step will start by making a condition, which is “*if ethernet_frame != False*”. By making this, if the packet has a valid “Ethernet frame” continue, if not, do not inspect the packet and showing “*IPv6 is only supported*” message.

If the previous condition met, another condition will take in place, which is “*if ethernet_frame.type == IPV6_TYPE_ETHERNET*”. “*ethernet_frame.type*” is the Ethernet frame type of the captured packet that is recalled and assign the variable from “*get_ethernet_frame(buf)*” to “*ethernet_frame*”. “*IPV6_TYPE_ETHERNET*” is value of “*dpkt.ethernet.ETH_TYPE_IP6*” that assigned previously. This condition will try to enforce only IPv6 packet pass this point and begin inspection. Now it is time to pass the data that was previously captured and decoded by Packet Decoder to the inspection system. In order to do this “*ip_packet*” is set to “*get_ip_packet(ethernet_frame)*” which is a function from Packet Decoder part and contains the data of the Ethernet frame. The data of Ethernet frame will be the IPv6 packet (IPv6 Header + if any extension headers + data).

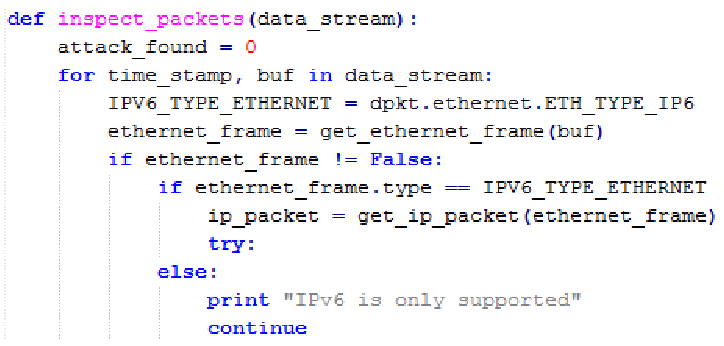
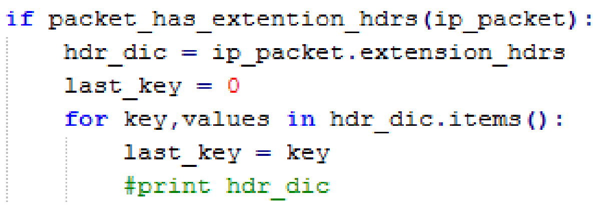

```
def inspect_packets(data_stream):
    attack_found = 0
    for time_stamp, buf in data_stream:
        IPV6_TYPE_ETHERNET = dpkt.ethernet.ETH_TYPE_IP6
        ethernet_frame = get_ethernet_frame(buf)
        if ethernet_frame != False:
            if ethernet_frame.type == IPV6_TYPE_ETHERNET :
                ip_packet = get_ip_packet(ethernet_frame)
                try:
            else:
                print "IPv6 is only supported"
                continue
```

Figure 4.61: Detection Engine code – Part 1

Figure 4.62 is the next part of our Detection Engine code where it will check if the packet has any extension headers. It will start by implying “*if packet_has_extention_hdrs(ip_packet)*” where the extension header(s) is already captured, decode and extract from packet in §4.3.1 and §4.3.2 . Then we will create a dictionary for extension header “*hdr_dic = ip_packet.extension_hdrs*” where “*ip_packet.extension_hdrs*” is the extension headers of the inspected packet. Once the Detection Engine had the all extension headers it will create a "*for loop*" to make list of the extension header with their data.

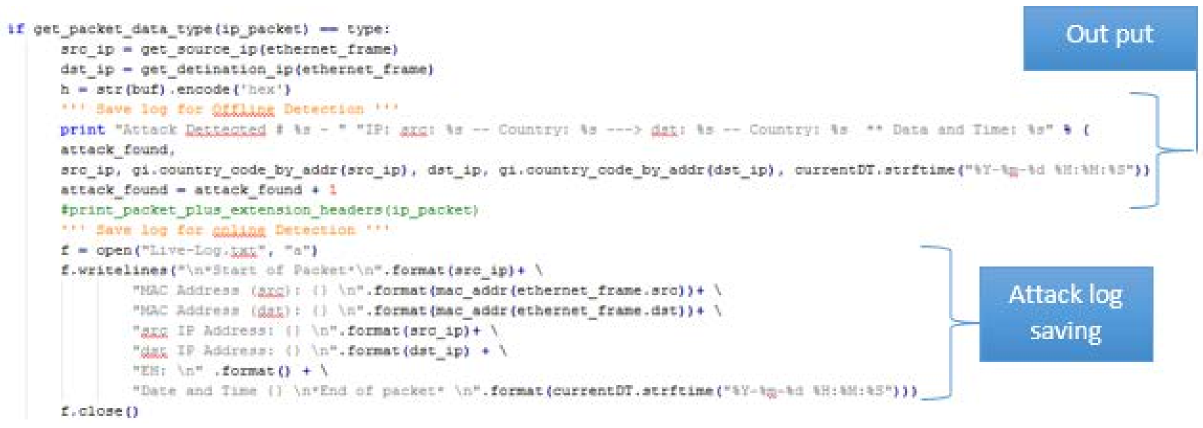


```
if packet_has_extention_hdrs(ip_packet):
    hdr_dic = ip_packet.extension_hdrs
    last_key = 0
    for key,values in hdr_dic.items():
        last_key = key
        #print hdr_dic
```

Figure 4.62: Detection Engine code – Part 2

Now it is time to start comparing the decoded packet and extracted data with some values. As code in Figure 4.63 shows, if the user chooses "type" option it will call this function and start to match the defined type with the packet data type. For example if the user chooses type 128, the detection engine will compare this value to the decoded and extracted data from the packet from the process shown in §4.3.2 and if any match is found the Output will generate an alarm with attacker info (source and destination IP address, country and data and time). Afterwards it will pass information into the Attack Logger process to save the attack log. In this part, a text file will be opened and the process will start to write the line. The log file contains:

- The Source and Destination MAC address of the Ethernet frame
- Source and Destination IP address of the packet
- Extension headers of the packet
- Date and time of when attack detected



```
if get_packet_data_type(ip_packet) == type:
    src_ip = get_source_ip(ethernet_frame)
    dst_ip = get_detination_ip(ethernet_frame)
    h = str(buf).encode('hex')
    ''' Save log for Offline Detection '''
    print "Attack Dettected # %s - " "IP: src: %s -- Country: %s ---> dst: %s -- Country: %s  ** Data and Time: %s" % (
    attack_found,
    src_ip, gi.country_code_by_addr(src_ip), dst_ip, gi.country_code_by_addr(dst_ip), currentDT.strftime("%Y-%m-%d %H:%M:%S"))
    attack_found = attack_found + 1
    #print_packet_plus_extension_headers(ip_packet)
    ''' Save log for online Detection '''
    f = open("Live-Log.txt", "a")
    f.writelines("\n*Start of Packet*\n".format(src_ip)+ \
            "MAC Address (src): {} \n".format(mac_addr(ethernet_frame.src))+ \
            "MAC Address (dst): {} \n".format(mac_addr(ethernet_frame.dst))+ \
            "src IP Address: {} \n".format(src_ip)+ \
            "dst IP Address: {} \n".format(dst_ip) + \
            "EH: \n" .format() + \
            "Date and Time {} \n*End of packet* \n".format(currentDT.strftime("%Y-%m-%d %H:%M:%S")))
    f.close()
```

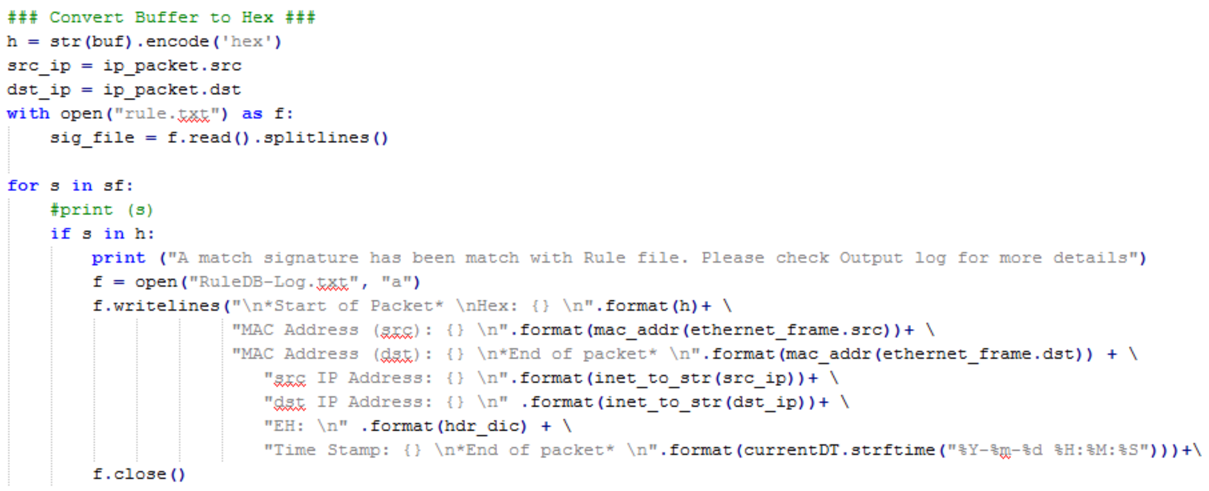
Figure 4.63: Detection Engine code – Part 3

Figure 4.64 shows the code for inspecting the packet with database rule. If the user has chosen to apply a rule database or file instead of type option, this part starts to work by getting the Ethernet packet data which is IPv6 header + data and will convert it to hex.

Once the conversion is done it will run for a loop looking for any match of the rule database file and the content of captured packet. Furthermore, if any match is found it will generate an alert and save the details into RuleDB-Log.txt

The log file will contain the following:

- Detected packet hex file
- The Source and Destination MAC address of the Ethernet frame
- Source and Destination IP address of the packet
- Extension headers of the packet
- Date and time of when attack detected

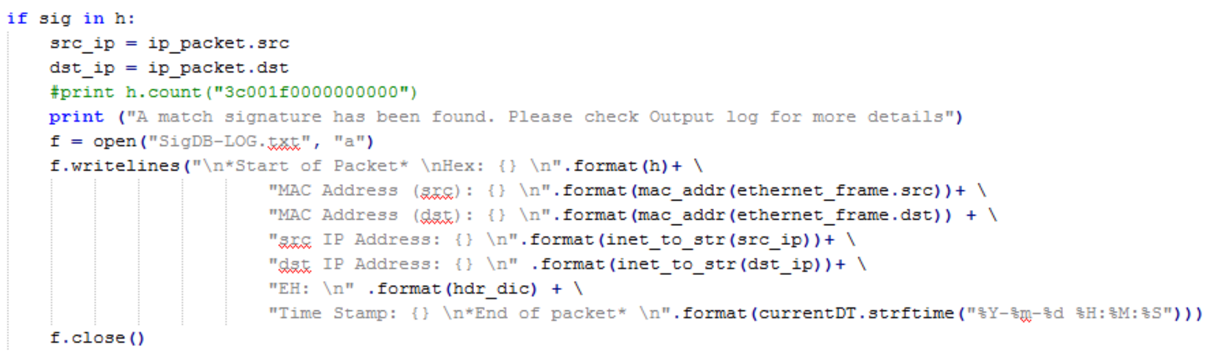


```
### Convert Buffer to Hex ###
h = str(buf).encode('hex')
src_ip = ip_packet.src
dst_ip = ip_packet.dst
with open("rule.txt") as f:
    sig_file = f.read().splitlines()

for s in sf:
    #print (s)
    if s in h:
        print ("A match signature has been match with Rule file. Please check Output log for more details")
        f = open("RuleDB-Log.txt", "a")
        f.writelines("\n*Start of Packet* \nHex: {} \n".format(h)+ \
                        "MAC Address (src): {} \n".format(mac_addr(ethernet_frame.src))+ \
                        "MAC Address (dst): {} \n*End of packet* \n".format(mac_addr(ethernet_frame.dst)) + \
                           "src IP Address: {} \n".format(inet_to_str(src_ip))+ \
                           "dst IP Address: {} \n" .format(inet_to_str(dst_ip))+ \
                           "EH: \n" .format(hdr_dic) + \
                           "Time Stamp: {} \n*End of packet* \n".format(currentDT.strftime("%Y-%m-%d %H:%M:%S")))+\
        f.close()
```

Figure 4.64: Detection Engine code – Part 4

Figure 4.65 shows the part if the user chooses the signature option. This only can only look for one signature at a time. In this option, the packet will be converted into hex from the Ethernet frame data buffer that was captured and decoded previously and if the defined signature was in the packet it will generate an alert and log the packet details as follows:

- Detected packet hex file
- The Source and Destination MAC address of the Ethernet frame
- Source and Destination IP address of the packet
- Extension headers of the packet
- Date and time of when attack detected



```
if sig in h:
    src_ip = ip_packet.src
    dst_ip = ip_packet.dst
    #print h.count("3c001f0000000000")
    print ("A match signature has been found. Please check Output log for more details")
    f = open("SigDB-LOG.txt", "a")
    f.writelines("\n*Start of Packet* \nHex: {} \n".format(h)+ \
                    "MAC Address (src): {} \n".format(mac_addr(ethernet_frame.src))+ \
                    "MAC Address (dst): {} \n".format(mac_addr(ethernet_frame.dst)) + \
                    "src IP Address: {} \n".format(inet_to_str(src_ip))+ \
                    "dst IP Address: {} \n" .format(inet_to_str(dst_ip))+ \
                    "EH: \n" .format(hdr_dic) + \
                    "Time Stamp: {} \n*End of packet* \n".format(currentDT.strftime("%Y-%m-%d %H:%M:%S")))
    f.close()
```

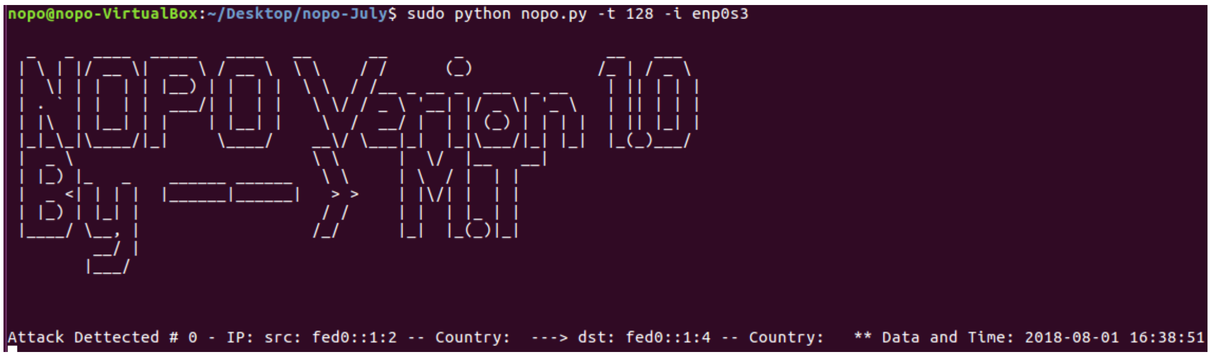
Figure 4.65: Detecting Engine code - part 5

4.3.4 Output

Depending on the decision made by the Detection Engine, the output will be an alert (Figure 4.66). If any packet is detected by our system, it will show the packet details (Source and Destination address with country of origin.) and save the packet into a PCAP file. The reason that the country is shown as Unknown is that we are using local addresses. As shown in Figure 4.66, NOPO runs on a network interface “enp0s3” and will be looking for any packet(s) that contains an “itype 128”. The log shows that a packet with an “itype 128” has been identified and has the following parameters:

- **src:** Attacker IP address which is here “fed0::1:2”
- **Country:** This will be read from an IPv6 Global Database for Country and City of an IPv6 address and regularly updated
- **Dst:** destination host IP address which here is “fed0::1:10”
- **Date and Time:** When attack has been detected.

Depending on user needs, the output log can be alter to meet their requirements. In addition a log file will be saved with all packet details as well.



```
nopo@nopo-VirtualBox:~/Desktop/nopo-July$ sudo python nopo.py -t 128 -i enp0s3

 _   _  ____  _____   ____  __      __          _             __   ___
| \ | |/ __ \|  __ \ / __ \ \ \    / /         (_)           /_ | / _ \
|  \| | |  | | |__) | |  | | \ \  / /__ _ __ _  ___  _ __    | || | | |
| . ` | |  | |  ___/| |  | |  \ \/ / _ \ '__| |/ _ \| '_ \   | || | | |
| |\  | |__| | |    | |__| |   \  /  __/ |  | | (_) | | | |  | || |_| |
|_| \_|\____/|_|     \____/   __\/ \___|_|  |_|\___/|_| |_|  |_(_)___/
|  _ \                        \ \      |  \/  |__   __|
| |_) |_   _   ______ ______   \ \     | \  / |  | |
|  _ <| | | | |______|______|   > >    | |\/| |  | |
| |_) | |_| |                  / /     | |  | |_ | |
|____/ \__, |                 /_/      |_|  |_(_)|_|
        __/ |
       |___/

Attack Dettected # 0 - IP: src: fed0::1:2 -- Country:  ---> dst: fed0::1:4 -- Country:   ** Data and Time: 2018-08-01 16:38:51
```

Figure 4.66: Alert

4.3.5 Simulation

Figure 4.67 shows the topology of network design. Virtual box is used to create a virtualised environment for our model and testing scenarios. Six different Operating Systems (OS) for our end hosts have been taken into account. As mentioned earlier each OS deals with IPv6 packet differently, some of them deal with IPv6 packet according to RFC’s, but the others may not. In addition, this will be depending on the packet(s) as well. For result validation and detection accuracy rate, two popular, up-to-date and open source Network Intrusion Detection Systems, Snort and SURICATA have been employed.

The following Operating Systems have been employed as an end-host which, as mentioned before each OS have different response and implantation for IPv6, therefore to test Evasion (End-host accept packet) and Insertion (End-host reject packet) we used different OS. However, in this thesis, we are not concerned about the OS implantation for IPv6. The OS that we used for our test simulation are:

- Windows 7
- Windows Server 2008
- Windows 8.1
- Debian 7.7
- Ubuntu 12.04
- Centos 6.5

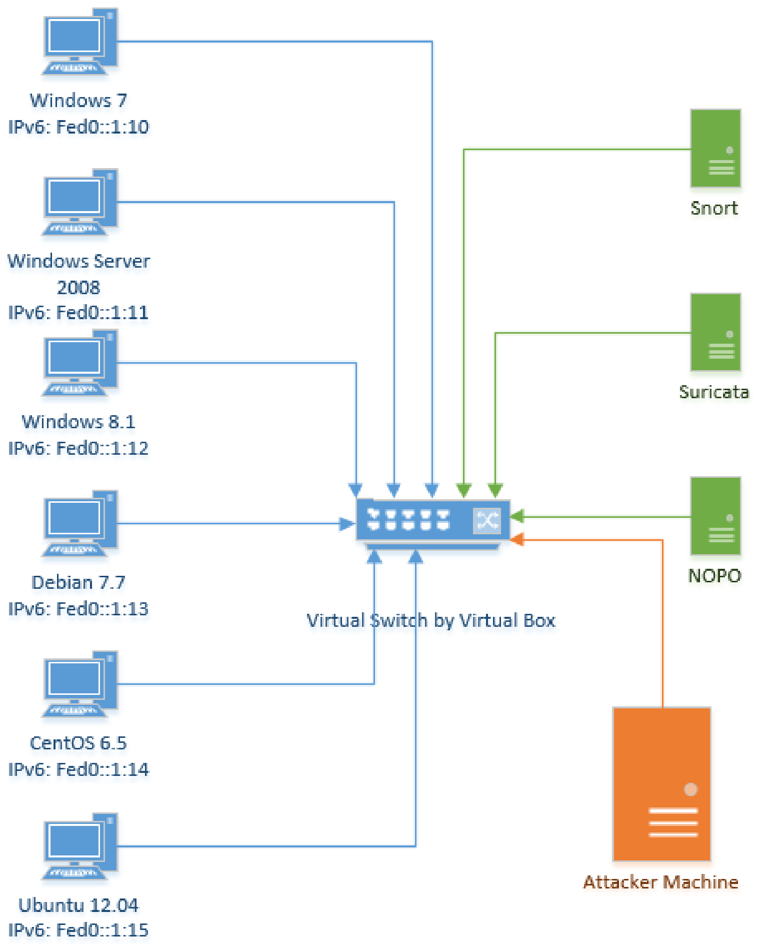


Figure 4.67: Simulation topology

We connected both Network Intrusion Detection Systems to the same virtual switch as the end hosts and the attacker machine has been connected to a virtual switch. Each NIDS network card is configured as promiscuous mode. Promiscuous mode is a security policy option that can be defined at the virtual switch or portgroup level in Virtual Machine setting. This mode allows portgroup to see all traffic traversing the virtual switch [130]. Each NIDS (Snort, Suricata and NOPO) has 2GB of RAM and 2 core of CPU.

For the development stage, we used Python as our programing language to implement our solution into a framework. We used Python because it is a high-level, interpreted and general purpose dynamic programing language that focuses on code readability and the syntax will help to do the coding in fewer steps comparing to C++, Java and etc.

4.3.5.1. Snort

Snort was originally developed by Martin Roesch in 1998 as a lightweight cross platform network sniffing tool. In 1999 a pattern matching system for Snort sniffer output had been created and then pre-processers protocol normalization to Snort engine were added which allows a single rule to be applied into any variation protocol. Snort is an open source Network Intrusion Detection and Prevention System that can be used as a real-time traffic analyser and packet logging on IP networks. Snort can detect attacks such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts and many more [53].

4.3.5.2. Suricata

Suricata was funded by the US Department of Homeland Security when it created an organisation called Open Information Security Foundation (OISF). Suricata was built by OISF as an alternative for Snort, but the developer of Suricata used Snort architecture. In other words Suricata is an upgraded version of Snort [115].

4.3.5.3. Framework usage

Table 4.9 shows the command usage for NOPO. To use the framework the user must use the command as following:

Live detection command usage

- Run NOPO on Interface with packet type: python nopo.py -i enp0s3 -t 128
- Run NOPO on Interface with Signature: python nopo.py -i enp0s3 -sf “signature”
- Run NOPO on Interface with Rule file: python nopo.py -i enp0s3 -sig “packet to rule file”

Analysing a PCAPfile:

- Run NOPO on Interface with packet type: python nopo.py -o “path to pcap file” -t 128
- Run NOPO on Interface with packet type: python nopo.py -o “path to pcap file” -sf “signature”
- Run NOPO on Interface with packet type: python nopo.py -o “path to pcap file” -sig “packet to rule

Table 4.9: NOPO Usage

Option name	**Command**	**Usage**
Interface Selection	-i	To choose live interface to traffic sniffing
Read from file	-o	Read from saved pcap file.
Type	-t	If chosen, the framework will look into all packets type for a match
Signature	-sf	Looking for an specific signature
Rule file	-sig	Trying to match the rules against traffic
Help	-h	Showing help menu

By using the proposed framework, we were able to detect Evasion, Insertion and DoS attack when using fragmentation and extension headers. By finalizing the development of the presented solution into a framework, we are going to test the NOPO framework against different packets and methods and compare the detection results with Snort and Suricata.

4.4 Summary

This chapter has presented a detailed overview of the current issues with IPv6 packet that attackers can take advantage of to evade detection methods. This chapter presented a real-world example of how attackers can use IPv6 Extension Headers to bypass detection and performed a simple Operating System finger print attack to identify the live host on the network.

This chapter presented a solution to detect Evasion, Insertion and DoS attack and provide an improved version of packet decoder. The presented framework was developed specifically to detect IPv6 Evasion, Insertion and DoS methods (IDS evasion, Firewall evasion, OS fingerprint, Network Mapping, DoS/DDoS attack and Remote code execution attack) that use IPv6 Extension Headers and Fragmentation to hide their activity and bypass detection. The theoretical part of the aims has been achieved successfully using the presented methodology. In this regard, we have started by presenting the structure of how the packet needs to be captured. Then, a code was presented to be used for sniffing packets on a live interface while being read from a PCAP file. We have also presented how the proposed solution decodes the packet and extracts the necessary information from the packet. Decoding the packet in a correct format is one of the objectives of this research, which has been achieved successfully, as well as others. The structure of the Detection Engine and how the Detection Engine works to detect the malicious activity based on user input has been presented, and explained how this processes making decision and analysing packet that received from previous processes (Packet Capturing and Packet Decoder) and showing the results of packet inspection. At the end of this chapter, a simulation framework was presented. This part will explain what tools and programing language was used to develop the proposed framework and explain the command usage of the NOPO framework.

Chapter 5

Tests and Results

5.1 Introduction

This chapter provides an evaluation of the proposed solution by performing some different tests. Each test has been categorised based on similarity of the crafted packets. These tests examine the detection accuracy of the proposed solution along with two standard open source, free and wildly used Network Intrusion Detection Systems (NIDSs). At the end of the tests, by comparing the results, we have validated that the proposed solution in this thesis achieved the main aim of this thesis, which is to develop a framework that can detect Evasion, Insertion and DoS attacked based on IPv6 network while using Extension Headers and/or IPv6 Fragmentation.

After the successful development of the proposed solution in chapter 4 and providing the details of how the proposed framework will work and the details of the test bed topology, in this chapter a set of tests is used to test the detection accuracy of two up-to-date Network Intrusion Detection Systems (NIDS's). Those NIDSs are Snort and Suricata and the detection accuracy results of these two NIDS will be compared to the results of the proposed framework, NOPO.

As explained earlier, the scenario was implemented, which is shown in Figure 4.67. In this scenario, there are six end-hosts and each end-host has a different Operating System.

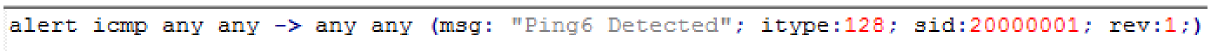
Figure 5.68 shows one of our tests that bypassed both IDS's and an end host with Ubuntu Operating System replying back to the ping Echo-request.

ICMPv6 (Internet Control Message Protocol version 6) was used for those tests and specifically the Echo Request type (128) of ICMPv6. For the tests purpose, ICMPv6 has been taken into consideration as it is simple to implement, test, and get response. TCP does not require a three-way handshaking. In addition, it will do ECHO Reply with the payload of the ECHO Request (Figure 5.68 and Figure 5.69). By using such a method, attackers can easily identify the fragmentation reassembly policy of the target host, therefore they can construct more different packets to bypass detection and get the target host to resemble the packets.

Figure 5.68: Echo request with Payload

Figure 5.69: Echo reply with Payload

The following rule (Figure 5.70) was added to each NIDS to trigger the alert if the rule matched with any incoming packet.



```
alert icmp any any -> any any (msg: "Ping6 Detected"; itype:128; sid:20000001; rev:1;)
```

Figure 5.70: Test bed detection rule

- **Alert**: generate an alert when rule matched
- **icmp**: means the rule will apply to “icmp” packet
- **any**: used for source IP address .In our case it shows that the rule will be applied to all packets.
- **any**: used for the source port number. Since port numbers are irrelevant at the IP layer, the rule will be applied to all packets.
- **->**: sign showing the direction of the packet. In our example it will be applied to any incoming packet
- **any**: used for destination IP address and shows that the rule will be applied to all packets.
- **any**: used for the destination port number. Since port numbers are irrelevant at the IP layer, the rule will be applied to all packets.
- **msg**: if any match found the log will contain the message
- **itype**: the ICMP type number
- **sid**: stands for Snort ID and it uniquely identifies Snort rules

- **rev**: stands for "Revision" is used to uniquely identify revisions of Snort rules

This rule will generate an alarm if any of the incoming packets have an "itype 128". ICMPv6 Echo Request has been chosen for the test bed as it is simple to implement, test, and get response and like TCP does not require a three-way handshaking and is not complicated for implementation.

5.1.1 Hop-By-Hop Extension Headers

The Hop-by-Hop Options header is used to carry optional information that must be examined by every node along a packet's delivery path. As shown in Table 5.10, three different packet structures are used. Each packet contain 1, 2 and 118 Hop-by-Hop extension headers. Table 5.10 shows the packet that is sent to the end host that does not have any "option" in the Hop-by-Hop option extension header. However, because of recommendation from RFC 2460, each packet can have only one Hop-by-Hop extension header and this header must appear immediately after the IPv6 Header. Therefore, this mean if we have a packet with more than one Hop-by-Hop extension header the destination host will not accept the packet. Here the test is to see if either of the NIDS is able to detect the ICMPv6 request or not. As shown in Figure 5.71 the packets have the following structure:

- Ethernet frame – The whole packet
- Ethernet frame data – which is IPv6 header + data
- IPv6 Header: with Next Header value = 0 + Hop-by-Hop extension header with next value = 0 + "N" Hop-by-Hop extension header with next value = 58 + data, which is ICMPv6 Echo Request here.

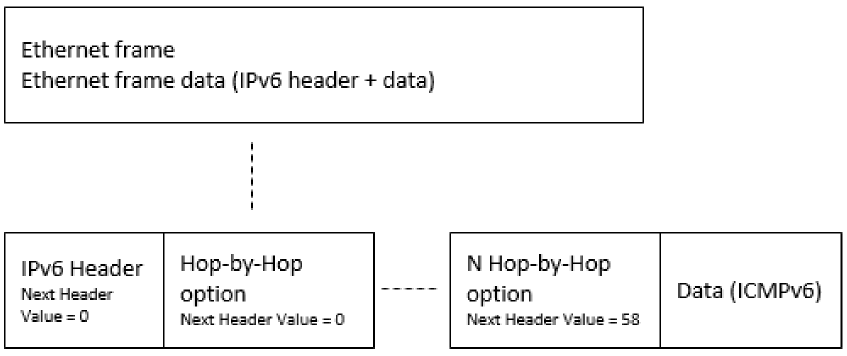


Figure 5.71: Illustrated overview of crafted packet.

As the results show in Table 5.10, Snort was not able to detect a packet with 118 Hop-by-Hob extension headers. All packets contained an ICMPv6 Echo Request as data and all NIDS have a rule, which is shown in Figure 5.70. Why Snort is not able to detect test number 3 is that, it cannot decode the packets correctly and match the packet data with its rule.

Table 5.10: Tests with Hop-By-Hop Extension Headers

Test Number	Test method	Snort	Suricata	NOPO
1	hop-by-hop ignore option	1	1	1
2	2 hop-by-hop headers	1	1	1
3	118 hop-by-hop headers	0	1	1

5.1.2 Distension Option Extension Headers

The Destination Option Extension Header is used to carry optional information that needs to be examined only by the end-host. As has been shown in Table 5.11, five different packets were used to test the evasion and insertion. For this test bed, each packet will have a different number of Destination Option extension headers. As has been shown in Table 5.11 Snort was able to detect the ICMPv6 Echo Request when packets have two Destination Option Extension Headers.

Table 5.11: Tests with Distension Option Extension Headers

Test Number	Test method	Snort	Suricata	NOPO
4	destination ignore option	1	1	1
5	2 destination headers	1	1	1
6	119 destination headers	0	1	1
7	1329 destination headers(with 12 fragmentation header)	0	1	1
8	5427 destination headers (with 46 fragmentation header)	0	0	1

By further investigation and referring to the results, we found that Snort could not decode any packet with more than eight extension headers and Suricata could not decode any packet with more than 5,422 extension headers. We generated a packet with 1,329 Destination Option Extension Headers that fragmented into 12 fragmentations and another packet with 5,427 Destination Option Extension Headers that fragmented into 46 fragmentations. Figure 5.72

shows the illustrated packet overview (test 7 and test 8) of the packet structure that has been used in this section. As shown in the figure the Ethernet frame (number 1), has the data, which is the IPv6 header + data. Packet number 2 shows the IPv6 header with some of the extension headers and the fragmentable part.

In number 3, packet has been fragmented and it has the IPv6 header with next header value of 44 which is the identification number of Fragment Extension Header then the next part is the Fragment Extension Header with next header value of 60 and more fragment packet set to 1 which means more fragment is coming. The next part in packet number 3 is the Destination Option Header with next header value of 60, which is the identification of the extension header. Packet 4 is the last fragmented part and the next header value of the Destination Option Header is 58 which, is the identification number of ICMPv6 header and data is the ICMPv6 Echo Request (itype 128). All end-hosts replied to the ping request. That means all of them accepted the packet as a valid packet.

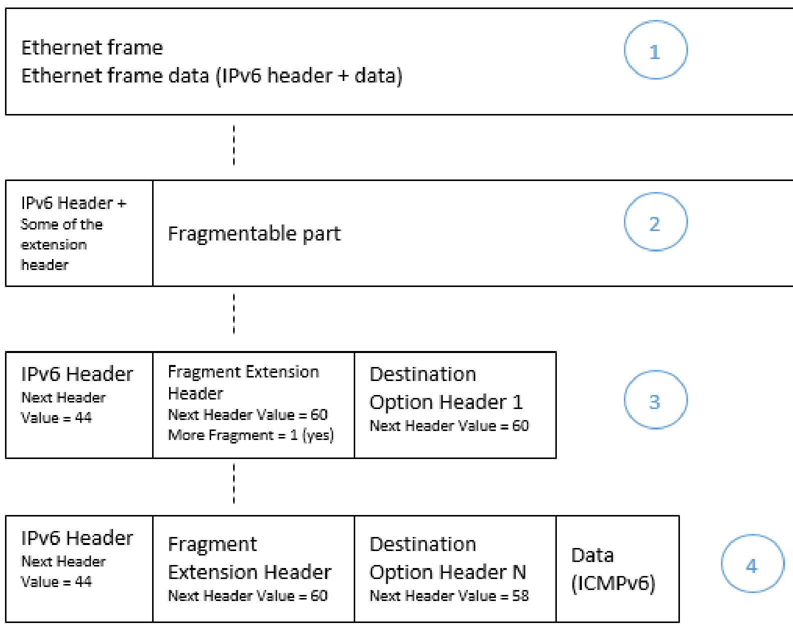


Figure 5.72: Illustrated overview of fragmented packet with Destination Option Header

5.1.3 Fragmentation Extension Headers

By using Fragmentation Extension Headers, a set of overlapping packets has been created. The results in Table 5.12 show that Snort is not capable of detecting and overlapping fragmented packets. As mentioned before, by future tests we found that both NIDS have limitations on the packet decoding part. Overlapping other packets is the set of fragmented packets where one fragment will overlap another. For example in test 11 Snort and Suricata were not able to detect the test. The test 11 structure is as follows:

- pkt1 and pkt2 and pk3
- pkt3 overlaps pkt1
- pkt3 overlaps partially pkt2 – the overlapped part will be the IPv6 header and Extension header and the data will be the pkt2 data.

Table 5.12: Tests result wit Fragmentation Extension Headers

Test Number	Test method	Snort	Suricata	NOPO
9	correct fragmentation	1	1	1
10	one-shot fragmentation	1	1	1
11	overlap-first-zero fragmentation	0	0	1
12	overlap-last-zero fragmentation	0	1	1
13	overlap-first-dst fragmentation	0	1	1

5.1.4 Routing Extension Header

Table 5.13 shows the results of tests done with the Routing Extension Header. By using such a header and ICMPv6 error message, an attacker can perform a DoS attack by sending error-causing Source Routing Headers in back-to-back datagrams. However due to the same problem, which is decoding packets correctly, Snort was not able to detect two tests out of four. Up to now, Snort performed weak packet decoding according to the results. In addition, Suricata had a better detection accuracy. As the proposed framework packet decoder was designed and developed accurately, it has detected all the tests up to now.

Table 5.13: Tests with Routing Extension Header

Test Number	Test method	Snort	Suricata	NOPO
14	source-routing (done)	1	1	1
15	source-routing (todo)	0	1	1
16	unauth mobile source-route	0	1	1
17	mobile+source-routing (done)	1	1	1

5.1.5 Packets with Authentication Header (AH) Extension Header and multicast

Results in Table 5.14 show two tests, one with Authentication Header and the other one with ping6 request from local mulicast address (ff02::1). The Authentication header is used to provide connectionless integrity and data origin authentication for IP datagrams, and to provide protection against replays. This extension header is similar in format and used for the IPv4 authentication header defined in RFC2402 [79]. Both NIDS detect the packet, which has an ICMPv6 Echo request.

Table 5.14: Packets with mixture of Extension Headers

Test Number	Test method	Snort	Suricata	NOPO
18	ping6 with a zero AH extension header	1	1	1
19	ping from multicast (local!)	1	1	1

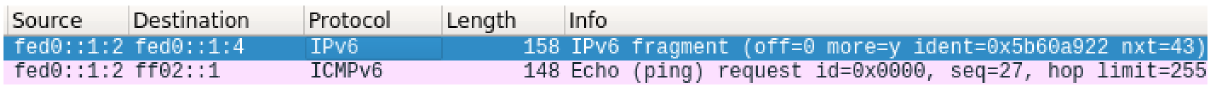
5.1.6 Fragmented packets with Routing Extension Headers

Table 5.15 shows the result for using fragmentation to send a packet with Routing Extension Header to link local (end-host), to multicast address (ff02::1) and from multicast address to end-host. The result shows Snort unable to detect test 20 and 21 due to the same problem as previously discussed.

Table 5.15: Fragmented packets with Routing Extension Headers

Test Number	Test method	Snort	Suricata	NOPO
20	frag+source-route to link local	0	1	1
21	frag+source-route to multicast	0	1	1
22	frag+srcroute from link local (local!)	1	1	1
23	frag+srcroute from multicast (local!)	1	1	1

As a proof, Figure 5.73 shows test 21 packet which was captured by Wireshark. As it is visible, the destination of the ICMPv6 Echo Request is multicast address "ff02::1".



```
Source     Destination   Protocol   Length   Info
fed0::1:2  fed0::1:4     IPv6                158 IPv6 fragment (off=0 more=y ident=0x5b60a922 nxt=43)
fed0::1:2  ff02::1       ICMPv6              148 Echo (ping) request id=0x0000, seq=27, hop limit=255
```

Figure 5.73: frag+source-route to multicast packet

5.1.7 Packets with Jumbo Option size and error option in Extension Headers

As mentioned before, IPv6 header has a Payload Length field, which is 16bit, therefore it can support payloads up to 65,535 octet. Jumbo Payload option is an option for IPv6 Hop-by-Hop extension headers and designed to carry a 32-bit length in order to allow transmission of IPv6 packet with payloads between 65,536 and 4,294,967,295 octets. As Table 5.16 test results show, both NIDS were able to detect three packets.

Table 5.16: Packets with Jumbo Option size and error option

Test Number	Test method	Snort	Suricata	NOPO
24	jumbo option size < 64k	1	1	1
25	jumbo option size < 64k, length 0	0	0	1
26	jumbo option size < 64k, length 167	0	0	1
27	error option in hop-by-hop	1	1	1
28	error option in dsthdr	1	1	1

However, as test 25 and 26 results show, Snort and Suricata failed to detect the attack. In test 25, the IPv6 header payload length is zero (Figure 5.74) and in test 26, the IPv6 header payload length is 167 (Figure 5.75). As discussed earlier, because Snort and Suricata could not decode packets and extract packet data, therefore they cannot decode such packets (test 25 and 25). As results show, the proposed framework detected all attacks.

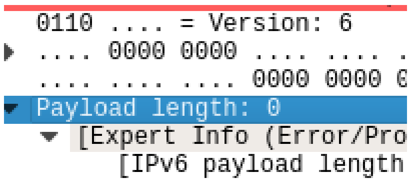

```
 0110 .... = Version: 6
 .... 0000 0000 .... .... .
 .... .... .... 0000 0000 6
 Payload length: 0
   [Expert Info (Error/Pro
     [IPv6 payload length
```

Figure 5.74: jumbo option size < 64k, length 0

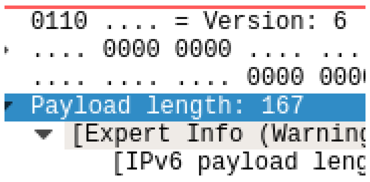

```
 0110 .... = Version: 6
 .... 0000 0000 .... ...
 .... .... .... 0000 000
 Payload length: 167
   [Expert Info (Warning
     [IPv6 payload leng
```

Figure 5.75: jumbo option size < 64k, length 167

5.1.8 Packets with large and small length field and ping6

In this series of tests carried out and as Table 5.17 results show, Snort had the lower detection rate compared with Suricata. The IPv6 header in test 29 packet had the payload length = 0, the IPv6 header in test 30 packet had the payload length = 258 and the IPv6 header in test 30 packet had the payload length = 60. When IPv6 packet payload length was set to 0 and 258, both Snort and Suricata bypassed. The reason behind that is as mentioned before, the decoder could not decode the packets, as they were invalid to them, therefore it could not extract the packet content and match it with a rule.

In test 35, the packet had the following structure:

- pkt1 – contain ICMPv6 Echo Request
- pkt3 – will overlap the header of pkt1
- pkt3 header + pkt1 data (ICMPv6 Echo Request)

Therefore pkt2 is a missed fragmented packet which was needed to complete the reassembly process. As the results show, Snort and Suricata could not detect the ICMPv6 request because of the missed fragmented packet. Therefor an Insertion attacked happened where the packet looked invalid to both NIDS, the proposed framework detected all tests. This is because of the way that we handle the packets and decode them extracting the necessary information out of the packets.

Table 5.17: packets with large and small length field and ping6:

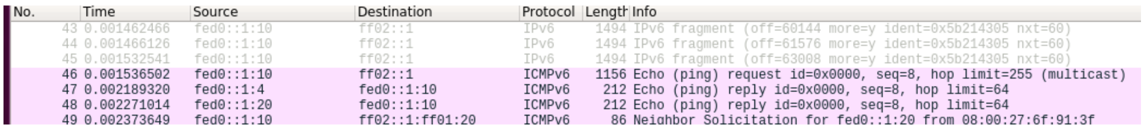
Test Number	Test method	Snort	Suricata	NOPO
29	0 length field	0	0	1
30	too large length field	0	0	1
31	too small length field	0	1	1
32	ping6 with bad checksum	0	1	1
33	ping6 with zero checksum	0	1	1
34	ping with hop count 0	1	1	1
35	fragment missing	0	0	1

5.1.9 ICMPv6 Amplification

As discussed earlier, Broadcast Amplification (Smurf) attack is one of the most common attacks for both versions of IPs. Attackers use this type of attack to generate a storm of traffic. Attackers use Smurf attack to launch a DoS attack by sending an ECHO request packet to a multicast address with spoofed source address of victim machine. Once all nodes of the targeted multicast address have received a packet, all nodes start to reply to the source (Figure 5.76), which is the victim and flood it with a large number of ECHO replies. The detection of Smurf attack in IPv6 is much harder than IPv4. Because in IPv6 the attacker can use the mixture of extension headers and fragmentation to bypass the detection.

As shown in Figure 5.76, we sent the packet to multicast address (ff02:1) from spoofed targeting host address (fed0::1:10). Once the packet was received as shown in Figure 5.76

frame 47 and 48, the other two hosts on the network replied back to the targeted host. By resending the packet a couple more times, we can bring both network and targeted hosts down.



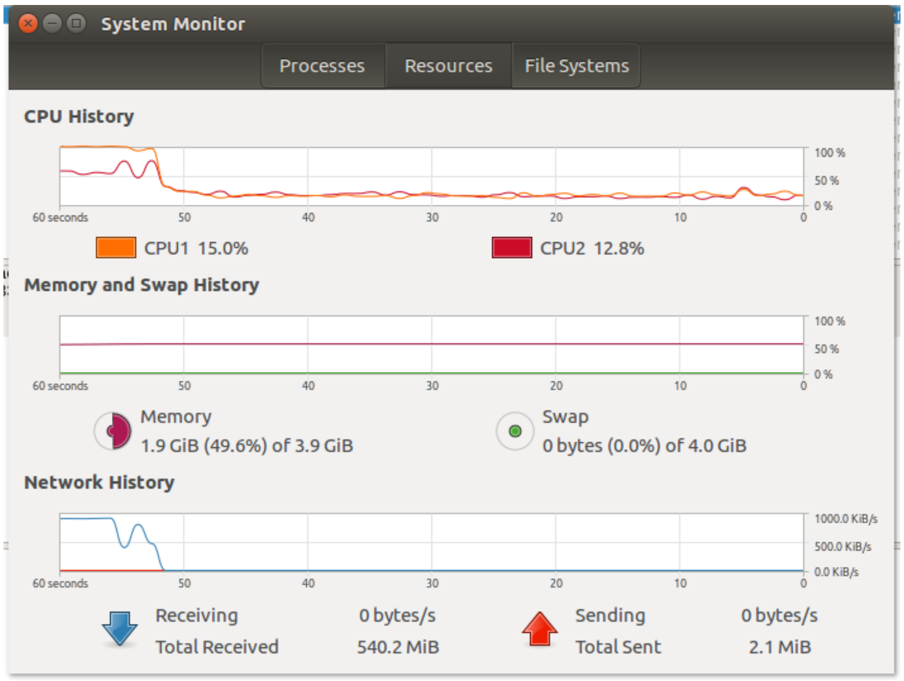
```
No.    Time          Source        Destination      Protocol  Length Info
    43 0.001462466   fed0::1:10    ff02::1          IPv6      1494 IPv6 fragment (off=60144 more=y ident=0x5b214305 nxt=60)
    44 0.001466126   fed0::1:10    ff02::1          IPv6      1494 IPv6 fragment (off=61576 more=y ident=0x5b214305 nxt=60)
    45 0.001532541   fed0::1:10    ff02::1          IPv6      1494 IPv6 fragment (off=63008 more=y ident=0x5b214305 nxt=60)
    46 0.001536502   fed0::1:10    ff02::1          ICMPv6    1156 Echo (ping) request id=0x0000, seq=8, hop limit=255 (multicast)
    47 0.002189320   fed0::1:4     fed0::1:10       ICMPv6     212 Echo (ping) reply id=0x0000, seq=8, hop limit=64
    48 0.002271014   fed0::1:20    fed0::1:10       ICMPv6     212 Echo (ping) reply id=0x0000, seq=8, hop limit=64
    49 0.002373649   fed0::1:10    ff02::1:ff01:20  ICMPv6      86 Neighbor Solicitation for fed0::1:20 from 08:00:27:6f:91:3f
```

Figure 5.76: Smurf attack packet

5.1.10 DoS Attack

In this test, a crafted packet was used to perform the DoS attack. Figure 5.77 shows the target host resources before performing the DoS attack. In this DoS attack, we have sent a ping request in fragment extension headers with hop-by-hop and a router alert message (in our test environment there are no routers so this part does not affect anything in our scenario). A few seconds later as is shown in Figure 5.78 the CPU usage went to 99% and the target host OS was frozen until the packet sending stopped. To detect this attack we placed Snort, Suricata and NOPO on the same network and launched the test.

Due to the limitation of both Snort and Suricata, they are not capable of detecting attack in this case, while our presented technique could overcome this detected this DoS attack.



CPU1 15.0%
CPU2 12.8%

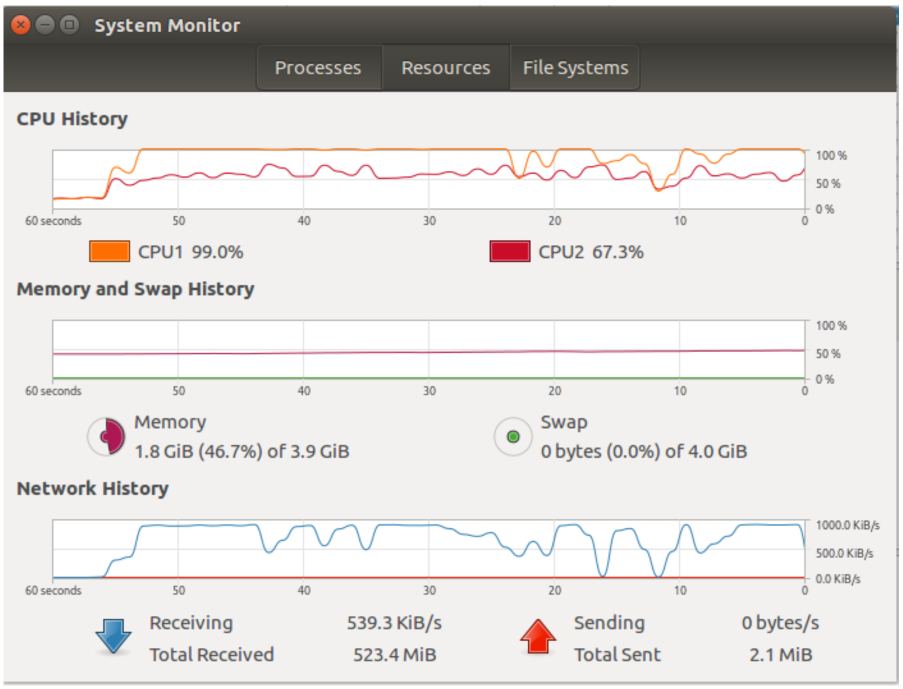
Memory
1.9 GiB (49.6%) of 3.9 GiB

Swap
0 bytes (0.0%) of 4.0 GiB

Receiving 0 bytes/s
Total Received 540.2 MiB

Sending 0 bytes/s
Total Sent 2.1 MiB

Figure 5.77: Target system before attack



CPU1 99.0%
CPU2 67.3%

Memory
1.8 GiB (46.7%) of 3.9 GiB
Swap
0 bytes (0.0%) of 4.0 GiB

Receiving 539.3 KiB/s
Total Received 523.4 MiB
Sending 0 bytes/s
Total Sent 2.1 MiB

Figure 5.78: Target System after attack

5.1.11 Results

A set of different tests and methods (Evasion, Insertion and DoS) were performed to assess the detection accuracy of two existing NIDS (Snort and Suricata) and evaluate the proposed solution.

By comparing the results from NIDSs in Figure 5.79 Snort only detected 41% and SURICATA detected 70%. These results show that SURICATA detection accuracy is much better than Snort, but in the mixture of tests cannot detect 100% of the attacks.

This means there is a need to have to a solution that can detect such methods and ways and cover this gap by detecting all attacks that are used plus any other methods that are using the same techniques to bypass IDSs. NOPO results reveal that the proposed solutions are able to detect all those tests and methods that are used to evade the other two NIDSs. One of the aims of this thesis was that the proposed solution should have a better packet decoding performance.

By referring to the results, it is obvious that the proposed solution had a much better performance in packet decoding compared to Snort and Suricata. In addition, it proves that the proposed framework has a better rate of detecting an Evasion, Insertion and DoS attack when using IPv6 extension headers and/or fragmentation.

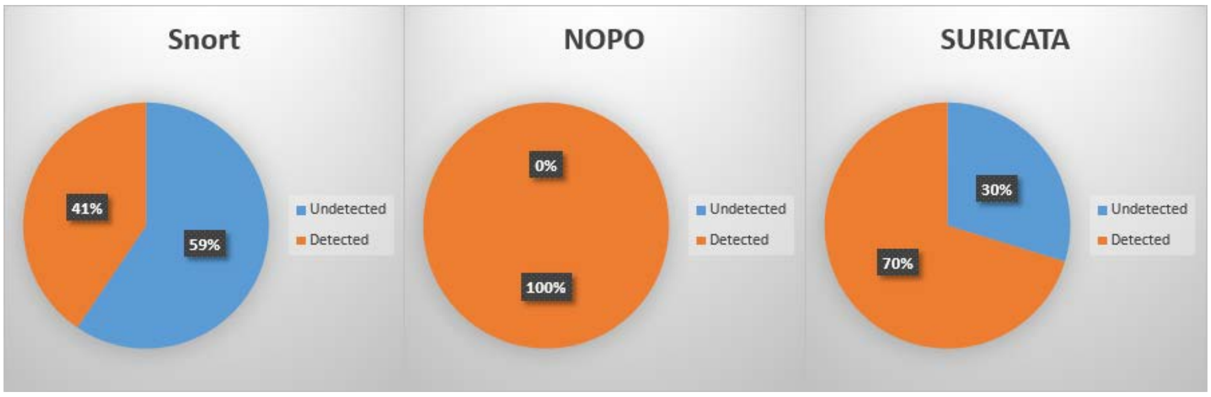


Figure 5.79: Results comparison charts

5.2 Summary

This chapter presented different methods of tests that were performed to examine the detection accuracy of two Open Source and free Network Intrusion Detection Systems (NIDSs) and validate the proposed framework with the same tests and methods preformed for the other two NIDS.

The results presented in this chapter outlined that attackers can use the IPv6 Extension Headers and Fragmentation techniques to bypass detection methods and perform a successful evasion. However due to the design structure of IPv6, we show that by using an integration of existing techniques and running different types of attack such as Evasion, Insertion, Fragmentation, DoS, Smurf and ICMPv6 Amplification on IPv6 network, an attacker can bypass detection.

The results show that the proposed framework detection accuracy based on the methods that were used in this research is 100% accuracy. This means the proposed solution achieved the most important aim of this thesis, which is to develop a framework that can handle and detect evasion and insertion on IPv6 network by using Extension Headers and IPv6 Fragmentation.

The other two aims of this research were to improve packet decoding performance and improving the detection accuracy of the Evasion, Insertion and DoS attack where the results prove both of these aims have been achieved. In addition, the proposed framework is able to decode the packet correctly and extract necessary information from the packets.

A good example of an improved packet decoder could be the test 30 in Table 5.17 where Snort and Suricata were unable to decode the packet and detect the attack, because the IPv6 header payload length was 256 (too large) and therefore they could not decode the packet and match the packet content with the rule defined in Figure 5.70. Further more the proposed framework can be used to detect DoS and ICMPv6 Amplification attacks when using IPv6 extension headers or fragmentation. The code that used for test bed has been placed in Appendix 1.

Chapter 6

Conclusion and Future Work

6.1 Overall Conclusion

By conducting a literature review to find out the current Internet Protocol (IP) vulnerabilities and limitations of existing detection methods and solutions, we found that most attack techniques are common between IPv4 and IPv6 (such as Denial of Service (DoS)/Distributed Denial of Service (DDoS), Operating System (OS) Fingerprint, IP Fragmentation, Time to Live (TTL) Evasion and Source routing attack).

IPv6 introduces new features and capabilities. These result in new issues, and security issues is one of the most important of them. Most of the vulnerabilities are common between IPv4 and IPv6 and because of the changes that were made in the IPv6 implantation, arise additional vulnerabilities as well. There are many features, which are new and unique to IPv6. One of them is the improved support of headers (extensions and options) which were not existing before in IPv4.

A successful evasion attack can lead to IDS evasion, Firewall evasion, OS fingerprint, Network Mapping, DoS/DDoS attack and Remote code execution attack. In addition, we listed some of the most commonly used attacking tools that are available for both IPv4 and IPv6. By using such attacks, an attacker can easily identify the target system OS, discovering the live host on the network and even the version of the OS and this may reveal missed installations of security patches on the target host machine. Here is when Intrusion Detection System takes its place to detect such attacks and methods. However, by using the Extension Headers in IPv6, attackers can easily bypass existing solutions and perform a successful attack. Therefore, an attacker can easily gain control on a victim host machine or launch attacks to disturb the target host network or machine.

The gaps of current methods and solutions are, limited research for IPv6 evasion attacks, limited in-depth research for IPv6 security challenges, limited research for IPv6 evasion detection methods, limited number of Network Intrusion Detection System (NIDS) for IPv6 and limited number of IPv6 packet decoders. Based on our findings, we implemented two open source NIDS to test their detection accuracy and detection mechanism. As shown in Chapter (§)5 both up-to-date NIDS can be evaded using various IPv6 packets.

By reviewing existing solutions and methods for detecting IPv6 attack, we proposed a framework that can detect IPv6 evasion, insertion, ICMPv6 Amplification and DoS when using IPv6 Extension Headers and fragmentation. A common basis of our detection system is an ability to capture and decode the packets correctly and then extract the necessary information. The proposed framework relies on decoding packets in a correct format.

The proposed framework has four layers. The first layer captures the network traffic and passes it to the second layer for packet decoding. The decoding part is the most important part of the detection process. It is because if NIDS could not decode and extract the packet content, it would not be able to pass the correct information on to the Detection Engine process for detection. Once a packet has been decoded by the decoding process, it sends the decoded packet to the third layer which is the brain of proposed solution to make a decision by comparing the information with the defined value to see whether the packet is a threat or not, this layer is called Detection Engine. Once the packet(s) has been examined by detection processes, the result will be sent to the output layer. If the packet matches with a type or signature that system admin chose, it raises an alarm and automatically logs all details of the packet and saves for system admin for further investigation.

In addition, the proposed method is a dynamic framework that can work with different inputs, which can be defined by system-admin. These inputs can be packet type or a signature of a specific attack. In other words, the proposed framework can easily extract the packet content and try to match it with the value or rules that are given to it.

We implemented the proposed theoretical solution into a proposed framework for evolution tests. To develop the framework, we used Python language and “dpkt” module to capture and decode the packet. During the development phase, we found a critical bug in “dpkt” module where it would not decode and read the packet header correctly. We patched the bug and reported to the developers and they updated on their next version. Once the development phase finished the implementation part was started.

For the implementation part, we created a lab environment with a set of different Virtual Machines (VMs) and Operating Systems. All VMs were connected locally without any connection to the university network. We performed the same test as we had done for the other two NIDS. As results show in § 5.1.11, NOPO detected all methods that were used for test beds. In addition, NOPO was able to detect any ICMPv6 Amplification and Smurf attack which attackers could launch and evade detection by using the methods.

This thesis demonstrates that due to the stochastic nature of an IPv6 packet, when a decoding process is properly modelled and developed, it can be used for identifying certain malicious network activities with high detection accuracy rate.

By referring to the results in this study, we successfully achieved all aims and objectives, demonstrated in § 1.3.

The first objective of this research was to identify the limitations of existing solutions. This objective was achieved by reviewing existing solutions and related works. The outcome of this objective is shown in a table in §3.5.

The next objective was to investigate IPv6 evasion, detection methods and attack techniques. This objective was achieved by background research about Internet Protocols, IPv6 structure, Network Intrusion Detection System techniques and methodology, IP vulnerabilities and discussing some of the most common attacking tools to perform an attack in Chapter 2.

The third objective was to improve packet decoding. This objective was achieved by providing a patch for one of the open source packet parser and decoder systems and the solution was presented in § 4.3.2.

The forth objective was achieved by the proposed solution and developed framework. Therefore the test results of the proposed framework proved that we successfully achieved our fifth objective, which was to detect some different evasion methods, which are using Fragmentation and Extension Headers to evade Network Intrusion Detection System.

The last objective was achieved by the proposed solution in § 4.3.1 that shows we were able to detect malicious activities both on live traffic and by analysing the captured traffic.

In addition, by referring to the objectives of this thesis and the results of the proposed framework we can show that all of our aims are achieved. The first aim was to develop a network-based solution to detect IPv6 evasion. This aim was achieved by a solution that was presented in § 3.5.

The other aims of this research and thesis were to improve detection of Evasion, Insertion and DoS based on IPv6 when using fragmentation and/or extension headers and detect DoS and ICMPv6 Amplification attack when using such methods. The results presented in § 5.1.11 prove that the presented framework had an absolutely better detection accuracy for such attacks when compared with the detection rate of the other two NIDS, Snort and Suricata .

The third aim for this thesis was to improve the packet decoding performance. This aim was achieved by an improvement of the existing module named dpkt and the presented framework was demonstrated in § 4.3.2. The standard data unpack function included in the module follows extension headers, which means following its parsing, one has no access to all the extension headers in their original order. By defining, a new fields and adding each header to it before it is moved along, we will have access to all the extension headers while keeping the original parse speed of the framework virtually untouched at the same time. In addition, the extra memory footprint from this is also insignificant as it will be a linear fraction of the size of the whole set of packets. The results in § 5.1.11 confirming that the improved version of dpkt and the way that it was integrated into the presented framework, worked and we improved the performance for packet decoding comparing with the other two NIDS.

6.2 Contributions

This thesis makes the following contributions to the field of Network Intrusion Detection Systems to overcome the existing limitations:

1- Proposed a new framework that can detect evasion attacks in IPv6 network in real time traffic and offline traffic.
2- Detecting 37 different evasion methods and techniques. The test results in § 5.1.11 show the proposed solution is able to detect all those methods where the other two IDSs are unable to detect all those methods.
3- Improved a python module that is used in our framework for parsing, capturing and decoding packets. During the development stage, we found a bug in that module which

would not let the user read and decode the IPv6 packet header and extension header correctly.

4- Proposed a system that is able to detect DoS/DDoS attack (ICMPv6 flooding, Amplification and Smurf attack) and OS Fingerprint method.

6.3 Future Work

It is impossible to design a bulletproof solution to detect all techniques and methods used by intruders, as the saying goes, “Police always one step behind the criminals”. The proposed solution has some limitations of course. These limitations can be categorised as follows:

- Not able to detect evasion on User Datagram Protocol (UDP) and Transmission Control Protocol (TCP)
- No classification of attack types

This research can be expanded into two directions, first the technical part such as another decoding packet in other protocols such as TCP or UDP and the second on extending the options such as an attack classifier that can classify the attacks. However, for future work we can suggest the following improvements:

- NOPO could be enhanced by an improvement in the detection process to detect TCP and UDP based attack. This would help against an attack such as port scan.
- NOPO could be an enhanced Database for attack log and attack classifier.
- All experiments undertaken in this thesis have been designed to evaluate the success of various aspects of the proposed framework. Further analysis is required in order to evaluate the true extent of its limitations.

References

[1] Icann. (2011). *Internet protocol (ip) addresses*. *Beginner's Guide*. icann. Retrieved from https://www.icann.org/en/system/files/files/ip-addresses-beginners-guide-04mar11-en.pdf

[2] Viega, J., 2009. Cloud computing and the common man. Computer, 42(8), pp.106-108.

[3] Reese, G., 2009. Cloud application architectures: building applications and infrastructure in the cloud. " O'Reilly Media, Inc.".

[4] https://www.virtualbox.org/

[5] https://www.python.org/doc/essays/blurb/

[6] Atlasis, A., 2012. Attacking ipv6 implementation using fragmentation. BlackHat Europe, pp.14-16.

[7] Garfinkel, T. and Rosenblum, M., 2003, February. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Ndss* (Vol. 3, No. 2003, pp. 191-206).

[8] Pinyathinun, M. and Sathitwiriyawong, C., 2009, November. Dynamic policy model for target based intrusion detection system. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human* (pp. 930-934). ACM.

[9] Sujatha, P., Priya, C.S. and Kannan, A., 2012, August. Network intrusion detection system using genetic network programming with support vector machine. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics* (pp. 645-649). ACM.

[10] Arkko, J. and Nikander, P., 2003, April. Limitations of IPsec policy mechanisms. *In International Workshop on Security Protocols* (pp. 241-251). Springer, Berlin, Heidelberg.

[11] Munz, G. and Carle, G., 2007, May. Real-time analysis of flow data for network attack detection. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on* (pp. 100-108). IEEE.

[12] Diadem Firewall European Project, http://www.diadem-firewall.org

[13] Alhamaty, M., Yazdian, A. and Al-qadasi, F., 2007. Intrusion Detection System Based On The Integrity of TCP Packet. *Transactions On Engineering, Computing And Technology V11 February 2006 ISSN 1305-5313*, *39*, pp.241-256.

[14] Ziemba, G., Reed, D. and Traina, P., 1995. *Security considerations for IP fragment filtering* (No. RFC 1858).

[15] Hinden, R. and Deering, S., 2006. *IP version 6 addressing architecture* (No. RFC 4291).

[16] Kent, S. and Seo, K., 2005. *Security architecture for the internet protocol* (No. RFC 4301).

[17] Kent, S., 2005. *IP authentication header* (No. RFC 4302).

[18] Lent, S. (2005). *IP Encapsulating Security Payload (ESP)* (No. RFC 4303) .

[19] Jankiewicz, E., Loughney, J. and Narten, T., 2011. *Ipv6 node requirements* (No. RFC 6434).

[20] Deering, S., 1998. Internet Protocol, Version 6 (IPv6) Specification.

[21] Conta, A., Deering, S. and Gupta, M., 2006. *Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification* (No. RFC 4443).

[22] Narten, T., Nordmark, E., Simpson, W. and Soliman, H., 2007. *Neighbor discovery for IP version 6 (IPv6)* (No. RFC 4861).

[23] Hankins, D.W. and Mrugalski, T., 2011. *Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Options for Dual-Stack Lite*.

[24] Thomson, S., Narten, T. and Jinmei, T., 2007. *IPv6 stateless address autoconfiguration* (No. RFC 4862).

[25] Ismail, R., Syed, T.A. and Musa, S., 2014, January. Design and implementation of an efficient framework for behaviour attestation using n-call slides. In *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication* (p. 36). ACM.

[26] IAB, I., 2001. *IESG Recommendations on IPv6 Address Allocations to Sites*. IETF RFC 3177.

[27] Narten, T., Huston, G. and Roberts, L., 2011. *IPv6 address assignment to end sites* (No. RFC 6177).

[28] Perkins, C., Johnson, D. and Arkko, J., 2011. *Mobility support in IPv6* (No. RFC 6275).

[29] Nordmark, E. and Gilligan, R., 2005. *Basic transition mechanisms for IPv6 hosts and routers* (No. RFC 4213).

[30] Haas, J. and Hares, S., 2005. *Definitions of managed objects for BGP-4* (No. RFC 4273).

[31] Carpenter, B. and Moore, K., 2001. *Connection of IPv6 domains via IPv4 clouds* (No. RFC 3056).

[32] Huitema, C., 2001. *An anycast prefix for 6to4 relay routers* (No. RFC 3068).

[33] Gont, F. and Liu, W., 2014. *Security implications of IPv6 on IPv4 networks* (No. RFC 7123).

[34] Townsley, W. and Troan, O., 2010. *IPv6 Rapid Deployment on IPv4 Infrastructures (6rd)--Protocol Specification* (No. RFC 5969).

[35] Carpenter, B. and Jung, C., 1999. *Transmission of IPv6 over IPv4 domains without explicit tunnels* (No. RFC 2529).

[36] Wu, J., Cui, Y., Li, X., Xu, M. and Metz, C., 2010. *4over6 transit solution using IP encapsulation and MP-BGP extensions* (No. RFC 5747).

[37] Templin, F., Gleeson, T., Talwar, M. and Thaler, D., 2008. Intra-Site Automatic Tunnel Addressing Protocol (ISATAP)", RFC 5214.

[38] Huitema, C., 2006. *Teredo: Tunneling IPv6 over UDP through network address translations (NATs)* (No. RFC 4380).

[39] Thaler, D., 2011. *Teredo extensions* (No. RFC 6081).

[40] Tsirtsis, G. and Srisuresh, P., 2000. *Network address translation-protocol translation*

(NAT-PT) (No. RFC 2766).

[41] Aoun, C. and Davies, E., 2007. *Reasons to move the Network Address Translator-Protocol Translator (NAT-PT) to historic status* (No. RFC 4966).

[42] Baker, F., Li, X., Bao, C. and Yin, K., 2011. *Framework for ipv4/ipv6 translation* (No. RFC 6144).

[43] Bao, C., Huitema, C., Bagnulo, M., Boucadair, M. and Li, X., 2010. *IPv6 addressing of IPv4/IPv6 translators* (No. RFC 6052).

[44] Li, X., Bao, C., Wing, D., Vaithianathan, R. and Huston, G., 2012. *Stateless source address mapping for icmpv6 packets* (No. RFC 6791).

[45] Bagnulo, M., Matthews, P. and van Beijnum, I., 2011. *Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers* (No. RFC 6146).

[46] Bagnulo, M., Sullivan, A., Matthews, P. and Van Beijnum, I., 2011. *DNS64: DNS extensions for network address translation from IPv6 clients to IPv4 servers* (No. RFC 6147).

[47] Droms, R., 2004. *Stateless dynamic host configuration protocol (DHCP) service for IPv6* (No. RFC 3736).

[48] http://www.sixscape.com/joomla/sixscape/index.php/technical-backgrounders/tcp-ip/ip-the-internet-protocol/ipv4-internet-protocol-version-4/ipv4-packet-header (Accessed 10/04/2014)

[49] Erickson, J., 2008. *Hacking: the art of exploitation*. No starch press.

[50]http://www.juniper.net/techpubs/software/junos-es/junos-es93/junos-es-swconfig-security/understanding-teardrop-attacks.html (Accessed 10/04/2014)

[51] Clark, D.D., 1982. *IP datagram reassembly algorithms* (No. RFC 815).

[52] Barreno, M., Nelson, B., Sears, R., Joseph, A.D. and Tygar, J.D., 2006, March. Can machine learning be secure?. In Proceedings of the 2006 ACM Symposium on Information, computer and communications security (pp. 16-25). ACM.

[53] https://snort.org/

[54] Ptacek, T.H. and Newsham, T.N., 1998. *Insertion, evasion, and denial of service: Eluding network intrusion detection*. SECURE NETWORKS INC CALGARY ALBERTA.

[55] Gilad, Y. and Herzberg, A., 2011, August. Fragmentation considered vulnerable: blindly intercepting and discarding fragments. In *Proceedings of the 5th USENIX conference on Offensive technologies* (pp. 2-2). USENIX Association.

[56] Erickson, john. (2007). the Art of Exploitation, 1–492. Retrieved from https://leaksource.files.wordpress.com/2014/08/hacking-the-art-of-exploitation.pdf

[58] Eddy, W., 2007. *TCP SYN flooding attacks and common mitigations* (No. RFC 4987).

[58] Axelsson, S., 2000. *Intrusion detection systems: A survey and taxonomy* (Vol. 99). Technical report.

[59] Wu, P., Cui, Y., Wu, J., Liu, J. and Metz, C., 2013. Transition from IPv4 to IPv6: A state-of-the-art survey. *IEEE Communications Surveys & Tutorials*, *15*(3), pp.1407-1424.

[60] Wang, K. and Stolfo, S.J., 2004, September. Anomalous payload-based network intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection* (pp. 203-222). Springer, Berlin, Heidelberg.

[61] Wespi, A., Dacier, M. and Debar, H., 2000, October. Intrusion detection using variable-length audit trail patterns. In *International Workshop on Recent Advances in Intrusion Detection* (pp. 110-129). Springer, Berlin, Heidelberg.

[62] LANcope, "StealthWatch vs. Existing IDS Technology," 2001; http://www.lancope.com

[63] Kumar, V. and Sangwan, O.P., 2012. Signature based intrusion detection system using SNORT. *International Journal of Computer Applications & Information Technology*, *1*(3), pp.35-41.

[64] Majeed, P.G. and Kumar, S., 2014. Genetic algorithms in intrusion detection systems: A survey. *International Journal of Innovation and Applied Studies*, *5*(3), p.233.

[65] Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D. and Levy, H.M., 2007, August. SpyProxy: Execution-based Detection of Malicious Web Content. In *USENIX Security Symposium* (pp. 1-16).

[66] Bace, R. (1999). An Introduction to Intrusion Detection & Assessment. *Technical White Paper, ICSA*, (Security 401). Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:An+Introduction+to+Intrusion+Detection+and+Assessment#0

[67] Sandhu, U.A., Haider, S., Naseer, S. and Ateeb, O.U., 2011. A survey of intrusion detection & prevention techniques. In *2011 International Conference on Information Communication and Management, IPCSIT* (Vol. 16, pp. 66-67).

[68] Papadogiannakis, A., Polychronakis, M. and Markatos, E.P., 2010, April. Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In *Proceedings of the Third European Workshop on System Security* (pp. 15-21). ACM.

[69] Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C. and Vigna, G., 2013, August. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *USENIX Security Symposium* (pp. 637-652).

[70] Mabu, S., Chen, C., Lu, N., Shimada, K. and Hirasawa, K., 2011. An intrusion-detection model based on fuzzy class-association-rule mining using genetic network programming. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *41*(1), pp.130-139.

[71] Khayam, S.A., Ashfaq, A.B. and Radha, H., 2011. Joint network-host based malware detection using information-theoretic tools. *Journal in computer virology*, *7*(2), pp.159-172.

[72] Govindarajan, M. and Chandrasekaran, R.M., 2011. Intrusion detection using neural based hybrid classification methods. *Computer networks*, *55*(8), pp.1662-1671.

[73] Amaral, J.P., Oliveira, L.M., Rodrigues, J.J., Han, G. and Shu, L., 2014, June. Policy and network-based intrusion detection system for IPv6-enabled wireless sensor networks. In *Communications (ICC), 2014 IEEE International Conference on* (pp. 1796-1801). IEEE.

[74] Farnaaz, N. and Jabbar, M.A., 2016. Random forest modeling for network intrusion detection system. *Procedia Computer Science, 89*, pp.213-217.

[75] Hussain, J., Lalmuanawma, S. and Chhakchhuak, L., 2016. A two-stage hybrid classification technique for network intrusion detection system. *International Journal of Computational Intelligence Systems*, *9*(5), pp.863-875.

[76] Al-mamory, S.O. and Jassim, F.S., 2015. On the designing of two grains levels network intrusion detection system. *Karbala International Journal of Modern Science*, *1*(1), pp.15-25.

[77] Davies, E. and Mohacsi, J., 2007. *Recommendations for filtering icmpv6 messages in firewalls* (No. RFC 4890).

[78] Weon, I.Y., Song, D.H. and Lee, C.H., 2006. Effective intrusion detection model through the combination of a signature-based intrusion detection system and a machine learning-based intrusion detection system. *Journal of information science and engineering, 22*(6), pp.1447-1464.

[79] Atkinson, R. and Kent, S., 1998. IP authentication header.

[80] Zheng, K., Cai, Z., Zhang, X., Wang, Z. and Yang, B., 2015. Algorithms to speedup pattern matching for network intrusion detection systems. *Computer Communications*, *62*, pp.47-58.

[81] Kent, S., 2005. *IP encapsulating security payload (ESP)* (No. RFC 4303).

[82] http://datacomsystems.com/news-events/news/2015/nov/3/explore-new-firmware-that-s-helping-organizations-see-attacks-as-they-happen - accessed 14/11/2016

[83] Yao, L., Li, Z.T. and Hao, T., 2005, March. Dynamic immune intrusion detection system for IPv6. In *Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security 2005* (Vol. 5812, pp. 374-382). International Society for Optics and Photonics.

[84] Davies, E., Krishnan, S. and Savola, P., 2007. *IPv6 transition/co-existence security considerations* (No. RFC 4942).

[85] Curtin, B., 1999. *Internationalization of the file transfer protocol* (No. RFC 2640).

[86] Saeys, Y., Inza, I. and Larrañaga, P., 2007. A review of feature selection techniques in bioinformatics. *bioinformatics, 23*(19), pp.2507-2517.

[87] Song, S. and Manikopoulos, C.N., 2006, March. IP Spoofing Detection Approach (ISDA) for Network Intrusion Detection System. In *Sarnoff Symposium, 2006 IEEE* (pp. 1-4). IEEE.

[88] Chunyue, Z., Yun, L. and Hongke, Z., 2006, December. A pattern matching based network intrusion detection system. In *Control, Automation, Robotics and Vision, 2006. ICARCV'06. 9th International Conference on* (pp. 1-4). IEEE.

[89] Abuadlla, Y., Kvascev, G., Gajin, S. and Jovanovic, Z., 2014. Flow-based anomaly intrusion detection system using two neural network stages. *Computer Science and Information Systems*, *11*(2), pp.601-622.

[90] Vizváry, M. and Vykopal, J., 2013. Flow-based detection of RDP brute-force attacks. In *Proceedings of 7th International Conference on Security and Protection of Information, SPI* (Vol. 13, pp. 131-138).

[91] Muraleedharan, N. and Parmar, A., 2010. ADRISYA: a flow based anomaly detection system for slow and fast scan. *International Journal of Network security and its Applications (IJNSA)*, *2*(4).

[92] Hofstede, R., Bartos, V., Sperotto, A. and Pras, A., 2013, October. Towards real-time intrusion detection for NetFlow and IPFIX. In *Network and Service Management (CNSM), 2013 9th International Conference on* (pp. 227-234). IEEE.

[93] Hellemons, L., Hendriks, L., Hofstede, R., Sperotto, A., Sadre, R. and Pras, A., 2012, June. SSHCure: a flow-based SSH intrusion detection system. In *IFIP International Conference on Autonomous Infrastructure, Management and Security* (pp. 86-97). Springer, Berlin, Heidelberg.

[94] https://www.google.com/intl/en/ipv6/statistics.html - accessed on 09/10/2017

[95] http://6lab.cisco.com/stats/index.php?option=all – accessed on 09/10/2017

[96] https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml - accessed on 20/09/2017

[97] Cheswick, W.R., Bellovin, S.M. and Rubin, A.D., 2003. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc..

[98] http://phrack.org/issues/48/1.html - accessed 01/02/2017

[99] Lonvick, C. and Ylonen, T., 2006. The Secure Shell (SSH) protocol architecture. *IETF RFC 4251*.

[100] Ylonen, T. and Lonvick, C., 2005. *The secure shell (SSH) authentication protocol* (No. RFC 4252).

[101] Ylonen, T. and Lonvick, C., 2005. *The secure shell (SSH) transport layer protocol* (No. RFC 4253).

[102] Paterson, K.G., Poettering, B. and Schuldt, J.C., 2014, March. Plaintext recovery attacks against WPA/TKIP. In *International Workshop on Fast Software Encryption* (pp. 325-349). Springer, Berlin, Heidelberg.

[103] Lancaster, T., 2006. IPv6 & IPv4 Threat Review with Dual-Stack Considerations. Individual Research Project.

[104] Convery, S. and Miller, D., 2004. Ipv6 and ipv4 threat comparison and best-practice evaluation (v1. 0). *Presentation at the 17th NANOG, 24*, p.16.

[105] Kent, C.A. and Mogul, J.C., 1987. *Fragmentation considered harmful* (Vol. 17, No. 5, pp. 390-401).

[106] Barker, K., 2013. The security implications of IPv6. *Network Security*, *2013*(6), pp.5-9.

[107] Gao, J. and Chen, Y., 2014. Detecting DOS/DDOS Attacks Under Ipv6. In Proceedings of the 2012 International Conference on Cybernetics and Informatics (pp. 847-855). Springer, New York, NY.

[108] Lin, Z.W., Wang, L.H. and Ma, Y., 2006. Possible attacks based on ipv6 features and its detection. In *Asia-Pacific Advanced Network (APAN) 24th Meeting*.

[109] https://github.com/vanhauser-thc/thc-ipv6

[110] https://github.com/kbandla/dpkt/pull/403

[111] Mali, P., McManus, J., Rao, J. and Sanghvi, R., 2015. Mitigating IPv6 Vulnerabilities.

[112] Satrya, G.B., Chandra, R.L. and Yulianto, F.A., 2015, May. The detection of ddos flooding attack using hybrid analysis in ipv6 networks. In *Information and Communication Technology (ICoICT), 2015 3rd International Conference on* (pp. 240-244). IEEE.

[113] Lee, K., Kim, J., Kwon, K.H., Han, Y. and Kim, S., 2008. DDoS attack detection method using cluster analysis. *Expert systems with applications*, *34*(3), pp.1659-1665.

[114] Hogg, S., & Vyncke, E. (2013). *IPv6 Security*. 800 East 96th Street Indianapolis, IN 46240 USA: Cisco Press.

[115] https://suricata-ids.org/

[116] Martin, C.E. and Dunn, J.H., 2007, October. Internet Protocol version 6 (IPv6) protocol security assessment. In *Military Communications Conference, 2007. MILCOM 2007. IEEE* (pp. 1-7). IEEE.

[117] Supriyanto, Hasbullah, I.H., Murugesan, R.K. and Ramadass, S., 2013. Survey of internet protocol version 6 link local communication security vulnerability and mitigation methods. *IETE Technical Review*, *30*(1), pp.64-71.

[118] Tripathi, N. and Mehtre, B.M., 2013. DoS and DDoS attacks: Impact, analysis and countermeasures. In *National Conference on Advances in Computing, Networking and Security, Nanded, India*.

[119] Nikander, P., Kempf, J. and Nordmark, E., 2004. *IPv6 neighbor discovery (ND) trust models and threats* (No. RFC 3756).

[120] Stallings, W. (2011). *the William Stallings Books on Computer Data and Computer Communications , Eighth Edition*. *Network* (Vol. 139). http://doi.org/10.1007/11935070

[121] Cisco. (2011). Cisco IOS IPv6 Configuration Guide, (6387). Retrieved from http://www.cisco.com/en/US/docs/ios/ipv6/configuration/guide/12_4/ipv6_12_4_book.pdf

[122] Grundemann, Chris. chrisgrundemann.com. https://chrisgrundemann.com/index.php/2012/introducing-ipv6-classifying-ipv6-addresses/ (accessed June 14, 2017).

[123] Ullrich, J., Krombholz, K., Hobel, H., Dabrowski, A. and Weippl, E.R., 2014, August. IPv6 Security: Attacks and Countermeasures in a Nutshell. In *WOOT*.

[124] “Tiny Fragment Attack.” definedterm.com. https://definedterm.com/tiny_fragment_attack (accessed June 14, 2018).

[125] Atlasis, A., 2012. Security impacts of abusing IPv6 extension headers. In Black Hat security conference (pp. 1-10).

[126] Yang, D., Song, X., & Guo, Q. (2010). Security on IPv6. In *Proceedings - 2nd IEEE International Conference on Advanced Computer Control, ICACC 2010* (Vol. 3, pp. 323–326). http://doi.org/10.1109/ICACC.2010.5486848

[127] Arkko, J., Aura, T., Kempf, J., Mäntylä, V.M., Nikander, P. and Roe, M., 2002, September. Securing IPv6 neighbor and router discovery. In *Proceedings of the 1st ACM workshop on Wireless security* (pp. 77-86). ACM.

[128] Arkko, J., Kempf, J., Zill, B. and Nikander, P., 2005. *Secure neighbor discovery (SEND)* (No. RFC 3971).

[129] AlSa'deh, A. and Meinel, C., 2012. Secure neighbor discovery: Review, challenges, perspectives, and recommendations. *IEEE Security & Privacy*, *10*(4), pp.26-34.

[130]https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1002934

[131] Saad, R.M., Almomani, A., Altaher, A., Gupta, B.B. and Manickam, S., 2014. ICMPv6 flood attack detection using DENFIS algorithms. *Indian Journal of Science and Technology*, *7*(2), pp.168-173.

[132] Saad, R.M., Anbar, M., Manickam, S. and Alomari, E., 2016. An intelligent icmpv6 ddos flooding-attack detection framework (v6iids) using back-propagation neural network. *IETE Technical Review*, *33*(3), pp.244-255.

[133] Rafiee, H. and Meinel, C., 2013, July. SSAS: A simple secure addressing scheme for IPv6 autoconfiguration. In *2013 Eleventh Annual Conference on Privacy, Security and Trust (PST)* (pp. 275-282). IEEE.

[134] Eckstein, C. and Atlasis, A., 2011. OS fingerprinting with IPv6. *Infosec reading room, SANS Institute*.

[135] Allen, J.M., 2007. OS and Application Fingerprinting Techniques. *SANS Institute InfoSec Reading Room*.

[136] Xu, Y., Li, X., Zhou, J. and Qian, H., 2009, December. Worm detection in an ipv6 internet. In *2009 International Conference on Computational Intelligence and Security* (pp. 366-370). IEEE.

[137] Headquarters, A., 2012. Security Configuration Guide: Zone-Based Policy Firewall Cisco IOS Release 15.0 M.

[138] Choudhary, A.R., 2009, November. In-depth analysis of IPv6 security posture. In *Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on* (pp. 1-7). IEEE.

[139] https://www.darknet.org.uk/2010/09/havij-advanced-automated-sql-injection-tool - accessed 17/07/2017

[140] https://www.acunetix.com/ - accessed 16/02/2017

[141] Gorton, A. and Champion, T., 2004. Combining evasion techniques to avoid network intrusion detection systems. *North Chelmsford, MA, US: Skaion*

[142] Atlasis, A. (2017). The Impact of Extension Headers on IPv6 Access Control Lists Real-Life Use Cases. White Paper.

–Appendix1

```
/*
 * Tests various IPv6 specific options for their implementations
 * This can also be used to test firewalls, check what it passes.
 * A sniffer on the other side of the firewall or running implementation6d
 * shows you what got through.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <time.h>
#include <pcap.h>
#include "thc-ipv6.h"

int ret_code = 1, matched = 0, gtype1, gtype1a, gtype2, gtype2a, gpos,
epos, onecase = 0;
unsigned char *gpattern, *gsrc, *gdst, etype, ecode;

void help(char *prg) {
  printf("%s %s (c) 2018 by %s %s\n\n", prg, VERSION, AUTHOR, RESOURCE);
  printf("Syntax: %s [-p] [-s sourceip6] interface destination [test-case-
number]\n\n", prg);
  printf("Options:\n");
  printf("  -s sourceip6  use the specified source IPv6 address\n");
  printf("  -p            do not perform an alive check at the beginning
and end\n");
  printf("\nPerforms some IPv6 implementation checks, can be used to test
some\nfirewall features too. Takes approx. 2 minutes to complete.\n");
  exit(-1);
}

void ignoreit(u_char *foo, const struct pcap_pkthdr *header, const unsigned
char *data) {
  return;
}

void check_packet_n(u_char *foo, const struct pcap_pkthdr *header, const
unsigned char *data) {
  unsigned char *ipv6hdr = (unsigned char *) data, len = header->caplen;
  int off = 0;

  ipv6hdr = (unsigned char *) (data + 14);
  len -= 14;
  if (do_hdr_size) {
    ipv6hdr = (unsigned char *) (data + do_hdr_size);
    len -= (do_hdr_size - 14);
    if ((ipv6hdr[0] & 240) != 0x60)
      return;
  }

  if (debug) {
    printf("DEBUG: packet received\n");
    thc_dump_data(ipv6hdr, len, "Received Packet");
  }
  if (ipv6hdr[6] == NXT_FRAG)
    off = 8;
  if (86 + off < len) {
    if (debug)
      printf("\nDEBUG: packet too short\n");
    return;
  }
  if (ipv6hdr[6] == NXT_ICMP6 && (ipv6hdr[40] == ICMP6_NEIGHBORSOL ||
ipv6hdr[40] == ICMP6_TTLEXEED))
    return;
  if (off == 8 && (ipv6hdr[40] == NXT_ICMP6 && (ipv6hdr[40+off] ==
ICMP6_NEIGHBORSOL || ipv6hdr[40+off] == ICMP6_TTLEXEED)))
    return;
  if ((ipv6hdr[6] == NXT_ICMP6 && ipv6hdr[40] == ICMP6_NEIGHBORADV) || (off
== 8 && ipv6hdr[40] == NXT_ICMP6 && ipv6hdr[40+off] == ICMP6_NEIGHBORADV))
{
    if (memcmp(ipv6hdr + 8, gdst, 16) == 0 && memcmp(ipv6hdr + 24 , gsrc,
16) == 0) {
      matched = 2;
      return;
    }
  } else if ((ipv6hdr[6] == NXT_ICMP6 && ipv6hdr[40] ==
ICMP6_PARAMPROB)||(off == 8 && ipv6hdr[40] == NXT_ICMP6 && ipv6hdr[40+off]
== ICMP6_PARAMPROB)) {
    if (memcmp(ipv6hdr + 8, gsrc, 16) == 0 && memcmp(ipv6hdr + 24, gdst,
16) == 0) {
      matched = 1;
      etype = ipv6hdr[40];
      ecode = ipv6hdr[41];
      return;
    }
  }

  return;
}


int check_for_reply_n(pcap_t * p, unsigned char *src, unsigned char *dst) {
  int ret = -1;
  time_t t;

  t = time(NULL);
  matched = 0;
  gsrc = src, gdst = dst;
  while (ret < 0) {
    (void) thc_pcap_check(p, (char *) check_packet_n, NULL);
    if (matched > 0)
      ret = 0;
    if (time(NULL) > t + 2 && ret < 0)
      ret = 0;
  }

  if (matched <= 0)
    printf("FAILED - no reply\n");
  if (matched == 1) {
    printf("FAILED - error reply [%d:%d]\n", etype, ecode);
    if (onecase == 0)
      sleep(2);
  }
  if (matched == 2) {
    printf("PASSED - we got a reply\n");
    ret_code = 0;
  }

  usleep(500);
  return matched;
}


void check_packet(u_char *foo, const struct pcap_pkthdr *header, const
unsigned char *data) {
  unsigned char *ipv6hdr = (unsigned char *) (data + 14);
  int len = header->caplen - 14, off = 0;

  if (do_hdr_size) {
    ipv6hdr = (unsigned char *) (data + do_hdr_size);
    len -= (do_hdr_size - 14);
    if ((ipv6hdr[0] & 240) != 0x60)
      return;
  }

  matched = 0;
  if (debug) {
    printf("DEBUG: packet received\n");
    thc_dump_data(ipv6hdr, len, "Received Packet");
  }
  if (ipv6hdr[6] == NXT_FRAG)
    off = 8;
  if (gpos+off > len && (epos == 0 || epos+off > len)) {
    matched = -1;
    if (debug)
      printf("\nDEBUG: packet too short (2)\n");
    return;
  }
  if ((ipv6hdr[6] == NXT_ICMP6 || (off == 8 && ipv6hdr[40] == NXT_ICMP6))
&& (ipv6hdr[40+off] == ICMP6_NEIGHBORSOL || ipv6hdr[40] ==
ICMP6_NEIGHBORADV || ipv6hdr[40+off] == ICMP6_TTLEXEED) && ipv6hdr[40+off]
!= gtype2
      && ipv6hdr[40+off] != gtype2a) {
    matched = -1;
    return;
  }
//printf("gpos: %d, pattern %x, found %x\n", gpos, gpattern[0],
ipv6hdr[gpos]);
//printf("epos: %d, pattern %x, found %x\n", epos, gpattern[0],
ipv6hdr[epos]);
  if (gpos > 0 && memcmp(ipv6hdr + gpos + off, gpattern, 4) != 0) {
    matched = -1;
    if (debug)
      printf("\nDEBUG: packet contents different\n");
    if (epos == 0)
      return;
  } else {
    matched = 1;
    etype = ipv6hdr[40];
    ecode = ipv6hdr[41];
  }
  if (epos > 0 && epos < len && memcmp(ipv6hdr + epos + off, gpattern, 4)
== 0) {
    matched = 1;
    etype = ipv6hdr[40];
    ecode = ipv6hdr[41];
  }
  if ((ipv6hdr[6] == gtype1 || gtype1 == 0) && (ipv6hdr[40] == gtype2 ||
gtype2 == 0)
      && (gpos <= 0 || (gpos < len && memcmp(ipv6hdr + gpos, gpattern, 4)
== 0)))
    matched = 2;
  if (off == 8 && ((ipv6hdr[40] == gtype1 || gtype1 == 0) &&
(ipv6hdr[40+off] == gtype2 || gtype2 == 0)
      && (gpos <= 0 || (gpos < len && memcmp(ipv6hdr + gpos + off,
gpattern, 4) == 0))))
    matched = 2;
  if ((ipv6hdr[6] == gtype1a || gtype1a == 0) && (ipv6hdr[40] == gtype2a ||
gtype2a == 0)
      && (gpos <= 0 || (gpos + off < len && memcmp(ipv6hdr + gpos,
gpattern, 4) == 0)))
    matched = 2;
  if (off == 8 && ((ipv6hdr[40] == gtype1a || gtype1a == 0) &&
(ipv6hdr[40+off] == gtype2a || gtype2a == 0)
      && (gpos <= 0 || (gpos + off < len && memcmp(ipv6hdr + gpos + off,
gpattern, 4) == 0))))
    matched = 2;
  if (debug)
    printf("\nDEBUG: hdr[6] %d|%d == %d, hdr[40] %d|%d == %d, pos[%d/%d]
%02x%02x%02x%02x == %02x%02x%02x%02x\n", ipv6hdr[6], gtype1, gtype1a,
ipv6hdr[40], gtype2, gtype2a, gpos,
           epos, gpos == 0 ? 0 : ipv6hdr[gpos], gpos == 0 ? 0 :
ipv6hdr[gpos + 1], gpos == 0 ? 0 : ipv6hdr[gpos + 2], gpos == 0 ? 0 :
ipv6hdr[gpos + 3],
           gpos == 0 ? 0 : gpattern[0], gpos == 0 ? 0 : gpattern[1], gpos
== 0 ? 0 : gpattern[2], gpos == 0 ? 0 : gpattern[3]);

  return;
}

int check_for_reply(pcap_t *p, int type1, int type2, int type1a, int
type2a, int pos, int pos2, unsigned char *pattern) {
  int ret = -1;
  time_t t;

  t = time(NULL);
  matched = 0;
  gtype1 = type1;
  gtype1a = type1a, gtype2 = type2;
  gtype2a = type2a, gpos = pos;
  epos = pos2;
  gpattern = pattern;
  while (ret < 0) {
    if (thc_pcap_check(p, (char *) check_packet, NULL) > 0)
      ret = 1;
    if (matched == -1) {
      ret = -1;
      matched = 0;
    }
    if (time(NULL) > t + 2 && ret < 0)
      ret = 0;
  }

  if (matched == 0)
```

```
    printf("FAILED - no reply\n");
  if (matched == 1) {
    printf("FAILED - error reply [%d:%d]\n", etype, ecode);
    if (onecase == 0)
      sleep(2);
  }
  if (matched == 2) {
    printf("PASSED - we got a reply\n");
    ret_code = 0;
  }

  usleep(500);
  return matched;
}


int check_alive(pcap_t * p, char *interface, unsigned char *src, unsigned
char *dst) {
  int ret = -2;
  time_t t;

  while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
  thc_ping6(interface, src, dst, 16, 1);
  t = time(NULL);
  while (ret < 0) {
    if (thc_pcap_check(p, (char *) ignoreit, NULL) > 0)
      ret = 1;
    if (time(NULL) > t + 1 && ret == -2) {
      thc_ping6(interface, src, dst, 16, 1);
      ret = -1;
    }
    if (time(NULL) > t + 2 && ret < 0)
      ret = 0;
  }

  return ret > 0 ? 1 : 0;
}


int main(int argc, char *argv[]) {
  int test = 0, count = 1;
  unsigned char buf[1500], bla[1500], bigbla[65536], tests[256], string[64]
= "ip6 and dst ", string2[64] = "ip6 and src ";
  unsigned char *dst6, *ldst6 = malloc(16), *src6 = NULL, *lsrc6, *mcast6;
  unsigned char *srcmac = NULL, *dstmac = NULL, *routers[2],
null_buffer[6];
  thc_ipv6_hdr *hdr;
  int i, j, k, srcmtu, fragsize, use_srcroute_type = -1, offset = 14;
  pcap_t *p;
  unsigned char *pkt = NULL, *pkt2 = NULL, *pkt3 = NULL;
  int pkt_len = 0, pkt_len2 = 0, pkt_len3 = 0, noping = 0;
  char *interface;

  setvbuf(stdout, NULL, _IONBF, 0);
  setvbuf(stderr, NULL, _IONBF, 0);

  if (argc < 3 || strncmp(argv[1], "-h", 2) == 0)
    help(argv[0]);

  while ((i = getopt(argc, argv, "pds:")) >= 0) {
    switch (i) {
      case 'p':
        noping = 1;
        break;
      case 'd':
        debug = 1;
        break;
      case 's':
        src6 = thc_resolve6(optarg);
        break;
      default:
        fprintf(stderr, "Error: unknown option %c\n", i);
        exit(-1);
    }
  }

  interface = argv[optind];
  dst6 = thc_resolve6(argv[optind + 1]);
  if (dst6 == NULL) {
    fprintf(stderr, "Error: can not resolve %s to a valid IPv6 address\n",
argv[optind + 1]);
    exit(-1);
  }
  memcpy(ldst6, dst6, 16);
  memset(ldst6 + 2, 0, 6);
  ldst6[0] = 0xfe;
  ldst6[1] = 0x80;
  mcast6 = thc_resolve6("ff02::1");
  if (argc >= optind + 3) {
    test = atoi(argv[optind + 2]);
    onecase = 1;
  }
  memset(buf, 0, sizeof(buf));
  memset(null_buffer, 0, sizeof(null_buffer));
  if (do_hdr_size)
    offset = do_hdr_size;

  if (src6 == NULL)
    src6 = thc_get_own_ipv6(interface, dst6, PREFER_GLOBAL);
  if (src6 != NULL && src6[0] == 0xfe)
    lsrc6 = src6;
  else
    lsrc6 = thc_get_own_ipv6(interface, ldst6, PREFER_LINK);
  if (lsrc6 == NULL) {
    fprintf(stderr, "Error: invalid interface %s\n", interface);
    exit(-1);
  }
  strcat(string, thc_ipv62notation(src6));
  strcat(string2, thc_ipv62notation(dst6));
  srcmac = thc_get_own_mac(interface);
  if ((dstmac = thc_get_mac(interface, src6, dst6)) == NULL) {
    fprintf(stderr, "ERROR: Can not resolve mac address for %s\n",
argv[2]);
    exit(-1);
  }
  if ((srcmtu = thc_get_mtu(interface)) <= 0) {
    fprintf(stderr, "ERROR: can not get mtu from interface %s\n",
interface);
    exit(-1);
  }
  fragsize = ((srcmtu - 62) / 8) * 8;
  if ((p = thc_pcap_init(interface, string)) == NULL) {
    fprintf(stderr, "Error: could not capture on interface %s with string
%s\n", interface, string);
    exit(-1);
  }

  setvbuf(stdout, NULL, _IONBF, 0);
  memset(tests, 0, sizeof(tests));

  printf("Performing implementation checks on %s via %s:\n", argv[optind +
1], argv[optind]);
  if (noping == 0) {
    if (check_alive(p, interface, src6, dst6) == 0) {
      fprintf(stderr, "Error: target %s is not alive via direct ping6!\n",
argv[optind + 1]);
      exit(-1);
    } else
      printf("Test  0: normal ping6\t\t\t\tPASSED - we got a reply\n");
  }

  /********************** TEST CASES ***********************/

  if (test == 0 || test == count) {
    printf("Test %2d: hop-by-hop ignore option\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: 2 hop-by-hop headers\t\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    for (i = 0; i < 2; i++)
      if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) <
0)
        return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
```

```
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: 118 hop-by-hop headers\t\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    for (i = 0; i < 128; i++)
      if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) <
0)
        return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 1200, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: destination ignore option\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
  }
  count++;
  if (test == 0 || test == count) {
    //for (int mkz1 = 0 ; mkz1<2; mkz1++)
    //{
    printf("Test %2d: 2 destination headers\t\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    for (i = 0; i < 8168; i++)
      if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
        return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_as_fragment6(interface, src6, dst6, NXT_DST,
                                 hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    //printf("\t Sent %d\n",mkz1);
    //}
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: 119 destination headers\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    for (i = 0; i < 128; i++)
      if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
        return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 1200, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: 1329 destination headers\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    for (i = 0; i < 2000; i++)
      if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
        return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_as_fragment6(interface, src6, dst6, NXT_DST,
                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 1200, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: 5427 destination headers\t\t", count);
    //for (int mkz1 = 0 ; mkz1<5000; mkz1++)
    //{
        if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len, src6, dst6, 255, 0, count, 0, 0)) == NULL)
          return -1;
        memset(bla, count % 256, sizeof(bla));
        memset(buf, 0, sizeof(buf));
        buf[0] = NXT_IGNORE;
        buf[1] = 0;
        for (i = 0; i < 8172; i++)
          if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) <
0)
            return -1;
        thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
        if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
          return -1;
        hdr = (thc_ipv6_hdr *) pkt;
        while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
        if (thc_send_as_fragment6(interface, src6, dst6, NXT_DST,
                                  hdr->pkt + 40 + offset, hdr->pkt_len - 40
- offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
          return -1;
    //printf("\t Sent %d\n",mkz1);
    //}
        pkt = thc_destroy_packet(pkt);
        if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 1200, bla))
          tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: correct fragmentation\t\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, sizeof(bla), 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_as_fragment6(interface, src6, dst6, NXT_ICMP6,
                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_FRAG, NXT_ICMP6, NXT_FRAG, NXT_ICMP6,
fragsize - 100, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: one-shot fragmentation\t\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_oneshotfragment(pkt, &pkt_len, getpid() + 70000) < 0)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, fragsize - 100, 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
//    hdr = (thc_ipv6_hdr *) pkt;
//    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
//    if (thc_send_as_fragment6(interface, src6, dst6, NXT_ICMP6,
//                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
//      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, fragsize - 200, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: overlap-first-zero fragmentation\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, srcmtu - 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    if ((pkt2 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len2, src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_dst(pkt2, &pkt_len2, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt2, &pkt_len2, ICMP6_PINGREPLY, 0, count, (unsigned
char *) &bla, srcmtu - 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt2, &pkt_len2) < 0)
      return -1;

    /* frag stuff */
    hdr = (thc_ipv6_hdr *) pkt;
    i = ((hdr->pkt_len - 40 - offset - 10) / 8) * 8;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, src6, dst6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 0, 1, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_add_data6(pkt3, &pkt_len3, 60, buf, hdr->pkt_len - 40 - offset
- 22))
      return -1;
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);
    hdr = (thc_ipv6_hdr *) pkt2;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, dst6, src6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 0, 0, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    if (thc_add_data6(pkt3, &pkt_len3, 60, buf, hdr->pkt_len - 40 - offset
- 22))
      return -1;
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);

    /* lets see if it worked */
    pkt = thc_destroy_packet(pkt);
    pkt2 = thc_destroy_packet(pkt2);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, fragsize - 200, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: overlap-last-zero fragmentation\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
```

```
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, srcmtu - 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    if ((pkt2 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len2, src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_dst(pkt2, &pkt_len2, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt2, &pkt_len2, ICMP6_PINGREPLY, 0, count, (unsigned
char *) &bla, srcmtu - 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt2, &pkt_len2) < 0)
      return -1;

    /* frag stuff */
    hdr = (thc_ipv6_hdr *) pkt2;
    i = ((hdr->pkt_len - 40 - offset - 10) / 8) * 8;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, src6, dst6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 0, 1, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    if (thc_add_data6(pkt3, &pkt_len3, NXT_HDR, buf, hdr->pkt_len - 40 -
offset - 22))
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);
    hdr = (thc_ipv6_hdr *) pkt;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, src6, dst6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 0, 0, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    if (thc_add_data6(pkt3, &pkt_len3, NXT_HDR, buf, hdr->pkt_len - 40 -
offset - 22))
      return -1;
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);

    /* lets see if it worked */
    pkt = thc_destroy_packet(pkt);
    pkt2 = thc_destroy_packet(pkt2);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, fragsize - 200, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: overlap-first-dst fragmentation\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = NXT_IGNORE;
    buf[1] = 0;
    if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, srcmtu - 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    if ((pkt2 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len2, src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_dst(pkt2, &pkt_len2, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt2, &pkt_len2, ICMP6_PINGREPLY, 0, count, (unsigned
char *) &bla, srcmtu - 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt2, &pkt_len2) < 0)
      return -1;

    /* frag stuff */
    hdr = (thc_ipv6_hdr *) pkt;
    i = ((hdr->pkt_len - 40 - offset - 10) / 8) * 8;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, src6, dst6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 0, 1, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    if (thc_add_data6(pkt3, &pkt_len3, NXT_DST, buf, hdr->pkt_len - 40 -
offset - 22))
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);
    hdr = (thc_ipv6_hdr *) pkt2;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, src6, dst6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 1, 0, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    if (thc_add_data6(pkt3, &pkt_len3, NXT_DST, buf + 8, hdr->pkt_len - 40
- offset - 22))
      return -1;
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);

    /* lets see if it worked */
    pkt = thc_destroy_packet(pkt);
    pkt2 = thc_destroy_packet(pkt2);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, fragsize - 200, 0, bla))
      tests[count] = 1;
  }
  count++;
  if (test == 0 || test == count) {
    printf("Test %2d: source-routing (done)\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    routers[0] = src6;          // route via ourself, but
    routers[1] = NULL;          // telling the target that this was already
performed
    if (thc_add_hdr_route(pkt, &pkt_len, routers, 0) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if ((k = check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))) {
      tests[count] = 1;
      if (k == 2 && use_srcroute_type < 0)
        use_srcroute_type = count;
    }
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: source-routing (todo)\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    routers[0] = src6;          // route via ourself, and
    routers[1] = NULL;          // telling the target that this was NOT
already performed
    if (thc_add_hdr_route(pkt, &pkt_len, routers, 1) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if ((k = check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 200, bla))) {
      tests[count] = 1;
      if (k == 2 && use_srcroute_type < 0)
        use_srcroute_type = count;
    }
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: unauth mobile source-route\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_mobileroute(pkt, &pkt_len, src6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 200, bla))      // XXX TODO: NOT SURE!
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: mobile+source-routing (done)\t\t", count);
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(bla, 0, sizeof(bla));
    bla[0] = 2;
    bla[1] = 1;
    memcpy(bla + 6, src6, 16);
    // 22 type, 23 routingptr, 24 reserved, 25-27 loose source routing
    memcpy(bla + 6 + 16 + 6, src6, 16);
    if (thc_add_hdr_misc(pkt, &pkt_len, NXT_ROUTE, -1, bla, 44) < 0)
      return -1;
    memset(bla, count % 256, sizeof(bla));
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if ((k = check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 200, bla))) {
      tests[count] = 1;
      if (k == 2 && use_srcroute_type < 0)
        use_srcroute_type = count;
    }
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: ping6 with a zero AH extension header\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    if (thc_add_hdr_misc(pkt, &pkt_len, NXT_AH, -1, (unsigned char *) &buf,
14) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    if (do_hdr_size != 0)
      hdr->pkt[do_hdr_size + 40 + 1] = 2;
    else
      hdr->pkt[14 + 40 + 1] = 2;
    thc_send_pkt(interface, pkt, &pkt_len);
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: ping from multicast (local!)\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
mcast6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    thc_pcap_close(p);
    p = thc_pcap_init(interface, string2);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
    thc_pcap_close(p);
    p = thc_pcap_init(interface, string);
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: frag+source-route to link local\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    routers[0] = lsrc6;          // route via ourself
    routers[1] = NULL;
    if (thc_add_hdr_route(pkt, &pkt_len, routers, 1) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    hdr = (thc_ipv6_hdr *) pkt;
    if (thc_send_as_fragment6(interface, src6, dst6, NXT_ROUTE,
                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: frag+source-route to multicast\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    routers[0] = mcast6;
    routers[1] = NULL;
    if (thc_add_hdr_route(pkt, &pkt_len, routers, 1) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    hdr = (thc_ipv6_hdr *) pkt;
    if (thc_send_as_fragment6(interface, src6, dst6, NXT_ROUTE,
                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 130, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: frag+srcroute from link local (local!)\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
lsrc6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    routers[0] = src6;
    routers[1] = NULL;
    if (thc_add_hdr_route(pkt, &pkt_len, routers, 0) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    thc_pcap_close(p);
    p = thc_pcap_init(interface, string2);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    if (thc_send_as_fragment6(interface, lsrc6, dst6, NXT_ROUTE,
                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
    thc_pcap_close(p);
    p = thc_pcap_init(interface, string);
  }
  count++;
  if (test == 0 || test == count) {
    printf("Test %2d: frag+srcroute from multicast (local!)\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
mcast6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    routers[0] = src6;
    routers[1] = NULL;
    if (thc_add_hdr_route(pkt, &pkt_len, routers, 0) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    thc_pcap_close(p);
    p = thc_pcap_init(interface, string2);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_as_fragment6(interface, mcast6, dst6, NXT_ROUTE,
                              hdr->pkt + 40 + offset, hdr->pkt_len - 40 -
offset, hdr->pkt_len > fragsize ? fragsize : (((hdr->pkt_len - 40 - 14) /
16) + 1) * 8) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ROUTE, NXT_ICMP6, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
    thc_pcap_close(p);
    p = thc_pcap_init(interface, string);
  }
  count++;


  if (test == 0 || test == count) {
    printf("Test %2d: jumbo option size < 64k\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = 0xc2;
    buf[1] = 4;
    buf[5] = 166;
    if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

    if (test == 0 || test == count) {
```

```
    printf("Test %2d: jumbo option size < 64k, length 0\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = 0xc2;
    buf[1] = 4;
    buf[5] = 166;
    if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    i = offset;
    hdr->pkt[4 + i] = 0;        // set ip length to 0
    hdr->pkt[5 + i] = 0;
    //hdr->pkt[5 + i] = 167;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: jumbo option size < 64k, length 167\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = 0xc2;
    buf[1] = 4;
    buf[5] = 166;
    if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    i = offset;
    hdr->pkt[4 + i] = 0;        // set ip length to 0
    //hdr->pkt[5 + i] = 0;
    hdr->pkt[5 + i] = 167;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;
  if (test == 0 || test == count) {
    printf("Test %2d: error option in hop-by-hop\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = 0xc3;
    buf[1] = 4;
    buf[5] = 166;
    if (thc_add_hdr_hopbyhop(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: error option in dsthdr\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    memset(buf, 0, sizeof(buf));
    buf[0] = 0xc3;
    buf[1] = 4;
    buf[5] = 166;
    if (thc_add_hdr_dst(pkt, &pkt_len, (unsigned char *) &buf, 6) < 0)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: 0 length field\t\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
```

```
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    i = offset;
    hdr->pkt[4 + i] = 0;        // set ip length to 0
    hdr->pkt[5 + i] = 0;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: too large length field\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    i = offset;
    hdr->pkt[4 + i] = 1;        // set ip length to 0
    hdr->pkt[5 + i] = 0;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: too small length field\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    i = offset;
    hdr->pkt[4 + i] = 0;        // set ip length to 0
    hdr->pkt[5 + i] = 60;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: ping6 with bad checksum\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0x6666);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt, &pkt_len)
< 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: ping6 with zero checksum\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0x6666);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    hdr = (thc_ipv6_hdr *) pkt;
    memset(hdr->pkt + hdr->pkt_len - 150 - 6, 0, 2);
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 100, 0, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: ping with hop count 0\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, -1, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, 150, 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    if (thc_send_pkt(interface, pkt, &pkt_len) < 0)
      return -1;
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, 140, 140 + i, bla))
      tests[count] = 1;
  }
  count++;

  if (test == 0 || test == count) {
    printf("Test %2d: fragment missing\t\t\t", count);
    memset(bla, count % 256, sizeof(bla));
    if ((pkt = thc_create_ipv6_extended(interface, PREFER_GLOBAL, &pkt_len,
src6, dst6, 255, 0, count, 0, 0)) == NULL)
      return -1;
    thc_add_icmp6(pkt, &pkt_len, ICMP6_PINGREQUEST, 0, count, (unsigned
char *) &bla, sizeof(bla) > 1400 ? 1400 : sizeof(bla), 0);
    if (thc_generate_pkt(interface, srcmac, dstmac, pkt, &pkt_len) < 0)
      return -1;

    /* frag stuff */
    hdr = (thc_ipv6_hdr *) pkt;
    i = ((hdr->pkt_len - 40 - offset - 10) / 8) * 8;
    if ((pkt3 = thc_create_ipv6_extended(interface, PREFER_GLOBAL,
&pkt_len3, src6, dst6, 0, 0, count, 0, 0)) == NULL)
      return -1;
    if (thc_add_hdr_fragment(pkt3, &pkt_len3, 128, 0, count))
      return -1;
    memcpy(buf, hdr->pkt + 40 + offset, hdr->pkt_len - 40 - offset);
    if (thc_add_data6(pkt3, &pkt_len3, 58, buf, hdr->pkt_len - 40 - offset
- 22))
      return -1;
    while (thc_pcap_check(p, (char *) ignoreit, NULL) > 0);
    thc_generate_and_send_pkt(interface, srcmac, dstmac, pkt3, &pkt_len3);
// ignore
    pkt3 = thc_destroy_packet(pkt3);

    /* lets see if it worked */
    pkt = thc_destroy_packet(pkt);
    if (check_for_reply(p, NXT_ICMP6, ICMP6_PINGREPLY, NXT_ICMP6,
ICMP6_PINGREPLY, fragsize - 200, 0, bla))
      tests[count] = 1;
  }
  count++;

  /******************* END OF TESTCASES ***************************/

  if (noping == 0) {
    if (check_alive(p, interface, src6, dst6))
      printf("Test %2d: normal ping6 (still alive?)\t\tPASSED - we got a
reply\n", count);
    else
      printf("Test %2d: normal ping6 (still alive?)\t\tFAILED - target is
unavailable now!\n", count);
  }

  thc_pcap_close(p);

  return ret_code;
}
```