University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Cybersecurity

Exploring IPv6 Extension Headers Security: From Manual Analysis to Differential Fuzzing

Supervisor
Mauro Conti
University of Padova

Co-supervisor
Enrico Bassetti
Delft University of Technology

Master Candidate
Francesco Drago

Student ID
2029019

Academic Year
2023-2024

To my unwavering family: your support fuels my journey. Forever grateful.

Abstract

The widespread adoption of IPv6 has seen significant traction, with nearly half of the world's networks transitioning to this protocol, a trend that continues to grow each year. However, this transition presents a number of challenges that require ongoing analysis and study. One important feature of IPv6 is its Extension headers, which have the potential to affect the behavior of routers and end systems. Past research has highlighted the vulnerabilities associated with Extension headers, including their exploitation for malicious purposes or unfair advantage. While efforts have been made to address some of these issues, many persist, and there are likely undiscovered vulnerabilities yet to be uncovered. This thesis addresses these challenges, beginning with an examination of Fragmentation headers, which have been identified as a source of problems such as overlapping fragments that can be used to create operating system fingerprints. The analysis then moves to atomic fragments, highlighting the harmful implications of their use, particularly in firewall evasion scenarios. In addition, a newly discovered bug, the ICMPv6 zero bug, is introduced, which allows sending an ICMPv6 echo request packet with ID=0 and receiving a response from a firewall-protected system that should block such packets. Finally, the thesis presents a specialized differential fuzzer designed for remote testing of IPv6 implementations on different operating platforms. This tool evaluates how different operating systems handle IPv6 packet manipulation of the header, identifying any difference between their reply that may be caused by non-compliance with one or more RFC standards. To the best of our knowledge, this is the first time differential fuzzing has been applied to network protocols, especially IPv6 and IPv6 extension headers.

Contents

Listing of figures

Listing of tables

Listing of acronyms

RFC . . . . . Request for Comments

OS . . . . . Operating System

IPv6 . . . . . Internet Protocol version 6

ICMPv6 . . . . . Internet Control Message Protocol version 6

IETF . . . . . Internet Engineering Task Force

MTU . . . . . Maximum Transmission Unit

PMTU . . . . . Path Maximum Transmission Unit

PMTUD . . . . . Path Maximum Transmission Unit Discovery

DoS . . . . . Denial-of-Service

IDS . . . . . Intrusion Detection System

DNS . . . . . Domain Name System

TCP . . . . . Transmission Control Protocol

UDP . . . . . User Datagram Protocol

NFS . . . . . Network File System

PF . . . . . Packet Filter

IP . . . . . Internet Protocol

QoS . . . . . Quality of Service

OSPF . . . . . Open Shortest Path First

GNS3 . . . . . Graphical Network Simulator-3

LXC . . . . . Linux Containers

KVM . . . . . Kernel-based Virtual Machine

LAN . . . . . Local Area Network

1 Introduction

The Internet Protocol version 6, or IPv6, is a fundamental component of the development of digital communication. IPv6 overcomes the limitations of IPv4 with its larger address space and enhanced features, providing a scalable and long-lasting response to the escalating needs of the modern, networked world. Due to the shortage of IPv4 addresses and the constantly growing network landscape, IPv6 usage is increasing over time [4]. Even though this change is necessary, it also presents new difficulties that should be carefully considered, especially in the area of security.

One important aspect of IPv6 lies in its Extension Headers, positioned between the header and the upper-layer protocol header. These Extension headers provide additional information, influencing the behavior of routers and endpoints. Each Extension Header serves a specific function, and adherence to specific rules and norms is crucial in their packet creation. The Internet Engineering Task Force (IETF), as outlined in RFC 7045 [5], oversees the standards and specifications of Extension Headers.

Another important aspect of IPv6 is fragmentation, which allows communication between networks that have different Maximum Transmission Units (MTUs). The MTU represents the maximum size of a data unit that can be transmitted over a network connection. Without fragmentation, a packet that encounters a network with a lower MTU is dropped, compromising the correct function of communication. However, while fragmentation is essential for transmitting packets between different MTUs, on the other hand, it involves greater processing power and a greater amount of resources, which can add overhead to the network causing

greater latency and potential packet loss. For this reason, routers do not have to perform fragmentation and reassembly operations, but the task is delegated to the endpoints. This shift reduces processing overhead in routers, but it introduces new challenges and considerations, such as the management of overlapping fragments, which are fragmentation packets with overlapping offsets. Although they do not appear in regular communication, such overlapping fragments are made possible by how the fragment offset is specified, and they may be exploited by malicious actors.

As a part of our research, our goal is to identify how different operating systems handle packets, study their behavior in accordance with IETF guidelines, and analyze the methods used to reassemble the complete packet. We will also focus on a specific type of fragmented packet, known as atomic fragments. By conducting this analysis, we hope to gain a comprehensive understanding of IPv6 packet fragmentation.

In light of all these problems surrounding IPv6, and the discovery of new ones during the research, as in the case of "ICMPv6 zero", we recognized the importance of thoroughly examining and understanding the potential vulnerabilities of this protocol in different operating systems. To achieve this, we created a differential fuzzer, a tool that enables us to systematically test multiple operating systems and analyze their responses to a variety of packets that manipulate Extension Headers and other parameters. In this way, we were able to quickly identify the various differences in packet management between the different systems, and we marked a possible starting point for future work aimed at fortifying IPv6 network security.

We make the following contributions in this paper:

- We performed tests on a large number of operating systems using a model that sends various combinations of overlapping fragments to determine if they are compliant with the RFC 5722 standard. We then compiled a list of compliant operating systems, and also investigated the use of time exceed packets to improve fingerprinting;
- We tested several firewall-protected operating systems to validate atomic fragment vulnerabilities, providing a list of vulnerable operating systems.
- We discovered a new bug in some operating systems, which we named ICMPv6 Zero. We provided a list of vulnerable operating systems and an in-depth analysis of this bug and its security implications;
- We developed a differential fuzzer tool capable of remotely testing the response of various operating systems to a diverse range of packets with manipulated headers. This tool enables us to analyze how different operating systems handle packet manipulation comprehensively and to identify any mismatch that might lead to fingerprinting or, in

some cases, even vulnerabilities when different operating systems are present in the same network segment.

1.1 Thesis structure

Starting with Section 2, we will look in detail at several aspects of IPv6 that are useful for getting a complete picture of the thesis. We will examine fragmentation in detail, analyzing "overlapping fragments" and "atomic fragments", and then move on to describe what the "differential fuzzing" technique is. We conclude with an illustration of the tools and languages used during the project.

The 3 section will follow, where we will delve into the tests that were performed. We will start with the tests on the "Overlapping Fragments", followed by the tests on the "Atomic Fragment". We will continue with tests on the "ICMPv6 zero" bug, and finally give a detailed discussion on the operation of the "differential fuzzer".

In the 4 section, we will present the results obtained and examine them in detail, discussing their implications and addressing the limitations found.

The thesis will end with the 5 section, where we will provide a general overview and a discussion of possible future developments that may arise from the topics covered in the thesis.

1.2 Related Works

Over the past decade, many comprehensive survey papers on IPv6 security have been published, covering a wide range of topics. This document focuses on the upper layer of IPv6 and explores its security implications.

Fragmentation issues are not new; they have been exploited for various attacks, including Denial-of-Service (DoS) [6], IDS/firewall evasion [1], and operating system fingerprinting [2]. As a result, various IPv6 specifications have been revised to address some IP fragmentation issues, with notable RFCs such as 3128 [7], 5722 [8], 6946 [9], and 8200 [10] providing specific fixes. In addition, RFC 9099 [11] emphasizes the proper handling of Extension headers to prevent stateless filtering from being bypassed. However, as highlighted in RFC 8900 [12], the problem persists. A particular vulnerability that our work will address is that of overlapping fragments, which has been exploited in the past to perform several attacks [13] [7] [8]. Early

studies by Atlansis [2] and by Shankar and Paxson [1] were among the first to create a comprehensive model to address this problem. Their research included various combinations of overlapping fragments to evaluate different reassembly strategies used by different operating systems and how overlapping fragments could be exploited for evasion attacks. However, their approaches are considered outdated for modern operating systems. In 2023, Di Paolo et al. [3] proposed a new model for testing IPv6 fragment handling, which we adopted and extended in our experiments.

Atomic fragments, introduced in RFC 2460 [14], are generated in response to IPv6 packets sent to an IPv4 destination. These packets have been leveraged for fragmentation-based attacks, as highlighted in studies such as [15] and [16]. While solutions have been proposed to mitigate these problems, such as processing packets in isolation [9] or eliminating the need to create them [17], atomic fragments are still exploited today. In this paper, we will show how they can be used to bypass firewalls in certain operating systems.

In 2012, Atlantis conducted a detailed study on IPv6 Extension Headers to investigate whether their abuse could lead to significant security impacts [18]. The study revealed vulnerabilities in certain operating systems that were not predicted by the corresponding RFCs. However, it is important to note that the study's results may no longer be considered valid due to the amount of time that has passed since then and the updates that the systems have undergone. More recent studies have been conducted on the same topic of Extension headers. For instance, in 2020, a network steganography technique was proposed to hide secret data within IPv6 packets with zero or more (up to four) Extension headers [19] and carry it from one end to other ends over a network. In 2021, a defendable security model was proposed against IPv6 Extension headers denial-of-service attack [20]. All these studies highlight the importance of developing a differential fuzzer to investigate all the issues related to this subject.

Regarding IPv6 fuzzers, there is only a limited amount of research on IPv6 fuzzers. Previous work includes a fuzzing framework proposed in 2010 [21] that uses a machine learning approach based on reinforcement learning, and some fuzz attacks implemented in 2013 to test DNS node vulnerabilities in IPv6 [22]. A recent study by Ilja van Sprundel focused on creating a fuzzer for the TCP/IP stack [23].

It's important to acknowledge the progress made with AFLNET [24], which is a significant achievement in the field of protocol fuzzing. AFLNET is uniquely capable of fuzzing protocol implementations without the need for explicit protocol specifications or message grammar. By utilizing a mutational approach and state feedback, AFLNET can guide the fuzzing process, which increases code coverage and explores different states within the protocol implementation.

However, AFLNET's focus is primarily on protocol-level fuzzing and does not specifically target IPv6 implementations or address the nuances of IPv6 protocol handling.

In contrast, our fuzzer takes a specialized approach by concentrating on IPv6 protocol implementations. While AFLNET may be applied to a wide range of protocols, our fuzzer is tailored specifically for testing IPv6 implementations across various operating systems. This tool focuses on sending a variety of IPv6 packets with modified headers, including Extension headers, to assess compliance with RFC standards and uncover potential vulnerabilities unique to IPv6.

Unlike traditional fuzzers that typically target individual functions or components within an application, our differential fuzzer operates remotely, allowing us to conduct comprehensive tests across entire network stacks. This approach provides us with valuable insights into how operating systems respond to anomalous packets and how they handle complex network scenarios. The remote nature of our fuzzer introduces additional complexities, such as managing system and network states, rotating IP addresses, and handling network protocol behaviors. However, these challenges are essential for ensuring that our tests accurately reflect real-world network conditions and interactions.

2
Background

In this chapter, we will explain the key elements of IPv6 to help understand the work presented in this thesis. We will describe the structure of IPv6 and how fragmentation works. In addition, we will discuss overlapping fragments, describing the work of Shankar & Paxson and Atlantis. We will also define atomic fragments. Finally, we will introduce the concept of differential fuzzer and give a brief overview of the tools we use.

2.1 IPv6

Internet Protocol version 6, IPv6, is the latest version of the Internet Protocol created by the IETF to address the long-anticipated problem of IPv4 address exhaustion. Every endpoint on the Internet requires an IP address to make a point-to-point connection, and since 2011 there have been many more devices connected to the Internet than there are IP addresses available [25] [26]. Although technologies such as NAT [27] have provided temporary relief from this problem, the transition to IPv6 is becoming increasingly necessary to meet the growing demands of networked devices and to ensure the continued expansion of digital connectivity.

2.1.1 IPv6 Structure

A packet is a basic unit of data transmitted over an IP network. It encapsulates both the data to be transmitted and the control information needed for routing and delivery. Each packet

consists of two main components: the header and the payload. The header contains essential control information required for routing and delivery, including source and destination addresses, packet sequencing, error checking, and other metadata. The payload carries the actual data to be transmitted, such as a segment of a file, a Web page, or other information intended for communication between networked devices.

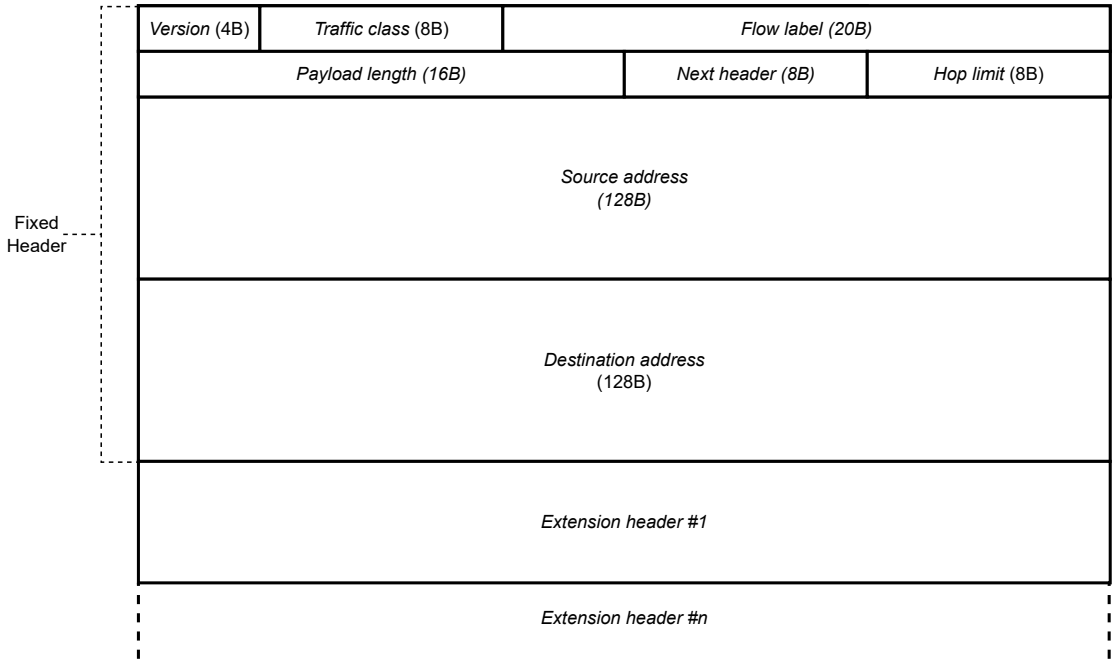


Figure 2.1: The IPv6 packet header is divided into two parts. The first part is a fixed header consisting of Version (4 bytes), Traffic class (8 bytes), Flow label (20 bytes), Payload length (16 bytes), Next header (8 bytes), and Hop limit (8 bytes). The second part is made up of all the Extension headers.

The header of an IPv6 packet [Figure 2.1] is divided into two parts: a mandatory fixed header and optional Extension headers. The fixed header is located at the beginning of each IPv6 packet and is 40 bytes in size. This design provides greater efficiency and clarity, making packet management on the network much easier. The fixed header contains 8 data fields that help with efficient routing, Quality of Service (QoS) management, and security. One of the most important data fields for our work is the Next header. It is used to indicate the type of transport layer protocol used in the payload and also indicates which Extension header, if any, was used.

An important feature of IPv6 is the support for Extension headers. They were introduced to provide backward-compatible and optional support for additional features and advanced options in the IPv6 packet. This allows the IPv6 design to be extensible and adaptable to future needs. The following is a list of the Extension headers that we will use throughout this article:

- **Hop-by-Hop Options Header (0):** Contains options to be examined by each router along the path;
- **Routing Header (43):** Used to specify particular paths for packets to follow through the network;
- **Fragment Header (44):** Used for packet fragmentation;
- **Destination Options Header (60):** Similar to the Hop-by-Hop Options header, but contains options that should only be examined by the destination node.

2.1.2 Path Maximum Transmission Unit

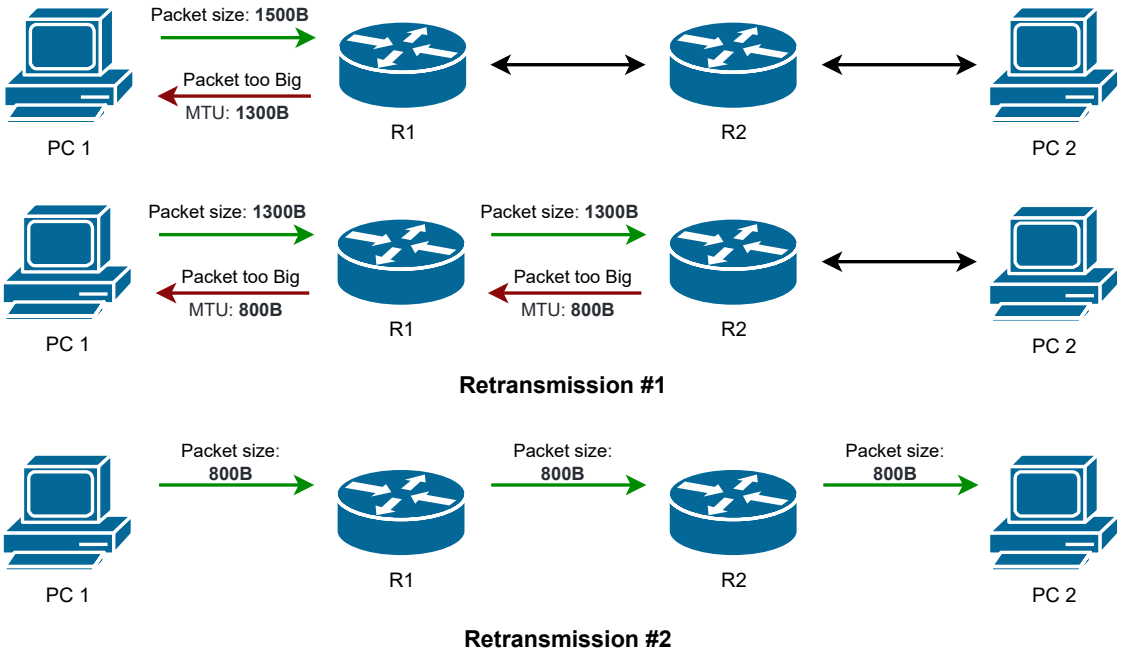


Figure 2.2: Path MTU Discovery starts by setting the initial PMTU to the first hop's MTU (1500 byte). If packets are dropped due to size, a "Packet Too Big" message is returned. PMTU is reduced based on the MTU of the restricting hop, as reported in the message. The process ends when the estimated PMTU is less than or equal to the actual PMTU (800-byte).

An Internet path is the route that connects a source node to a destination node. This path contains routers connected via different links, possibly of different technologies, such as fiber optic, wireless, and copper cables. A packet that goes from a source to a destination traverses

these links one after the other; each router sends the packet to the next hop in the path, according to its forwarding rules. Routers may forward a packet to a path that uses a different technology; if so, depending on the underlying technology, the MTU, Maximum Transfer Unit, may change. The MTU limits the amount of bytes a link can carry in a single unit of transmission. IPv6 requires that each link support an MTU of 1280 octets or greater [10], known as the IPv6 minimum link MTUs. For any given path, the path MTU (PMTU) is equal to the smallest of its link MTUs.

IPv6 nodes typically implement path MTU Discovery to discover and use paths with PMTUs greater than the IPv6 minimum link MTU. However, a minimal IPv6 implementation may choose not to implement PMTU discovery. In this case, nodes must use the IPv6 minimum link MTU as the maximum packet size. This often results in using smaller packets than necessary, wasting network resources, because most paths have a PMTU greater than the IPv6 minimum link MTU.

Path MTU Discovery (PMTUD) execution [28] starts with the source node generating an initial packet with maximum sizes equal to the MTU of the link towards the first hop in the path. If the packet sent along this path is too large to be forwarded to the next hop by any of the routers in the path, the router that is unable to forward the packet will drop it and return an ICMPv6 "Packet Too Big" message. Upon receiving such a message, the source node reduces its assumed PMTU for the path based on the MTU of the constricting hop as reported in the "Packet Too Big" message [Figure 2.2]. The path MTU Discovery process ends when the source node's estimated PMTU is less than or equal to the actual PMTU.

2.1.3 Fragmentation

IPv6 fragmentation occurs at the source device when the size of a packet exceeds the path MTU. To allow the transmission, the source node splits the packet into fragments and sends each fragment as a separate packet. The receiver then reassembles the fragments into the original packet [10].

The original packet [Figure 2.3] consists of three parts:

- The **Per-Fragment headers** must include the IPv6 header and all relevant Extension headers that need to be processed by nodes during transit to the destination (including Routing and Hop-by-Hop header);
- The **Extension & Upper-Layer headers** refer to all other headers that are not included in the Per-Fragment headers section of the packet;

Figure 2.3: The original packet consists of three parts: Per-Fragment headers, Extension & Upper-Layer headers, and the Fragmentable Part. By the standard, only the last part of the original packet can be fragmented, while headers and extensions cannot.

- The **Fragmentable Part** consists of the rest of the packet after the Upper-Layer header or after any header that contains a Next header value of No Next header.

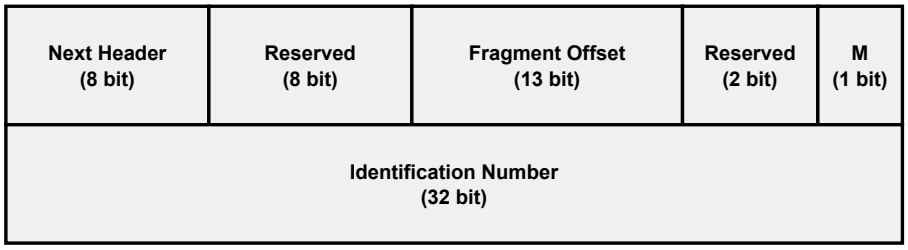


Figure 2.4: Representation of the Fragmentation header consisting of Next header, Fragment Offset, More Fragment flag (M), and the Identification Number.

To ensure that the fragments are reassembled correctly, each fragment contains additional information that allows the receiving device to reconstruct the original packet. For example, when a packet needs to be fragmented, an IPv6 Fragmentation Header is added that contains important details for the fragmentation process, such as:

- **Next header value:** Identifies the type of header following the Fragmentation header;
- **Reserved (8 bit):** Reserved field initialized to 0 for transmission and ignored on reception;
- **Fragment Offset:** indicates the position of the fragment relative to the beginning of the fragmentable part of the original packet;
- **Reserved (2 bit):** Reserved field initialized to 0 for transmission and ignored on reception;
- **More Fragments flag:** a field that tells us if there are more fragments after this one. It takes the value 0 if the fragment is the last one, and 1 if there are more fragments;

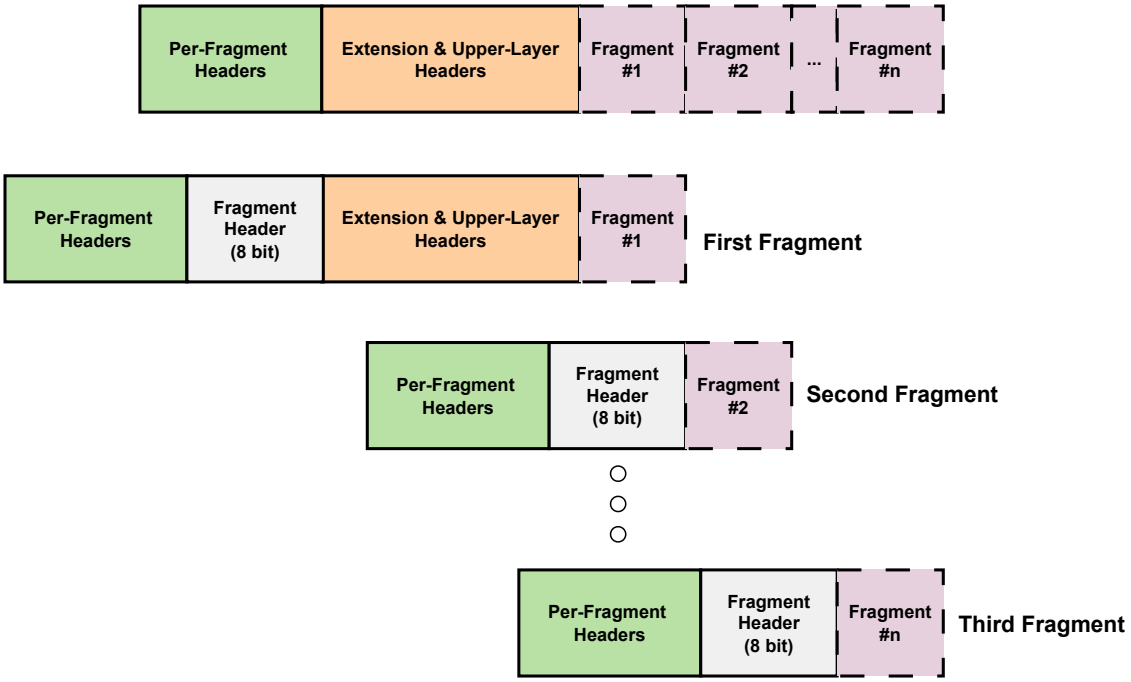


Figure 2.5: Fragmentation process where the original packet is split into different packets. All the packets are divided into the Per-Fragment Headers, the Fragment Header, and the Fragment, only the first packet contains all the extension & Upper-Layer headers.

- **Identification number:** the identification value created by the source node.

When the fragmentation occurs, the fragmentable part of a packet is divided into fragments with a length that fits the path MTU. The first fragment packet contains the Per-Fragment header with the payload length equal to the length of this fragment and the Next header field of the last header equal to 44. It also contains a Fragment header with the fragment offset equal to 0 and the M flag equal to 1, followed by all extension and Upper-Layer headers, and at the end, the first fragment of the fragmentable part. The subsequent fragments are similar to the first one, except for the Fragment header, which offset indicates the position of the fragment, and the M flag is equal to 1 for all fragments except for the rightmost fragment, which will be equal to 0. There are no extension and Upper-Layer headers, and each of them contains a fragment [Figure 2.5].

When reassembling a fragmented packet, all fragment packets with the same source address, destination address, and fragment identification are combined to reassemble the original packet.

The Per-Fragment headers of the reassembled packet consist of all the headers up to, but not including, the Fragment header of the first fragment packet. The Next header field of the last

header is the Next header field of the first fragment's Fragment header. The payload length of the reassembled packet is calculated from the length of the headers and the length and offset of the last fragment.

The fragmentable portion of the reassembled packet consists of the fragments following the Fragment headers in each of the fragment packets. The length of each fragment is calculated by subtracting the length of the headers between the IPv6 header and the fragment itself from the payload length of the packet. The relative position of each fragment in the fragmentable part is determined by its fragment offset value. The Fragment header is not present in the final reassembled packet.

If a fragment is an entire datagram (i.e., both the Fragment Offset field and the M flag are zero), it requires no further reassembly and should be processed as a fully reassembled packet. Any other fragments that match this packet (i.e., have the same IPv6 Source Address, IPv6 Destination Address, and Fragment Identification) should be processed independently.

IPv6 fragmentation, while critical for some applications including DNS [29], OSPFv2 [30], OSPFv3 [31], packet-in-packet encapsulation, is usually considered fragile and therefore discouraged [12] [10].

2.1.4 Overlapping Fragments

Overlapping fragments represent a situation where two fragments of a fragmented IPv6 packet overlap. More technically, this occurs when the offset field of a fragment does not start immediately after the end of the previous fragment, but overlaps with it. It's important to note that in a properly fragmented IPv6 packet, overlapping fragments should not occur. Their presence typically indicates packet construction errors or potentially malicious behavior.

The RFC 5722 states that, in the presence of overlapping fragments in a packet, reassembly of that packet must be abandoned, and all fragments received for that packet must be silently discarded. Nevertheless, each operating system has a personal fragment reassembly policy, which may cause them to behave differently when faced with anomalies such as overlapping fragments.

Shankar & Paxson model

There are several ways to evaluate IP fragmentation reassembly techniques. One such technique is the Shankar and Paxson model [1], which includes six unique combinations of fragments that overlap and overwrite each other with different offsets and lengths [Figure 2.6].

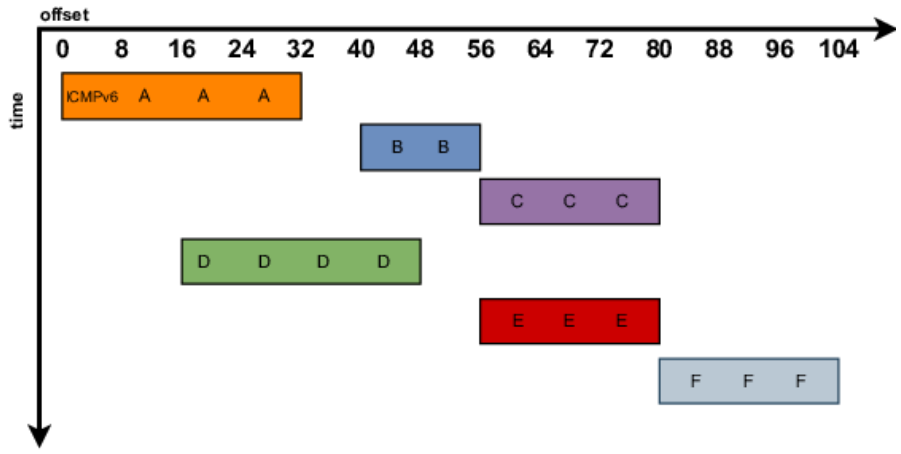


Figure 2.6: Shankar and Paxson model: each block in the sequence represents a fragment. The payload of each fragment is a sequence of bytes that encode a character. There is an 8-byte gap between the first block (orange) and the second block (blue). The fourth fragment (green) partially overlaps the first and second fragments, while the fifth fragment (red) completely overlaps the third fragment (purple) [1]

These combinations can be divided into the following groups:

- At least one fragment completely overlapped by a subsequent fragment with identical offset and length;
- At least one fragment partially overlapped by a subsequent fragment with a greater offset than the original;
- At least one fragment partially overlapped by a subsequent fragment with a smaller offset than the original.

The original paper that introduced this model also identified five fragment reassembly methods:

1. **BSD:** It prioritizes an original fragment, except when the subsequent segment begins before the original segment;
2. **BSD-right:** It prioritizes the succeeding segment except when the original segment ends after the succeeding segment or starts before the original segment and ends at the same time, or ends after the original segment;
3. **Linux:** Gives priority to the succeeding segment, except when the original segment begins before the succeeding segment or the original segment begins at the same time as and ends after the succeeding segment;

4. **First:** It favors the original fragment;

5. **Last:** It favors the following fragment.

However, this model is now obsolete because modern operating systems either reassemble the entire fragment or discard it entirely, as shown later in Section 3.2.

Atlantis model

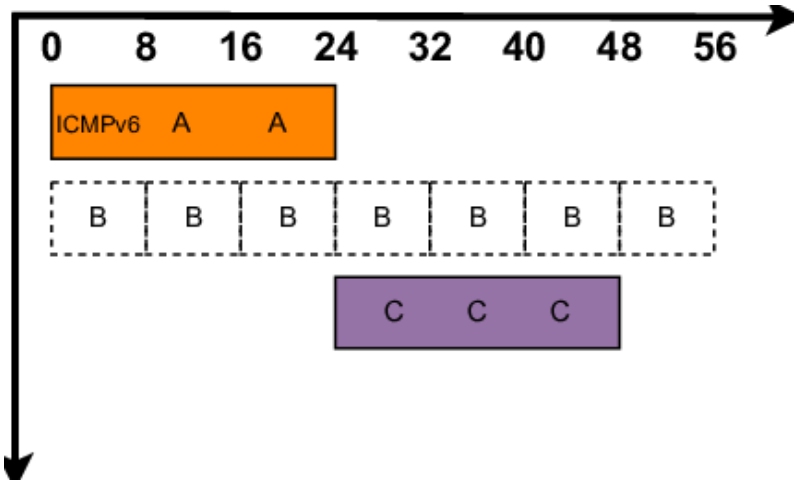


Figure 2.7: Atlantis model: each block in the sequence represents a fragment. The payload of each fragment is a sequence of bytes that encode a character. The first fragment is always 24 bytes long, consisting of an 8-byte ICMPv6 Echo Request and a 16-byte payload. The second fragment has a length of 56 bytes and completely overlaps with the first and third fragments. Finally, the third fragment is always 24 bytes long and has an offset of 24 [2].

Atlasis introduced another method that involves running multiple tests with only three fragments [2]. The overlapping fragment's offset, size, and M flag (which stands for "more fragment" flag) are varied in each test [Figure 2.7]. Here's how the model is defined:

- The first fragment has an offset of zero, a constant length, and carries the ICMPv6 header and part of the payload. The M flag is always set to 1;

- The third (last) fragment has a constant length, carries part of the payload, and the M flag is always set to 0. Each offset is equal to the size of the first fragment;

- The second fragment has a variable length and offset. For each test, the M flag is set to either 0 or 1.

This model has been successful in examining the reassembly methods of various operating systems. However, for the same reason as the Shankar and Paxson model, it is now obsolete.

In the model that we will use in our tests, which we will discuss in more detail in the 3.2 section, the fragments will have different arrival times on each test, but the same offset and content, so that we have a combination where packet reassembly can be done using or discarding entire fragments.

Attacks that exploit overlapping fragments

IPv6 fragmentation-based attacks take advantage of the IP fragmentation and reassembly policy by manipulating fragment parameters such as offsets, sizes, and payloads. These attacks aim to evade security controls, prevent servers from functioning as intended, or abuse servers' resources. Most IPv6 fragmentation-based attacks are adaptations of the earlier IPv4 fragmentation attacks. A classification of IPv6 fragmentation-based attacks is based on overlapping fragments.

The overlapping attack exploits the vulnerability of two fragments overlapping to bypass a firewall's secure policy [12]. Suppose we have a packet that is split into two fragments. The first fragment, which conforms to the local security policy, consists of an IP header, a Transport layer header, and some payload, and can pass through a stateless firewall. The second fragment, which overlaps the first fragment, also bypasses the firewall. When the packet is reassembled, the transport-layer header from the first fragment is overwritten by data from the second fragment, creating a packet that doesn't comply with the local security policy.

In the event of an attack, a stateless firewall cannot defend itself. Therefore, the responsibility falls to the destination nodes, which can handle the situation through their reassembly policies. If the destination nodes successfully detect overlapping fragments according to RFC 5722, they should discard them immediately.

Additionally, fragment reassembly is a stateful process within an otherwise stateless protocol. This means that it requires the storage of state information during processing. This can be exploited in resource exhaustion attacks, where an attacker sends a series of fragmented packets, each with a missing fragment, making the reassembly process impossible. This attack can cause resource exhaustion on the target node, potentially denying reassembly services to other flows. To make it easier for an attacker to forge malicious IP fragments, some implementations set the Identification field to a predictable value. A potential solution to this problem is to flush the fragment reassembly buffers. However, this approach may also cause legitimate packets to be dropped.

2.1.5 Atomic Fragment

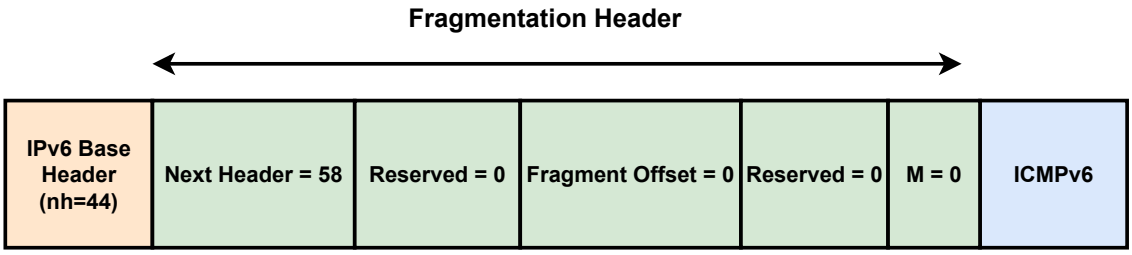


Figure 2.8: Representation of an atomic fragment, where the IPv6 Base Header Next header points to the Fragmentation header (nh=44) whose More Fragment flag (M) is zero.

In IPv6, the specification allows packets to contain a Fragment header without actually fragmenting the packet into multiple pieces. These packets, called "atomic fragments", are typically generated by hosts that have received an ICMPv6 "Packet Too Big" error message indicating a next-hop MTU less than 1280 bytes [17]. Specifically, an atomic fragment is an IPv6 packet with a Fragmentation header where the fragment offset is 0 and the "more fragment" bit is also 0 [Figure 2.8].

As reported in RFC 8021 [17] and RFC 6946 [9], the generation of IPv6 atomic fragments is considered harmful due to the security implications and interoperability issues associated with them. An attacker can manipulate ICMPv6 "Packet Too Big" error messages to trick hosts into using atomic fragments. Subsequently, these fragments may be processed by some implementations as normal "fragmented traffic", leading to potential vulnerabilities such as fragmentation-based attacks [32]. To address these issues, recent updates to RFC 2460[14] and RFC 5722[8] have been proposed. These updates ensure that IPv6 atomic fragments are processed independently of other fragments, eliminating the identified attack vector.

2.2 Differential Fuzzing

A fuzzer is a tool used in computer and software security to find and exploit security flaws or bugs in software applications. It does this by testing them with various inputs and examining their responses for vulnerabilities or unexpected behavior. In the case of network protocols such as IPv6, a fuzzer can help identify how systems respond to different packet structures and configurations.

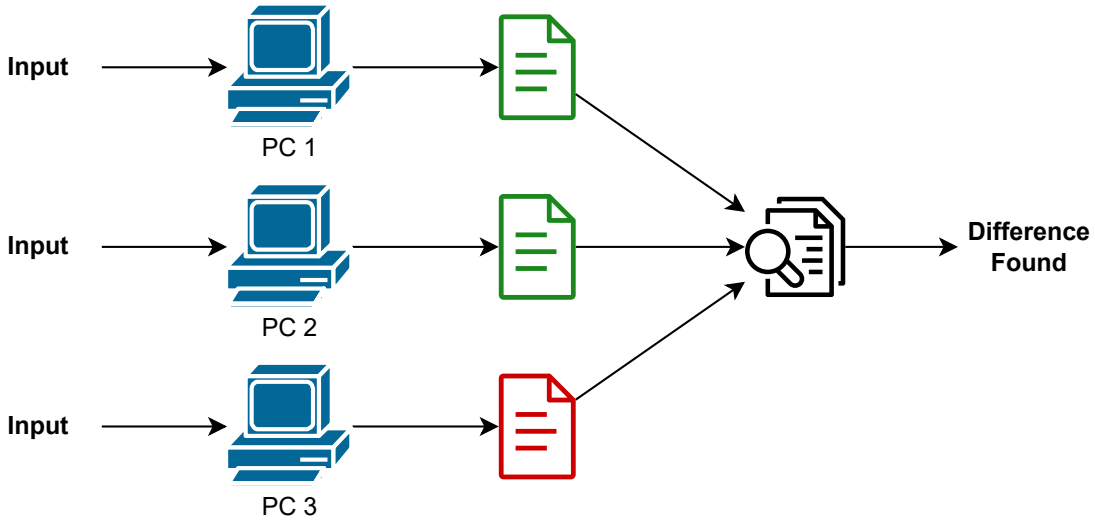


Figure 2.9: Illustration of a Differential Fuzzer in action. The fuzzer sends the same input to different operating systems, which should ideally provide the same response. Any discrepancies in the responses are analyzed and reported.

The primary goal of a fuzzer is to detect any weaknesses or vulnerabilities in a system by providing a variety of inputs and observing the responses. By generating a wide range of test cases, a fuzzer can uncover security problems, implementation errors, or unexpected behavior that may not be apparent under normal operating conditions.

Differential fuzzing is an advanced testing technique that compares the responses of multiple systems or implementations to the same set of input variations. This approach helps to identify differences in the behavior of different software or devices when exposed to similar stimuli.

In the context of IPv6 testing, a differential fuzzer is particularly useful. It allows researchers to examine how different operating systems handle the same set of packets, especially those with complex structures such as Extension headers and atomic fragments.

However, this tool can also provide adversaries valuable insights into the atypical behavior of different systems, enabling techniques such as fingerprint attacks. These attacks use the differences in system behavior to distinguish between various software implementations or versions. Through the analysis of a target system's responses to carefully crafted inputs, an attacker can obtain valuable information about the system's underlying architecture, configuration, or even the presence of specific vulnerabilities.

It is important to note that any discrepancies in system behavior can also serve as a launch pad for other types of attacks. Variations in reassembly policies across different systems can lead to divergent reassembly of packet payloads by intrusion detection systems (IDS), potentially

bypassing their detection mechanisms and facilitating malicious activities.

2.3 Tools

During the research, several tools were used to facilitate data development, management, and analysis. The main tools used in the project are listed below:

- **Oracle VM VirtualBox:** open source virtualization software that allows virtual machines to run on different operating systems;
- **Vagrant:** open source software that supports and simplifies the management of repeatable virtual machine environments. It uses Virtual Box as the underlying provider for the virtualization process;
- **Graphical Network Simulator-3 (GNS3):** open source software that allows users to simulate complex networks by integrating real network devices and running them in a virtual environment. It is commonly used for network engineering and training purposes.
- **Proxmox Virtual Environment:** an open-source virtualization platform based on Debian that integrates container-based virtualization (LXC) and hardware-based virtualization (KVM) technologies;
- **tcpdump:** a command-line packet analyzer that captures and displays network traffic. It is compatible with most Unix-like operating systems;
- **Wireshark:** a network protocol analyzer that allows analysis of network packets as they are captured and has an easy-to-use graphical interface with advanced sorting and filtering capabilities. It can also be used to examine tcpdump-generated files.

2.4 Languages and Libraries

Python was used for all programs created, with Scapy serving as the primary library:

- **Python:** an object-oriented, high-level, general-purpose, and interpreted programming language that served as the primary language for script development and results analysis;
- **Scapy:** a powerful interactive packet manipulation library written in Python, specifically designed for creating, sending, and receiving network packets of a wide range of protocols.

3 Methodology

In this section, we'll give a detailed explanation of how we conducted our tests. This includes their goals and development process. We will then present the results in the 4 section.

3.1 Experimental Setup

Figure 3.1: Network diagram of our initial setup, with one operating system acting as the attacker connected to another operating system acting as the victim. The diagram displays both systems connected to the same broadcast network

All of our experiments were conducted with a Linux-based machine acting as the attacker, while a variety of operating systems were used as the victims. The attacker's operating system did not affect the test results because we manually created packets using Scapy, bypassing the kernel code responsible for fragmentation and reassembly policies on the attacker's machine. Therefore, we had complete control over the flags and contents of individual fragments, making the attacker's operating system irrelevant to the experimental process. To ensure the relia-

bility and accuracy of the tests, we repeated each experiment multiple times. The list of victim operating systems is given in the table 4.2.

Operating System	Kernel Version	Vagrant box version
GNU/Linux Arch	6.7.9-arch1-1	20240315.221711
GNU/Linux Debian 12	6.1.0-18-amd64	20240319
GNU/Linux Ubuntu 23.04	5.15.0-91-generic	4.3.12
NetBSD	9.3 (GENERIC)	4.3.12
OpenBSD	6.9 GENERIC.MP#473	4.2.16
FreeBSD	13.1-RELEASE	4.2.16
FreeBSD	13.2-RELEASE	4.3.12
FreeBSD	14.0-RELEASE	4.3.12
MikroTik	6.49.13 (stable)	6.49.13
MikroTik	7.14.1 (stable)	7.14.1
Windows 10	10.0.19045 N/A Build 19045	20240201.01
Windows 11	10.0.22621.1702	2202.0.2305
MacOS Darwin	19.0.0	Physical Hardware
MacOS Darwin	21.6.0	Physical Hardware
Android (MIUI)	4.14.180-perf-g5d6f377	Physical Device
iOS	17.2.1	Physical Device
Cisco IOS router	15.2(4)M7	GNS3 Simulation

Table 3.1: Operating systems under testing along with their respective Kernel versions. Most operating systems were tested using Vagrant virtual machines, while MacOS Darwin versions 19.0.0 and 21.6.0 were tested on physical hardware. Android (POCO Huawei variant) and iOS were tested on physical devices. Cisco IOS 15 Router was tested using GNS3 simulation.

During our testing process, we encountered some challenges that required us to adapt our infrastructure. While we initially started testing with Vagrant, we soon discovered that certain virtual machines had compatibility issues and required specific configurations. To address this, we decided to explore other options and used GNS3 and Proxmox Virtual Environment for certain tests.

After careful evaluation, we found Proxmox Virtual Environment to be highly efficient and convenient for accessing and configuring virtual machines. As a result, we decided to move completely to Proxmox. However, during our testing process, we noticed discrepancies between the tests performed on Proxmox and Vagrant. Upon investigation, we found that Proxmox had some limitations that interfered with our testing. To ensure accurate results, we had to revert to Vagrant and GNS3.

To eliminate any potential side effects caused by other network elements during the experi-

ments, we set up a direct LAN connection between the attacker and the victim. This configuration applies to all operating systems except Google Android (MIUI variant) and Apple iOS, for which we used two physical devices.

For more information on the limitations we encountered during testing, refer to the 4.5 section.

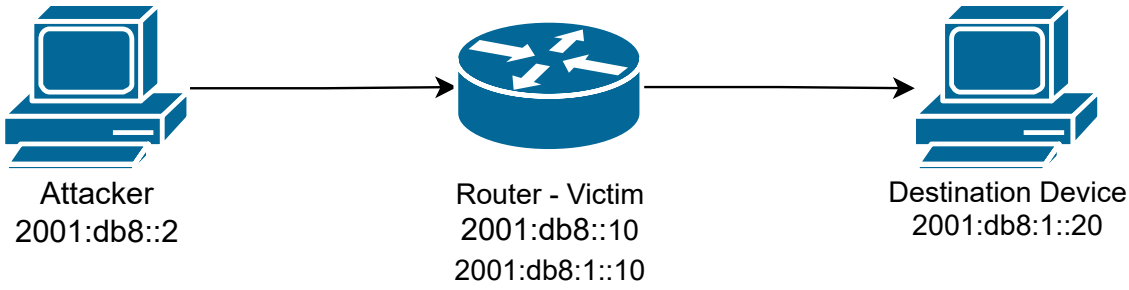


Figure 3.2: Network diagram of our second setup, with one operating system acting as the attacker, connected to another operating system acting as a router. The router is also a victim of the tests and is used to access a third machine. The diagram displays all three systems connected to the same broadcast network.

Additionally, we performed the same tests in a slightly different setup, where the victim is acting as a router, and the final destination for packets is a third machine behind the victim. This experiment aimed to determine if there are variations in the way that various operating systems handle packets when functioning as intermediate devices.

3.2 Overlapping Fragments

This test aims to provide the same information as the one offered by Shankar & Paxson, and Atlasis works. However, instead of relying on their work, we have chosen to use the approach proposed by Di Paolo et al. [3], expanding the range of operating systems tested. The reason for this decision is that the policy for discarding fragments has changed, making the work of Shankar, Paxson, and Atlasis obsolete.

This new model is based on the Shankar and Paxson model, in which all fragment offsets are reduced by one (8 bytes). This means that there will be some combinations where packet reassembly is done by using or discarding whole fragments, according to the reassembly policies of new operating systems.

The tests performed are divided into three types, each with 10 different permutations:

1. Single ICMPv6 packet fragmented using multiple permutations of fragments;

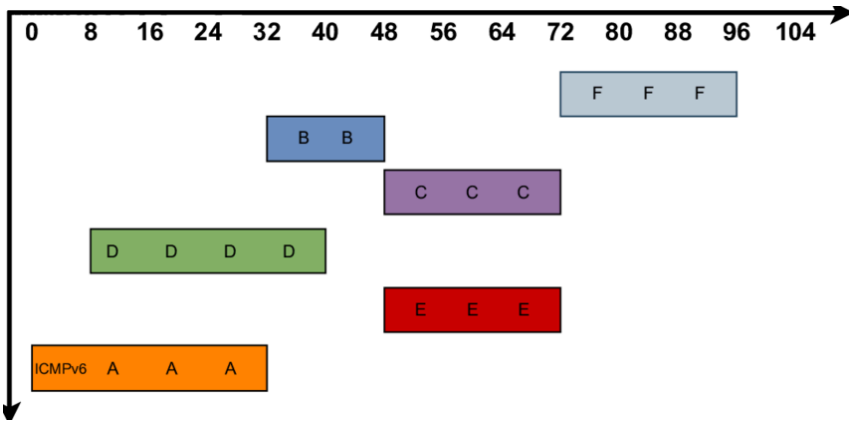


Figure 3.3: Example of a possible permutation of the new model. Each block in the sequence represents a fragment, the payload of each fragment is a sequence of bytes that encode a character. This permutation shows how the first fragment (orange) is the last to arrive.

2. Single ICMPv6 packet fragmented using multiple permutations, but fragments are sent multiple times;
3. Multiple ICMPv6 packets fragmented with multiple permutations.

It was decided to use the ICMPv6 protocol, and specifically the echo request type, for these tests. This protocol was chosen because it is the simplest one that allows us to receive a response. In addition, by assigning a unique payload to each fragment, we can use the response received to quickly identify which reassembly policies were used.

In the first test, a fragmented packet is sent to the victim for each permutation. The second test is the same, but all frames are sent four more times to simulate network retransmission or malicious action. In the third test, five different ICMPv6 packets are sent for each permutation. In the second test, all fragments have the same ID, while in the third test, the fragments are grouped by different IDs.

Once all the fragments of the transmission have reached their destination, the victim must reassemble them and verify that the checksum is correct. In IPv6, the header does not contain a checksum [10], and it is the responsibility of upper-layer protocols such as ICMPv6, UDP, and TCP to verify the integrity of the transmission. When a checksum is required, such as for ICMPv6, the first fragment contains the checksum of the entire packet within the Upper-Layer header. However, overlapping fragments can create a situation where the final checksum of the packet is incorrect because hosts may have different reassembly policies. This may cause the victim to discard the packet and not respond to our ICMPv6 ping tests, which could invalidate any results obtained so far.

To prevent this, the payload is constructed to generate the same checksum for each combination. This was achieved by taking advantage of the commutative property of the checksum [33], i.e. that the checksum contains the sum of 2-byte pairs of the entire IPv6 packet [Table 3.2].

Table 3.2: Payload definition proposed by Di Paolo et al. [3]: This payload exploits the checksum's commutative property to avoid re-assembly errors. The "odd" version is also used in tests with one packet, whereas both "odd" and "even" are used in tests with multiple packets.

Shankar and Paxson	Odd/single packet	Even
A	11223344	44113322
B	11332244	44331122
C	22113344	44332211
D	22331144	11224433
E	33112244	11334422
F	33221144	22114433

3.3 Atomic Fragments

The test involves sending a single atomic fragment with different payloads to a firewall-protected operating system. For this test, the atomic packet contains three IPv6 Fragmentation headers, which is a clear violation of the IPv6 specifications mentioned in RFC 9099 [11]. According to the specifications, an illegal repetition of headers, such as multiple Fragment headers must be dropped. The purpose of this test is to observe the response of an operating system when it encounters this particular type of fragment.

The firewall rules can be summarized as follows

- Block all ICMPv6 echo requests from the attacker;
- Pass every ICMPv6 neighbor solicitation;
- Pass every ICMPv6 neighbor advertise;
- Enable scrub and antispoof if possible.

Packet scrubbing is the process of normalizing packets to eliminate any ambiguity in their interpretation by the packet's ultimate destination. This is achieved through the use of the scrub directive, which also reassembles fragmented packets, protects certain operating systems from certain types of attacks, and drops TCP packets with invalid flag combinations. Although it is

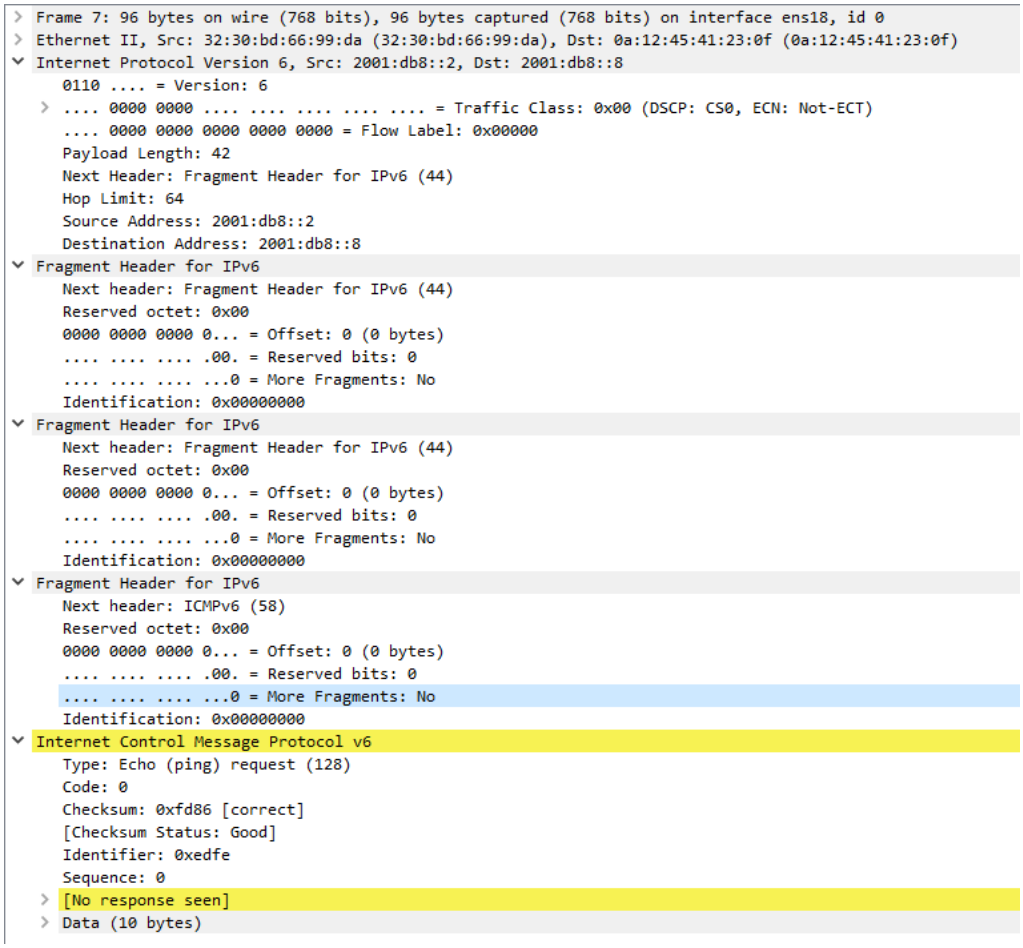


Figure 3.4: Example of an atomic fragment with three Fragment Headers. Each Fragment header, by definition, has an offset of 0 and an M flag equal to 0, indicating that there are no more fragments.

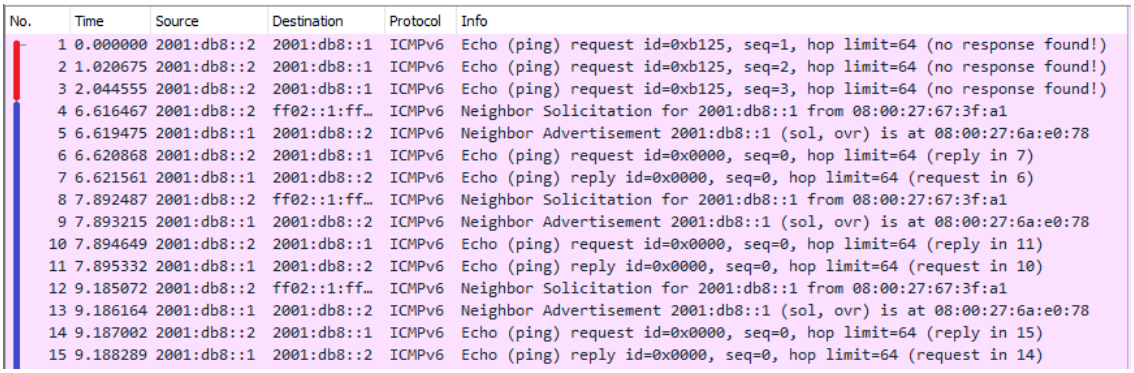
highly recommended to scrub all packets, there are some special situations where this should not be done. One such situation is when NFS is passed through PF on an interface. Some platforms send and expect strange packets, such as fragmented packets with the ”do not fragment” bit set, which are correctly rejected by scrubbing. Another reason to avoid scrubbing is that some multiplayer games experience connection problems when passing through PF with scrub enabled [34]. With this set of rules, we expected that all ICMPv6 traffic coming from the Attacker is blocked, in particular, Fragmented traffic should not pose a threat thanks to the scrub rule.

We tested the vulnerability several times using both Vagrant and Proxmox to make sure it was not caused by a virtual machine. However, during the first set of experiments, we discov-

ered another vulnerability that was corrupting the data. We called this vulnerability ”ICMPv6 zero” and it is explained in the next section. Because of this vulnerability, the second set of experiments was performed by sending multiple atomic fragments for each victim to evaluate the extent of its impact on the results.

3.4 ICMPv6 zero

ICMPv6 zero is a vulnerability that was discovered while testing the transmission of atomic fragments to a single operating system protected by a firewall. It was observed that after a certain number of packets had been dropped, the operating system began to accept and respond to the packets. As a result, various tests were performed to determine the cause of this behavior.



No.	Time	Source	Destination	Protocol	Info
1	0.000000	2001:db8::2	2001:db8::1	ICMPv6	Echo (ping) request id=0xb125, seq=1, hop limit=64 (no response found!)
2	1.020675	2001:db8::2	2001:db8::1	ICMPv6	Echo (ping) request id=0xb125, seq=2, hop limit=64 (no response found!)
3	2.044555	2001:db8::2	2001:db8::1	ICMPv6	Echo (ping) request id=0xb125, seq=3, hop limit=64 (no response found!)
4	6.616467	2001:db8::2	ff02::1:ff…	ICMPv6	Neighbor Solicitation for 2001:db8::1 from 08:00:27:67:3f:a1
5	6.619475	2001:db8::1	2001:db8::2	ICMPv6	Neighbor Advertisement 2001:db8::1 (sol, ovr) is at 08:00:27:6a:e0:78
6	6.620868	2001:db8::2	2001:db8::1	ICMPv6	Echo (ping) request id=0x0000, seq=0, hop limit=64 (reply in 7)
7	6.621561	2001:db8::1	2001:db8::2	ICMPv6	Echo (ping) reply id=0x0000, seq=0, hop limit=64 (request in 6)
8	7.892487	2001:db8::2	ff02::1:ff…	ICMPv6	Neighbor Solicitation for 2001:db8::1 from 08:00:27:67:3f:a1
9	7.893215	2001:db8::1	2001:db8::2	ICMPv6	Neighbor Advertisement 2001:db8::1 (sol, ovr) is at 08:00:27:6a:e0:78
10	7.894649	2001:db8::2	2001:db8::1	ICMPv6	Echo (ping) request id=0x0000, seq=0, hop limit=64 (reply in 11)
11	7.895332	2001:db8::1	2001:db8::2	ICMPv6	Echo (ping) reply id=0x0000, seq=0, hop limit=64 (request in 10)
12	9.185072	2001:db8::2	ff02::1:ff…	ICMPv6	Neighbor Solicitation for 2001:db8::1 from 08:00:27:67:3f:a1
13	9.186164	2001:db8::1	2001:db8::2	ICMPv6	Neighbor Advertisement 2001:db8::1 (sol, ovr) is at 08:00:27:6a:e0:78
14	9.187002	2001:db8::2	2001:db8::1	ICMPv6	Echo (ping) request id=0x0000, seq=0, hop limit=64 (reply in 15)
15	9.188289	2001:db8::1	2001:db8::2	ICMPv6	Echo (ping) reply id=0x0000, seq=0, hop limit=64 (request in 14)

Figure 3.5: Example of how different ICMPv6 packets behave on a Windows 10 system that has a firewall blocking ICMPv6 Echo Requests. The red line shows normal pings without any reply, while the blue line represents a Scapy Echo Request with all of its corresponding replies.

From Figure 3.5, we can see that the initial packets generated by the ”ping” terminal command are blocked by the firewall rules present in the victim’s Windows 10. The firewall rules allow the neighbor and advertisement packets to pass but block the echo requests. On the other hand, the echo requests made with Scapy are able to bypass the firewall. As we continued testing, we ruled out that the error was caused by a specific IP range or any other value of the packet, including sequence or payload. Therefore, we concluded that this vulnerability was caused by the ID, which Scapy sets to zero by default.

For this reason, we decided to run a series of tests using ICMPv6 with zero ID against a number of operating systems that are protected by firewalls and observe whether the packets were received or not. The firewall rules are the same as in the previous test:

- Block all ICMPv6 echo requests from the attacker;

(a) Normal ping

(b) ICMPv6 default Echo Request packet made by Scapy

Figure 3.6: Difference between a normal ping and an ICMPv6 packet made by Scapy. The normal ping with random Identifier, 1 as sequence number, and a 56-byte payload. The default ICMPv6 Echo Request packet has no payload, a sequence number of 0, and an ID number of 0. Both packets are sent to the same system protected by a firewall that blocks Echo Requests. As reported, the normal ping fails to receive any response.

- Pass every ICMPv6 neighbor solicitation;
- Pass every ICMPv6 neighbor advertise.

3.5 Differential Fuzzer

Noticing the large number of problems related to IPv6, we decided to define a methodology based on differential fuzzing to allow us to explore the differences between multiple operating systems. For this reason, we developed a differential fuzzer to help us with this operation.

The fuzzer was developed in Python using the Scapy library to handle all operations related to modifying, sending, and managing IPv6 packets. Currently, Scapy supports a limited number of Extension headers, only 4 of them can be used: Hop-by-Hop Options header, Routing header, Fragment header, and Destination Options header.

The fuzzer generates several IPv6 packets, mainly focused on modeled ICMPv6, with different types of Extension headers and payloads. These include:

- IPv6 packets with no Extension header or Next header, kept as small as possible;
- ICMPv6 packets with all possible combinations of Extension headers up to the maximum allowed by the MTU;

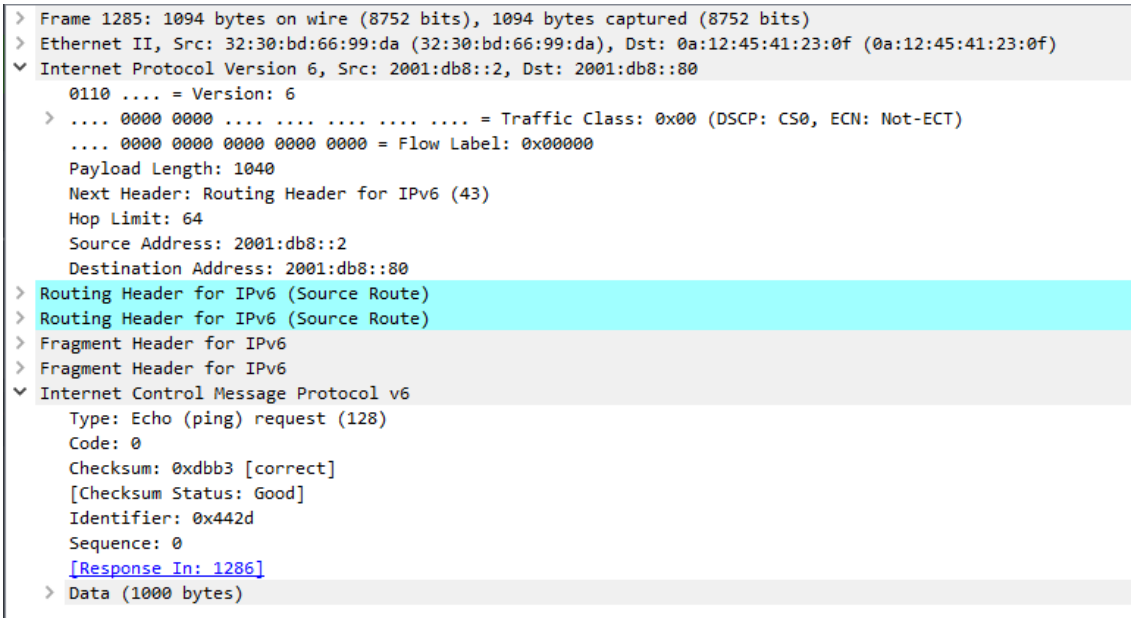


Figure 3.7: Example of a fuzzer-generated packet with fixed source address, two Routing headers, two Fragment headers, random ID, and sequence number equal to zero

- Packets with different or fixed source IP;
- ICMP IDs that are either different or fixed;
- Sequence numbers that are either fixed or zero;
- Packets with different payload sizes ranging from 0, 64, 128, and 1000 up to the maximum size of the MTU.

The fuzzer uses specific commands to send different types of IPv6 packets to a specific operating system, and thanks to tcpdump, the tool records the possible responses it gets in a .pcap file. To avoid manually analyzing all packets with Wireshark, we developed a program that analyzes multiple .pcap files from multiple operating systems and highlights the differences found.

Packet analysis is performed as follows:

- Packets in each file are filtered by removing unnecessary ones like Neighbor Solicitation, Neighbor Advertisement, or DNS packets if the source address is random;
- Packets from different files are grouped by operating system;
- ICMPv6 echo request packets are grouped by id (zero or random), sequence number (fixed or zero), source (fixed or random), type of Extension header present, and payload size;

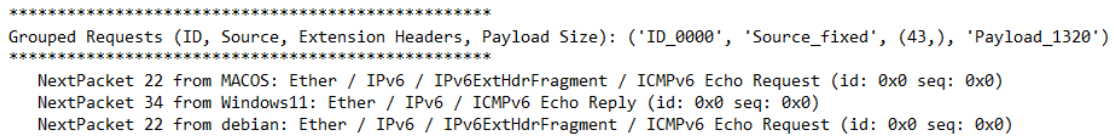


Figure 3.8: Visual representation of the differences found between MacOS, Windows 11, and Linux Debian. The first line provides a description of the packet sent, which includes the ID number (0), the fixed source, the Routing Header (43), and the length of the Payload (1320 bytes). The following lines show the next meaningful packet in the .pcap file after the one that was sent. Each line shows the position on the .pcap file, the operating system used, and a summary of the packet.

- The response to each echo request, if any, is extracted;
- A comparison is made of the responses of all packet groups for each operating system;
- Differences between operating systems, if any, are printed with their location in the .pcap file.

We have designed this differential fuzzer with a modular and flexible architecture to allow easy extension and adaptation to specific testing needs. The core of the fuzzer is a central module dedicated to generating and sending ICMPv6 packets, which is supported by numerous support modules that facilitate packet sending and customization.

These support modules include those dedicated to the calculations required to determine critical packet parameters, such as the maximum payload length and the maximum number of Extension headers allowed. These modules ensure that the generated packets are within Scapy's limits.

In addition, we have implemented modules that allow detailed packet customization. For example, specific content can be added to the payload of packets, or custom IPv6 addresses can be inserted to mask the origin of the broadcasts. These modules provide a high degree of flexibility to adapt the generated packets to the specific needs of the current tests.

It is important to note that when customizing IPv6 addresses, they must first be assigned to an interface before they can be used. This involves sending a Neighbor Solicitation message containing the tentative address as the target, in order to verify that the address is not already in use on the network [35]. To avoid conflicts and ensure accurate testing, we have included a 2-second delay before sending the packet each time we assign an IPv6 address to an interface.

The modular architecture of the differential fuzzer makes it much easier to expand and integrate new features. We can easily add new modules to support new types of tests or to enhance packet customization capabilities. In addition, this modular structure allows us to keep the

fuzzer up-to-date and adaptable to future developments of the IPv6 protocol and network security testing techniques.

4 Results

In this chapter, we will explore the results of our testing, discuss the practical implications of these findings, highlight their impact on network performance and security, and explain the limitations we faced. The goal is to provide a detailed picture of the information collected, contributing to a deeper understanding of the various issues addressed in our experiments.

4.1 Overlapping Fragments

Several tests were performed by expanding the number of operating systems [3] and confirming that the majority of operating systems are not RFC 5722-compliant [Table 4.1].

The columns "Type 1", "Type 2" and "Type 3" represent the types of tests performed, each characterized by a specific set of packet permutations with overlapping fragments; their description can be read in the 3.2 section. The presence of a "✓" indicates that at least one of the permutations of that type received a successful response, while the presence of an "X" indicates that no response was received for any of the permutations.

There is considerable variation in the results between the operating systems tested. For example, systems such as Linux (Arch, Debian, and Ubuntu), Windows 10, Windows 11, and macOS (Darwin and Sonoma) are shown to be vulnerable for every type of test, while Mikrotik and OpenBSD are partially vulnerable. Only some operating systems, such as NetBSD, iOS, and Cisco IOS 15 routers, demonstrate compliance with the standards dictated by RFC 5722.

Operating System	Type 1	Type 2	Type 3
GNU/Linux Arch	✓	✓	✓
GNU/Linux Debian 12	✓	✓	✓
GNU/Linux Ubuntu 23.04	✓	✓	✓
NetBSD	X	X	X
OpenBSD	X	✓	X
FreeBSD 13.1	X	✓	X
FreeBSD 13.2	X	✓	X
FreeBSD 14	X	✓	X
Mikrotik 6	X	✓	X
Mikrotik 7	✓	✓	✓
Windows 10	✓	✓	✓
Windows 11	✓	✓	✓
macOS Darwin	✓	✓	✓
macOS Darwin	✓	✓	✓
Android (MIUI)	✓	✓	✓
iOS	X	X	X
Cisco IOS 15 router	X	X	X

Table 4.1: The table shows the RFC 5722 conformance status of various Operating Systems. The "X" symbol indicates compliance, while the "✓" symbol indicates non-compliance. Each column Type represents a different variation of the model that was tested.

The result of these tests not only highlights the different systems that do not comply with RFC 5722 but also provides an analysis of how these systems perform the packet reassembly operation. This, in turn, provides several opportunities for a hypothetical attacker to launch targeted attacks against these systems. By recognizing how an operating system reassembles packets, it is possible to profile that system, or alternatively, it is possible to use this vulnerability to transmit malicious packets, bypassing IDS/IPS that do not reassemble the packet before examining it.

4.1.1 Fingerprint

Fingerprinting with overlapping fragments involves sending a series of permutations like those proposed in our model, each uniquely designed to probe the operating system's particularities. Once sent, these packets are monitored to observe the responses of the operating systems, in particular the presence or absence of Echo Replies and the contents of their payloads. By carefully analyzing the responses obtained from different permutations of packets with overlapping

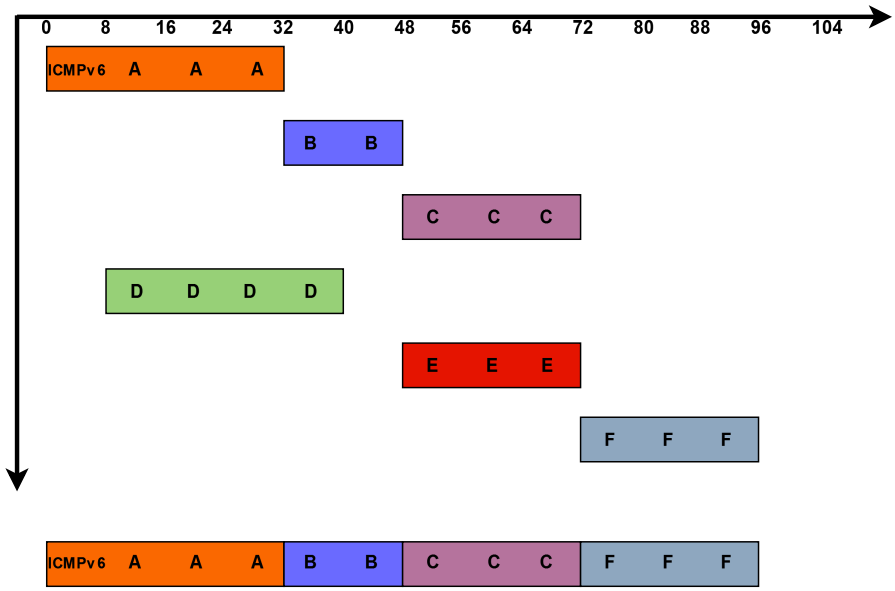


Figure 4.1: Example of how MacOS assembles policy handles a set of overlapping fragments. Each block in the sequence represents a fragment. The payload of each fragment is a sequence of bytes that encode a character. MacOS follows a policy where it accepts and assembles the first received fragments and discards all subsequent fragments that overlap with those already processed (green and red fragments are discarded).

fragments and comparing them to known patterns of operating system behavior, meaningful conclusions can be drawn about the target operating system. Variations in response patterns, such as the presence or absence of certain types of echo replies or differences in response payloads, can provide valuable clues to the target operating system.

For example, let us consider the analysis conducted on a Windows 10 system and a Mikrotik 6. We notice that with the first model, which involves the sending of a permutation of overlapping fragments, the two systems produce different results for the same permutation [Figure 4.2]. The assembly policy of Windows 10 receives a collection of fragments and discards any subsequent fragments that overlap with those already processed. As a result, it successfully constructs a valid packet and responds appropriately, indicating a deviation from RFC 5722 standards. In contrast, Mikrotik 6 discards all overlapping fragments and fails to construct a packet to respond. However, using the second model and sending the same permutation multiple times, we observed that while Windows 10 behaves consistently, Mikrotik responds after receiving some fragments from the second set of the same permutation. This suggests that Mikrotik 6 uses the new fragments from the second set to compose a new packet, incorporating fragments from both the first and second sets [Figure 4.3. These discrepancies in results provide us with valuable insights into the nature and behavior of the operating systems being

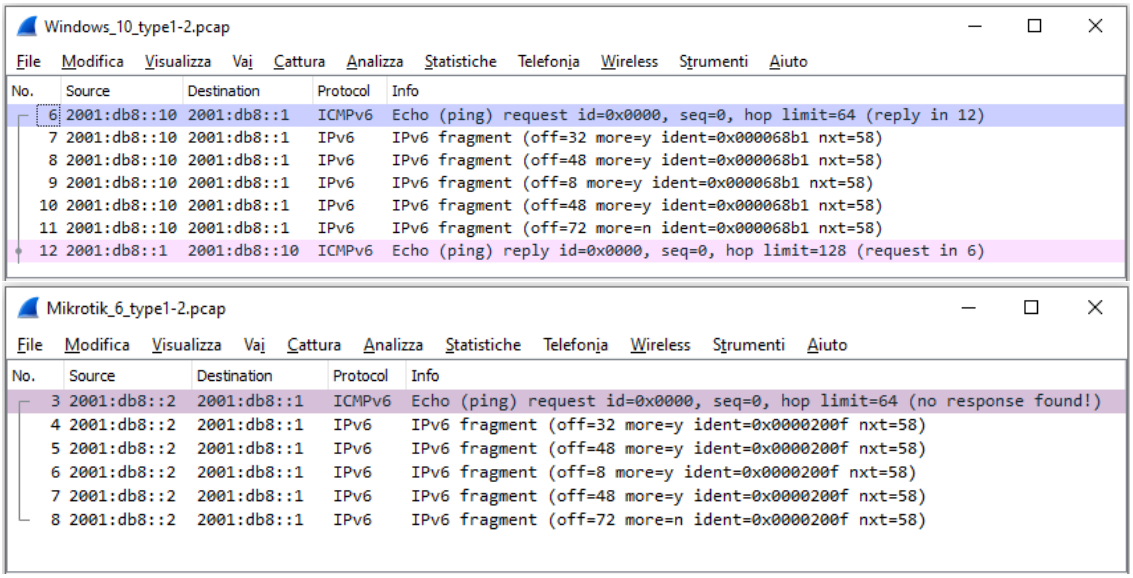


Figure 4.2: Different responses from Windows 10 and Mikrotik 6 obtained with the first model. Windows 10's assembly policy discards any subsequent fragments that overlap with those already processed and assembles a packet, allowing the system to respond to it. However, in Mikrotik 6, overlapping fragments are discarded without any reply.

analyzed.

However, it is important to note that fingerprinting does not always provide a clear distinction between different versions of the same operating system. For example, we may not be able to distinguish between an address associated with Windows 10 and one associated with Windows 11 if they both follow the same packet reassembly policy.

After several tests, it was discovered that certain operating systems always generate a Time Exceeded packet with Code 1 in response to certain sequences of overlapping fragments [Figure 4.4. This type of packet is used to indicate a fragment reassembly timeout [33] [36]. Although using this method alone may not be completely accurate, when combined with the previous results regarding overlapping fragments, it can provide more clarity about the target device.

4.1.2 Extra Attacks

In a potential attack scenario, an attacker could perform a **Denial of Service (DoS)** attack by predicting the ”id” field and spoofing the source address to send a spoofed fragment to the target host receiver before it can reassemble all the fragments received from the victim, causing packet rejection. However, this attack is highly situational and difficult to execute and therefore

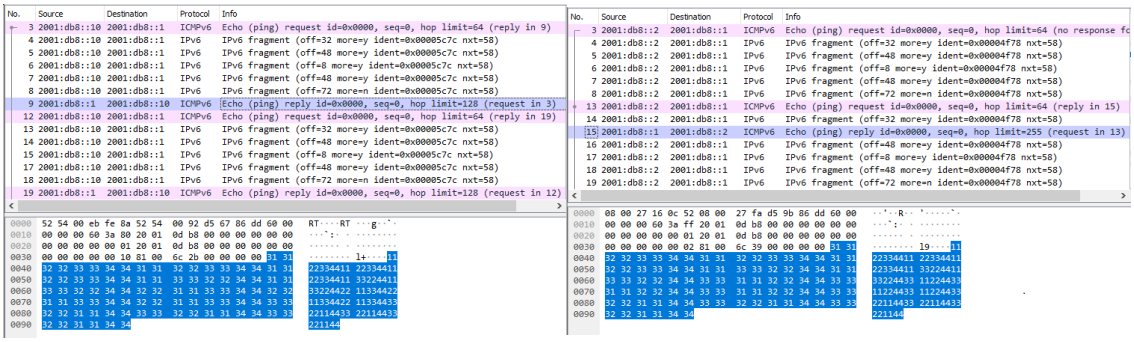


Figure 4.3: Different responses from Windows 10 (left) and Mikrotik 6 (right) obtained with the second model. Windows 10's assembly policy for each set of the same permutation discards any subsequent fragments that overlap with those already processed and assembles a packet, allowing the system to reply to it. Mikrotik 6 discards all the overlapping fragments and waits for the second set of the same permutation to create a new packet incorporating fragments from both the first and second sets and reply to it.

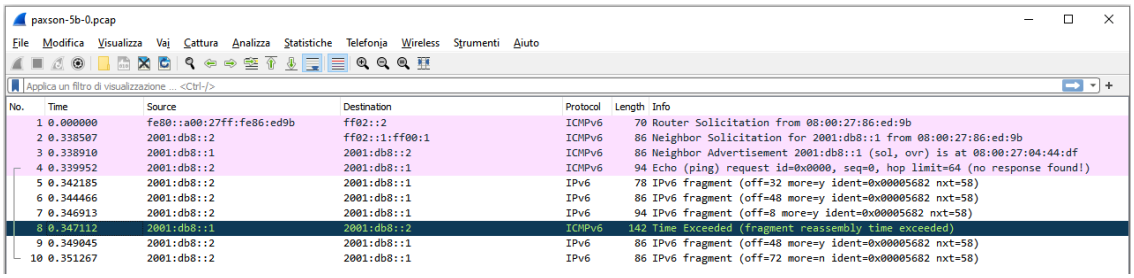


Figure 4.4: Example of a FreeBSD 13 system responding with a Time Exceed packet.

poses a limited real-world threat. Another possible attack, known as the **Modification Attack**, involves the attacker sending one or more fragments and ensuring that when these fragments are reassembled with legitimate ones, the resulting packet is correct and the final payload is as desired. To be successful, the attacker must calculate the correct checksum to avoid packet rejection due to a checksum mismatch.

4.2 Atomic Fragments

Operating System	Atomic Fragment
GNU/Linux Arch	X
GNU/Linux Debian 12	X
GNU/Linux Ubuntu 23.04	X
NetBSD	X
OpenBSD	X
FreeBSD 13.1	✓
FreeBSD 13.2	✓
FreeBSD 14	X
MikroTik 6	X
MikroTik 7	X
Microsoft Windows 10	X
Microsoft Windows 11	X

Table 4.2: Atomic Fragment vulnerability in different operating systems. The "✓" indicates that the system is vulnerable, while an "X" indicates that it is not.

Table 4.2 shows whether or not the Atomic Fragment vulnerability is present in different operating systems. The "✓" symbol indicates that the operating system is vulnerable, while the "X" symbol means it's not vulnerable.

According to the table, the vulnerability that allows atomic fragments to bypass firewalls was identified only in the case of FreeBSD 13.1 and FreeBSD 13.2. However, this finding does not rule out the possibility of the same vulnerability being present under different conditions or with other untested operating systems.

In FreeBSD 13.1, the scrubbing process rebuilds these types of atomic fragments into the original packet. However, the system fails to apply any rules meant for layers four and higher, such as the one that should block ICMPv6 packets. As a result, the fragment matches other rules in the firewall configuration and passes through without being blocked. Additionally, the scrubbing process fixes the packet, allowing any operating system behind the firewall to accept it. This issue was reported to FreeBSD developers, who addressed it in FreeBSD version 13.2 and FreeBSD version 14. However, from our tests, it's evident that while the problem was resolved in FreeBSD 14, it persists in version 13.2.

4.3 ICMPv6 zero

Operating systems	ICMPv6 zero	ICMPv6 zero + Router
Linux 5.x (Debian 11)	X	X
Linux 6.x (Debian 12)	X	X
Windows 10	✓	n/a
Windows 11	✓	n/a
NetBSD	✓	X
OpenBSD	X	X
FreeBSD 13	✓	X
FreeBSD 14	✓	X
macOS Sonoma	X	n/a
Mikrotik 7	X	X
Mikrotik 6 LTS	X	X
Cisco IOS 15 router	X	n/a

Table 4.3: Presence of the ICMPv6 Zero vulnerability in different operating systems. The "✓" symbol indicates that the system is vulnerable, while "X" indicates that it is not. "n/a" means tests were not conducted due to the inability to run these operating systems as routers. The first column shows the results of tests made with a direct connection between the attacker and the victim, while the second column shows the results of tests where the victim was used as a router.

The table 4.3 provides information on whether the ICMPv6 Zero vulnerability is present in various operating systems. The "✓" symbol indicates that the operating system is vulnerable, while the "X" symbol indicates that it is not. The label "n/a" indicates that tests were not performed because these operating systems cannot be run as routers [section 4.5].

The table indicates that various operating systems, such as NetBSD and FreeBSD, are prone to the ICMPv6 Zero vulnerability. Additionally, it is worth noting that even widely used systems like Windows 10 and Windows 11 have this problem. The table also suggests that if the system is used as a router, this issue appears to be resolved. However, further research is necessary to determine if there are any factors that may influence these results.

This vulnerability may appear differently under different circumstances. However, in all cases, if there has been no prior communication between the attacker and the victim, it will be necessary to send between 3 and 10 packets in rapid succession, with no pauses longer than 1-2 minutes in between, before expecting to receive an Echo Reply 4.5. Once the victim starts

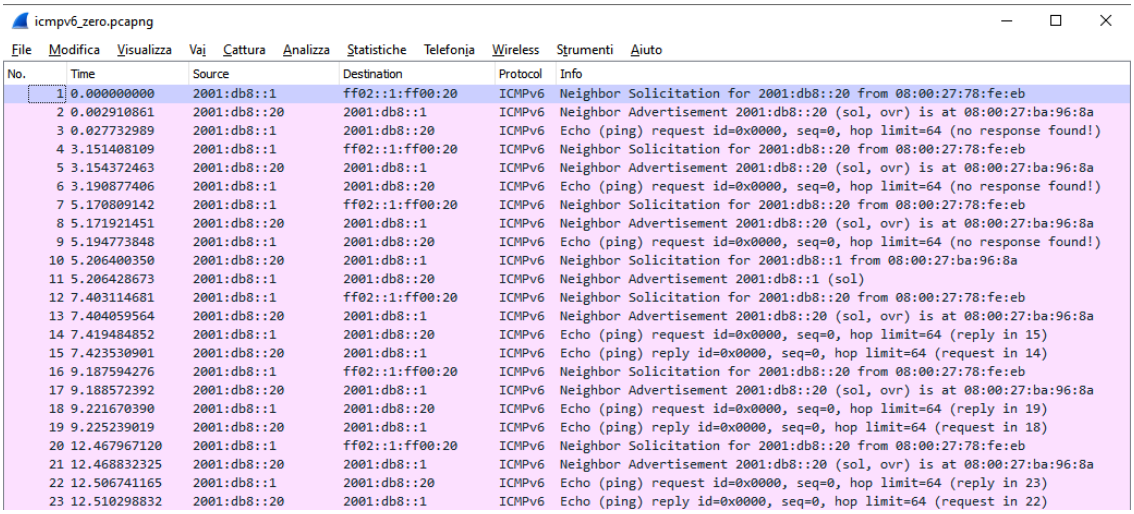


Figure 4.5: Representation of how ICMPv6 Zero bug manifests itself in a victim operating system without prior communication with the attacker. At the fourth Echo Request packet, the system begins to respond with an Echo Reply.

responding, it will continue to do so without having to retransmit a certain number of initial echo request packets, at least for a certain amount of time. This duration varies depending on the operating system used. In some cases, it may take several minutes, in others until the operating system shuts down. Suppose there has been previous communication between the attacker and the victim. In that case, certain operating systems may provide an immediate response if the firewall is reenabled and blocks the Echo Request packets.

Many operating systems use firewalls that block ICMPv6 Echo Request packets, allowing only neighbor solicitation and advertising to communicate their presence on the Internet. This means that some devices are hidden and cannot be contacted, making them difficult to locate and identify. However, the ICMPv6 Zero exploit can be used to find and identify such devices. Once identified, these devices may be vulnerable to targeted attacks or system intrusions. It is not clear whether this exploit can be used to cause harm, but it is a potential security risk that requires further investigation and evaluation.

4.4 Differential Fuzzer

The fuzzer tests were initially conducted while the firewall was active to ensure that the outcomes matched those of the atomic fragments and ICMPv6 Zero bug tests. After that, the tests were repeated with the firewall disabled to observe and analyze any discrepancies in the response of various operating systems under normal conditions.

The fuzzer tests were performed using each combination described in section 3.5, but limited to a maximum of four Extension headers per packet. This approach allowed us to test all four Extension headers together at least once. We decided not to exceed this maximum number of Extension headers because we did not believe it would provide any additional information, and because it would take too long for the program to test a single operating system. In the future, we plan to parallelize these operations to verify all possible combinations with the maximum number of Extension headers within an acceptable timeframe.

```
**************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (0,), 'Payload_0')
**************************************************
   NextPacket 6 from Windows10: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 8 from OpenBSD: Ether / IPv6 / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 4 from Debian: Ether / IPv6 / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
```

Figure 4.6: Example of the ICMPv6 Zero bug found in Windows 10 with firewall enabled. OpenBSD and Debian firewalls block the ICMPv6 Echo Request packet coming from the attacker while Windows 10 replies to them.

Thanks to the program developed to analyze .pcap files, we were able to confirm that the fuzzer could replicate all other tests except those related to overlapping fragments, which were not included in the original program, covering atomic fragments and the ICMPv6 Zero bug.

As shown in figure 4.6, the program detected a discrepancy between the results of Debian, OpenBSD, and Windows 10. The figure shows that when sending an echo request packet with a fixed ID, a constant source address, a hop-by-hop header (the third variable), and a null payload, Debian and OpenBSD do not respond, and the subsequent packet is another echo request packet. However, in the case of Windows 10, there is an immediate response that highlights the ICMPv6 zero bug explained earlier.

In Table 1 [Table 4.4], we provide a list of selected packets that are helpful in understanding how our fuzzer works. Each packet in the list is an ICMPv6 Echo Request that comes with its own ID number, a Sequence number, and a Source address that is either random or fixed (usually 2001:db8::10). Additionally, we mention the type of Extension header that each packet uses (0 = Hop-by-Hop, 43 = Routing Header, 44 = Fragment Header, 60 = Destination Options), and the size of the payload in bytes.

In Table 2 (Table 4.5), we use these packets to compare the responses of different operating systems. The label “Reply” indicates that we received an ICMPv6 Echo Reply, “Fragm” indicates that the operating system fragmented the packet into two fragments to send an echo reply, “Error” indicates either a Parameter Problem, an erroneous error field, or an unrecognized Next header type encountered, and “NoReply” indicates no response was received.

Packet	ID	Seq	Source	Extension Headers	Payload
Packet 1	0	0	Fixed	(0,)	Max
Packet 2	0	0	Fixed	(43,)	0
Packet 3	0	0	Fixed	(0,0)	0
Packet 4	0	0	Fixed	(43,43)	0
Packet 5	0	0	Fixed	(0,0,44)	0
Packet 6	0	0	Fixed	(0,0,44)	64
Packet 7	0	0	Fixed	(0,0,44)	128
Packet 8	0	0	Fixed	(0,0,44)	1000
Packet 9	0	0	Fixed	(0,0,44)	Max

Table 4.4: A selected list of ICMPv6 Echo Request packets with their own ID number, Sequence number, Source Address (always Fixed), their Extension headers (0 = Hop-by-Hop, 43 = Routing Header, 44 = Fragment Header, 60 = Destination Options), and the size of their payload in byte (Max = maximum packet size allowed by the MTU).

Packet	Mikrotik7	Windows10	FreeBSD14	NetBSD	OpenBSD	Debian
Packet 1	Reply	Reply	Reply	Fragm	Fragm	Reply
Packet 2	Reply	NoReply	Reply	Reply	Error	Reply
Packet 3	Error	NoReply	Error	Error	Error	Error
Packet 4	Reply	NoReply	Reply	NoReply	Error	Reply
Packet 5	Error	NoReply	Error	Error	Error	Error
Packet 6	NoReply	NoReply	Error	Error	Error	NoReply
Packet 7	Error	NoReply	Error	Error	Error	Error
Packet 8	NoReply	NoReply	Error	Error	Error	NoReply
Packet 9	Error	NoReply	Error	Error	Error	Error

Table 4.5: Comparison of different operating systems' behavior in response to specific packets. "Reply" indicates that an echo reply was received, "Fragm" indicates that the operating system fragmented the packet to send an echo reply, "Error" indicates a Parameter Problem, and "NoReply" indicates no response was received.

```
************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (0,), 'Payload_1420')
************************************************
   NextPacket 18 from Mikrotik7: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 16 from Windows10: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 17 from FreeBSD14: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 17 from NetBSD: Ether / IPv6 / IPv6ExtHdrFragment / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 17 from OpenBSD: Ether / IPv6 / IPv6ExtHdrFragment / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 16 from Debian: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
```

Figure 4.7: Response of different operating systems to Packet 1 (ID = 0, Seq = 0, Fixed Source Address, Extension Header = (43,) and payload size = 0): in this scenario, Windows 10 doesn't reply, and OpenBSD fragment the packet into two fragments.

Starting with Packet 1 [Figure 4.7], we observed that when NetBSD and OpenBSD receive

packets with large payloads, they fragment the packet into two fragments to respond with an Echo Reply. This fragmentation occurs for every packet over 1300 bytes, except those that cannot be replied to. As a result, it can be concluded that these systems are incapable of replying with packets that are too big, even if the MTU allows it.

```
**************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (43,), 'Payload_0')
**************************************************
   NextPacket 20 from Mikrotik7: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 18 from Windows10: Ether / IPv6 / IPv6ExtHdrRouting / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 19 from FreeBSD14: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 20 from NetBSD: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 20 from OpenBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrRouting / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 18 from Debian: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
```

Figure 4.8: Response of different operating systems to Packet 2 (ID = 0, Seq = 0, Fixed Source Address, Extension Header = (43,) and payload size = 0): in this scenario, Windows 10 doesn't reply while OpenBSD sends a Parameter Problem (erroneous header field encountered) packet.

Packet 2 is an Echo Request with a Routing Header type 0 (RH0) [Figure 4.8]. When we sent this packet we noticed that Windows 10 did not reply, and OpenBSD sent a Parameter Problem (erroneous header field encountered) packet, while all other systems replied. The Routing Header is a feature in IPv6 that allows the source to specify one or more intermediate nodes to be visited on the way to a packet's destination.

As described in RFC 5095 [37], a single RH0 may contain multiple intermediate node addresses and the same address may be included multiple times in the same packet. This feature enables the creation of a packet that alternates between two RH0-processing hosts or routers several times. This technique can be used by an attacker to send a stream of packets that can be amplified along the path between two remote routers. This can cause congestion along arbitrary remote paths and act as a denial-of-service mechanism. Due to this reason, this type of packet (Routing Header type 0) is deprecated and should be dropped by the firewall. We believe that this is why Windows 10 did not reply, and OpenBSD sent an error message.

```
**************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (0, 0), 'Payload_0')
**************************************************
   NextPacket 51 from Mikrotik7: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 44 from Windows10: Ether / IPv6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 50 from FreeBSD14: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 54 from NetBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 53 from OpenBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 49 from Debian: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
```

Figure 4.9: Response of different operating systems to Packet 3 (ID = 0, Seq = 0, Fixed Source Address, Extension Header = (0,0) and payload size = 0): in this scenario, all operating systems except for Windows 10 respond with an error message, while Windows 10 remains silent.

In the third line, Packet 3, we sent an Echo Request with two Hop-by-Hop Headers [Figure 4.9]. However, we observed that operating systems such as Windows 10 do not comply

with RFC 8883 [38]. According to this RFC, if a malformed packet with the same Extension headers (such as hop-by-hop) is repeated, the response should result in a Parameter Problem error. In our case, Windows 10 did not send back an error message despite not replying.

```
************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (43, 43), 'Payload_0')
************************************************
   NextPacket 92 from Mikrotik7: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 78 from Windows10: Ether / IPv6 / IPv6ExtHdrRouting / IPv6ExtHdrRouting / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 93 from FreeBSD14: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
   NextPacket 98 from NetBSD: Ether / IPv6 / IPv6ExtHdrRouting / IPv6ExtHdrRouting / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 98 from OpenBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrRouting / IPv6ExtHdrRouting / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 90 from Debian: Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
```

Figure 4.10: Response of different operating systems to Packet 4 (ID = 0, Seq = 0, Fixed Source Address, Extension Header = (43,43) and payload size = 0): in this scenario, Windows 10 and NetBSD don't reply while OpenBSD sends a Parameter Problem (erroneous header field encountered) packet.

Packet 4 is an echo request containing two routing headers of type 0. However, it is difficult to understand the behavior of some operating systems after we sent this packet. Windows 10 did not respond, while OpenBSD returned an error message. Unlike the situation with Packet 2, we can see that NetBSD doesn't respond either, indicating that there is something that triggers this behavior.

```
************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (0, 0, 44), 'Payload_0')
************************************************
   NextPacket 177 from Mikrotik7: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 135 from Windows10: Ether / IPv6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 175 from FreeBSD14: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 175 from NetBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 183 from OpenBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 176 from Debian: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)

************************************************
Grouped Requests (ID, Source, Extension Headers, Payload Size): ('ID_0000', 'Seq_0000', 'Source_fixed', (0, 0, 44), 'Payload_64')
************************************************
   NextPacket 181 from Mikrotik7: Ether / IPv6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 136 from Windows10: Ether / IPv6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 179 from FreeBSD14: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 177 from NetBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 185 from OpenBSD: Ether / IPv6 / ICMPv6ParamProblem / IPerror6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
   NextPacket 180 from Debian: Ether / IPv6 / IPv6ExtHdrHopByHop / IPv6ExtHdrHopByHop / IPv6ExtHdrFragment / ICMPv6 Echo Request (id: 0x0 seq: 0x0)
```

Figure 4.11: Response of different operating systems to Packet 5 & 6 (ID = 0, Seq = 0, Fixed Source Address, Extension Header = (0,0,44) and payload size = 0, 64): in these scenarios, we can notice how in response to Packet 5 only Windows 10 doesn't reply with an error message, while in response to Packet 6 in addition to Windows 10 also Mikrotik 7 and Debian don't reply with any error messages.

Packets 5, 6, 7, 8, and 9 are similar packets that contain the same Extension headers, two Hop-by-Hop Headers, and one Fragment Header [Figure 4.11]. The only difference between these packets is the payload size. Similar to Packet 3, when we sent Packet 5, 7, and 9 with payload sizes of 0, 128, and max, respectively, we noticed that all operating systems except for Windows 10 responded with an error message, while Windows 10 didn't reply. On the other hand, when we sent Packet 6 and 8 with payload sizes of 64 and 1000, respectively, not only Windows 10 but also Mikrotik 7 and Debian didn't respond with any packet.

These are just some of the differences that we found by comparing the responses of various operating systems. Similar behaviors were noted in response to other packages. Many of these behaviors require further investigation to understand why they occur and to determine if they can be exploited for some type of malicious purpose.

A special note should be made in the case of OpenBSD, which during our testing we found to be protected by a default firewall with the following rules:

- block return all
- pass all flags S/SA
- block return in on ! lo0 proto tcp from any to any port 6000:6010
- block return out log proto tcp all user = 55
- block return out log proto udp all user = 55

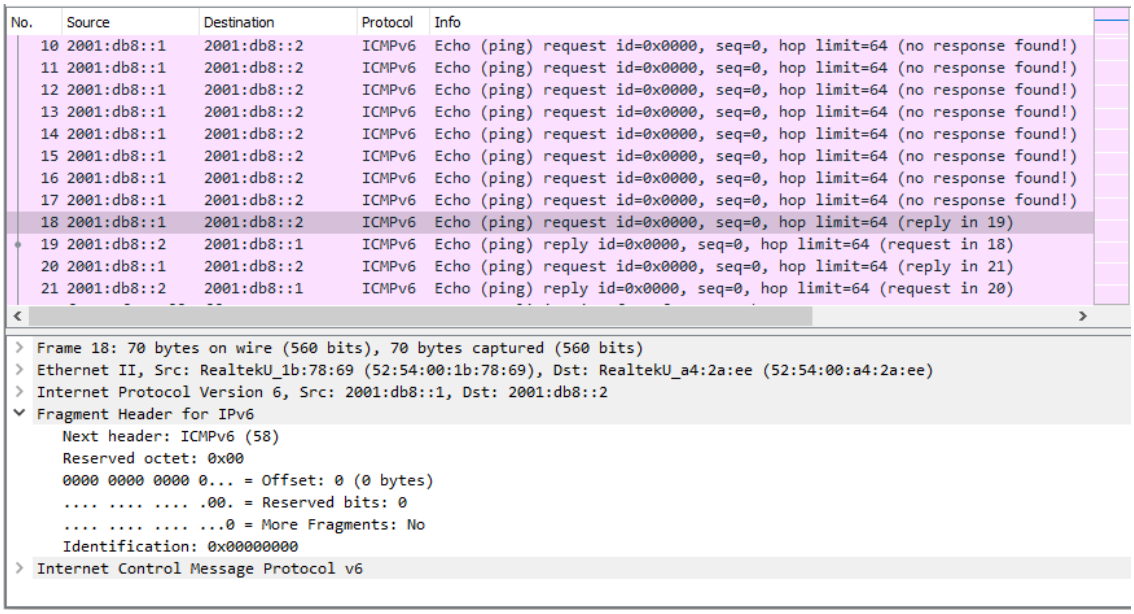


Figure 4.12: OpenBSD responds to ICMPv6 Echo Request packets with a single fragment header, even when the firewall should block all packets.

These rules should block all incoming and outgoing packets, with the exception of some TCP and UDP traffic. However, as you depicted in Figure 4.12, if you send an Atomic fragment with a single Fragment Header, OpenBSD responds to it

In conclusion, the use of this fuzzer and its subsequent enhancements could help to highlight any upper-layer IPv6 issues, show which operating systems are compliant, and in the case of bugs or vulnerabilities such as atomic fragments and ICMPv6 Zero, how to address them effectively.

4.5 Limitations

During the testing phase, we encountered several limitations that affected our ability to conduct comprehensive testing across all target operating systems and devices. The most significant limitations we encountered are described in the following sections.

4.5.1 Firewall and Network Limitations

Some tests, such as Atomic Fragment and ICMPv6 Zero, required a firewall to be enabled and configured on some operating systems. However, we ran into restrictions on systems such as mobile devices where we could not configure the firewall, limiting us to overlapping fragment tests.

Some tests required operating systems to be used as virtual routers in order to successfully route traffic to a secondary system. However, not all operating systems allowed routing configuration or did so incompletely. For example, mobile devices did not provide the ability to be used as IPv6 routers, which limited the scope of our tests. In addition, Windows 10 and 11 had limitations in applying firewall rules to the router role, resulting in a situation where ICMPv6 Echo Request packets were blocked when addressed to the router, but passed when addressed to the secondary system. In the case of ESXi and Cisco IOS, we have limited control over the firewall, and ESXi in particular is not able to act as a router, which eliminates the ability to perform some of the tests.

4.5.2 Virtualization Issues with Proxmox

Over the course of our testing, we encountered several problems configuring operating systems in a virtual environment, which led to repeated changes in the virtualization platform. We started with Vagrant, moved to GNS3, and then Proxmox. However, fragment management issues with Proxmox forced us to go back to Vagrant and GNS3.

Proxmox is a hypervisor that manages virtual machines (VMs) on a host system. These VMs run on a Debian Linux distribution and communicate over Ethernet bridges. There are two

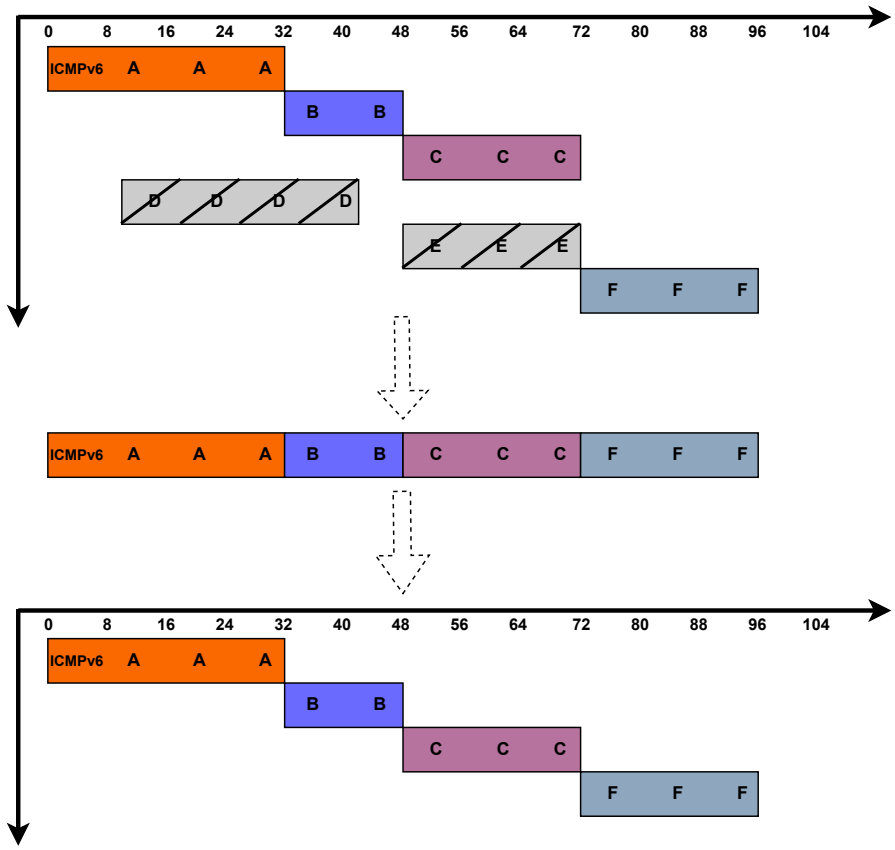


Figure 4.13: The process of reassembling fragments in Proxmox, where overlapping fragments are discarded and then re-fragmented. The resulting fragmented transmission does not contain the two additional fragments that are overlapping, hindering the tests.

types of Ethernet bridges that Proxmox uses: OpenVSwitch and Linux bridges. The issue with OpenVSwitches is that they interfere with fragment reassembly even when configured not to do so. Therefore, we opted to use Linux Bridges.

Proxmox automatically configures iptables when the Firewall is enabled. This configuration is essential to Proxmox's network security measures. When the Firewall is activated, Proxmox sets a kernel flag in Linux that directs all Ethernet frames within bridges to pass through the firewall. This flag is automatically set to 'active' by Proxmox itself. Without this flag being enforced in this manner, the issue we encountered would not have arisen.

The issue lies in how the firewall interacts with fragmented packets. When active, the Proxmox firewall subjects all Ethernet frames, including fragments, to its rules. The Linux kernel, under the influence of Proxmox, reassembles fragments based on its reassembly policy. This

policy is then applied again by the firewall, fragmenting the (now-reassembled) packet once more using the same policy employed in the reassembly process.

Consequently, the firewall's reassembly policy can lead to the exclusion of discarded (overlapping) fragments from the transmission. This means that redundant or overlapping fragments are not reintroduced into the packet transmission, potentially resulting in a different set of fragments to evaluate [Figure 4.13].

Disabling the infrastructure firewall completely would be the solution, but this may not be practical. To avoid these complications, we preferred to use another platform, such as Vagrant, to run the tests without such interference.

Furthermore, it is advisable that if conducting these tests in virtualized environments, one should verify the configurations of Linux bridges, specifically the settings within /proc/sys/net/bridge/. These settings determine how frames in the bridges should travel and where they should go - whether through the older arptables or iptables, or the newer nftables.

4.5.3 Testing timing with the fuzzer

Another limitation was the time required to run a complete test cycle with the fuzzer. We originally planned to test all possible combinations of Extension header arrangements, up to the maximum number allowed by the MTU. However, this would have taken a very long time, exceeding 30 days of continuous activity. As a result, we postponed this test until we could run tests with multiple devices in parallel to speed up the testing process.

5

Conclusion

The research conducted has examined in detail the dynamics of IPv6 packet manipulation through a series of tests on a variety of operating systems. The results obtained revealed various vulnerabilities and non-compliance with standards, highlighting the potential impact on network security and performance. The Overlapping Fragments section has expanded the number of operating systems tested and confirmed that most of them are not compliant with the RFC 5722 standard. This non-compliance not only highlights the different critical issues in the various systems but also provides opportunities for targeted attacks such as Denial of Service (DoS) and Modification Attacks. The analysis of the atomic fragments revealed a specific vulnerability against Freebsd, indicating the possibility that this vulnerability could manifest itself under different conditions or on other systems that have not yet been tested. This aspect is fundamental to understanding how certain systems handle IPv6 packets and potentially allows an attacker to bypass firewalls. The discovery of the ICPMv6 zero bug, which is present in a large number of operating systems including Windows 10 and Windows 11, led us to develop a systematic method for finding different IPv6 vulnerabilities. This resulted in the creation of a differential fuzzer, which will play an important role in discovering other issues in different operating systems. Finally, this research provides a detailed overview of the vulnerabilities and dynamics observed in IPv6 packet handling, highlighting the importance of rigorous standards compliance and implementation of appropriate security measures to protect networks from potential attacks.

5.1 Future Works

To extend this research and to address further challenges in the area of IPv6 network security, further studies and future developments are needed.

Analyzing Packet Frequency

After conducting numerous tests with various packet types, such as Atomic Fragments or packets with unique headers, the next step is to assess their frequency on the network. By monitoring network traffic and analyzing the frequency of these packets in real-world scenarios, researchers can obtain insights into the practical importance of specific types of attacks. Rare or anomalous packets may indicate potential malicious activity, prompting the implementation of proactive measures to mitigate such threats directly.

Expanding Device Testing

In order to comprehensively assess the security of IPv6 networks, it is imperative to expand the scope of device testing. This expansion should include the integration of IoT devices and other emerging technologies. IoT devices, with their diverse architectures and communication protocols, present unique security challenges that require careful study. By including these devices in testing protocols, researchers can better understand and address potential vulnerabilities within the broader network ecosystem.

Investigating ICMPv6 Vulnerabilities

An important area for further research is the investigation of the vulnerability of ICMPv6 with zero ID. This involves evaluating the feasibility of obtaining responses from IP addresses on the Internet that hide their presence behind firewalls or other obfuscation systems. Many operating systems employ firewalls that selectively block ICMPv6 echo request packets, effectively hiding certain devices from external communication. However, the use of ICMPv6 Zero provides an opportunity to identify and locate such hidden devices. Further research is needed to assess the potential security implications of this exploit and its effectiveness in bypassing network defenses. Understanding the risks associated with ICMPv6 vulnerabilities is essential to improving the overall security posture of networks and developing effective mitigation strategies.

Enhancing Fuzzer Capabilities

To further improve the effectiveness and versatility of the IPv6 fuzzer developed in this research, various improvements and extensions can be made. These include:

1. **Protocol Compatibility:** The current fuzzer only focuses on ICMPv6 protocols. Extending its compatibility to other IPv6 protocols such as UDPv6, TCPv6, or DHCPv6 would broaden its applicability and increase its utility. This would provide comprehensive testing coverage for a wider range of network services and functionalities;

2. **Integration of Deep Learning Analysis:** Introducing a deep learning-based analysis system to the fuzzer could significantly improve its ability to interpret and evaluate response data from target systems. By training the deep learning model on a diverse dataset of response patterns, the fuzzer could autonomously identify significant differences in responses between different operating systems or protocol implementations. This would enable more efficient identification and prioritization of potential vulnerabilities, helping researchers focus their efforts on the most critical areas;

3. **IPv4 Compatibility:** While IPv6 is gaining popularity, IPv4 is still widely used in many network environments. Developing a version of the fuzzer that supports IPv4 would allow researchers to assess the security posture of both IPv4 and IPv6 networks using a unified testing framework. By adapting the existing fuzzer to support IPv4 protocols, and addressing the unique characteristics of IPv4 packet structures and behaviors, researchers can perform comprehensive security assessments across heterogeneous network infrastructures.

References

[1] U. Shankar, "Active mapping: Resisting nids evasion without altering traffic," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-03-1246, Dec 2002. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5546.html

[2] A. Atlasis, "Attacking ipv6 implementation using fragmentation," *Blackhat europe*, pp. 14–16, 2012.

[3] E. Di Paolo, E. Bassetti, and A. Spognardi, "A new model for testing ipv6 fragment handling," in *European Symposium on Research in Computer Security*. Springer, 2023, pp. 277–294.

[4] G. Statistics, "IPv6 Adoption." [Online]. Available: https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption

[5] B. E. Carpenter and S. Jiang, "Transmission and Processing of IPv6 Extension Headers," RFC 7045, Dec. 2013. [Online]. Available: https://www.rfc-editor.org/info/rfc7045

[6] C. Kaufman, R. Perlman, and B. Sommerfeld, "Dos protection for udp-based protocols," 10 2003, pp. 2–7.

[7] I. Miller, "Protection Against a Variant of the Tiny Fragment Attack (RFC 1858)," RFC 3128, Jun. 2001. [Online]. Available: https://www.rfc-editor.org/info/rfc3128

[8] S. Krishnan, "Handling of Overlapping IPv6 Fragments," RFC 5722, Dec. 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5722

[9] F. Gont, "Processing of IPv6 "Atomic" Fragments," RFC 6946, May 2013. [Online]. Available: https://www.rfc-editor.org/info/rfc6946

[10] D. S. E. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, Jul. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8200

[11] Éric Vyncke, K. Chittimaneni, M. Kaeo, and E. Rey, "Operational Security Considerations for IPv6 Networks," RFC 9099, Aug. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9099

[12] R. Bonica, F. Baker, G. Huston, B. Hinden, O. Trøan, and F. Gont, "IP Fragmentation Considered Fragile," RFC 8900, Sep. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8900

[13] D. Reed, P. S. Traina, and P. Ziemba, "Security Considerations for IP Fragment Filtering," RFC 1858, Oct. 1995. [Online]. Available: https://www.rfc-editor.org/info/rfc1858

[14] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

[15] F. Gont, "Security implications of predictable fragment identification values," Tech. Rep., 2016.

[16] ——, "Security assessment of the internet protocol version 6 (ipv6)," *UK Centre for the Protection of National Infrastructure,(available on request)*.

[17] F. Gont, W. S. LIU, and T. Anderson, "Generation of IPv6 Atomic Fragments Considered Harmful," RFC 8021, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8021

[18] A. Atlasis, "Security impacts of abusing ipv6 extension headers," in *Black Hat security conference*, 2012, pp. 1–10.

[19] P. Bedi and A. Dua, "Network steganography using extension headers in ipv6," in *International Conference on Information, Communication and Computing Technology*. Springer, 2020, pp. 98–110.

[20] M. A. Naagas, A. R. Malicdem, and T. D. Palaoag, "Deh-dosv6: A defendable security model against ipv6 extension headers denial of service attack," *Bulletin of Electrical Engineering and Informatics*, vol. 10, no. 1, pp. 274–282, 2021.

[21] S. Becker, H. Abdelnur, R. State, and T. Engel, "An autonomic testing framework for ipv6 configuration protocols," in *Mechanisms for Autonomous Management of Net-*

works and Services, B. Stiller and F. De Turck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 65–76.

[22] H. Rafiee, C. Mueller, L. Niemeier, J. Streek, C. Sterz, and C. Meinel, "A flexible framework for detecting ipv6 vulnerabilities," in *Proceedings of the 6th International Conference on Security of Information and Networks*, ser. SIN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 196–202. [Online]. Available: https://doi.org/10.1145/2523514.2527001

[23] I. van Sprundel, "Fuzzing the tcp/ip stack," 2023. [Online]. Available: https://media.ccc.de/v/37c3-12235-fuzzing_the_tcp_ip_stack

[24] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.

[25] G. Huston, "IPv4 Address Report, daily generated," Archived from the original on 6 August 2011, 2011, retrieved 16 January 2011. [Online]. Available: https://www.potaroo.net/tools/ipv4/index.html

[26] APNIC, "APNIC's IPv4 pool usage," Archived from the original on 14 January 2011, 2011, retrieved 2 December 2011. [Online]. Available: https://web.archive.org/web/20110114193847/http://www.apnic.net/community/ipv4-exhaustion/graphical-information

[27] K. B. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC 1631, May 1994. [Online]. Available: https://www.rfc-editor.org/info/rfc1631

[28] J. M. <>, S. E. D. <>, J. M. <>, and B. Hinden, "Path MTU Discovery for IP version 6," RFC 8201, Jul. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8201

[29] "Domain names - implementation and specification," RFC 1035, Nov. 1987. [Online]. Available: https://www.rfc-editor.org/info/rfc1035

[30] J. Moy, "OSPF Version 2," RFC 2328, Apr. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2328

[31] D. Ferguson, A. Lindem, and J. Moy, "OSPF for IPv6," RFC 5340, Jul. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5340

[32] "CVE-2023-4809: Freebsd pf bypass when using ipv6," https://nvd.nist.gov/vuln/detail/CVE-2023-4809, 2023.

[33] M. Gupta and A. Conta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," RFC 4443, Mar. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4443

[34] OpenBSD, "PF: Scrub (Packet Normalization)," 2010, archived from the original on 23 December 2010. [Online]. Available: http://web.archive.org/web/20101223090933/http://www.openbsd.org/faq/pf/scrub.html

[35] D. T. Narten, T. Jinmei, and D. S. Thomson, "IPv6 Stateless Address Autoconfiguration," RFC 4862, Sep. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4862

[36] R. Bonica, D.-H. Gan, C. Pignataro, and D. Tappan, "Extended ICMP to Support Multi-Part Messages," RFC 4884, Apr. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4884

[37] G. Neville-Neil, P. Savola, and J. Abley, "Deprecation of Type 0 Routing Headers in IPv6," RFC 5095, Dec. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc5095

[38] T. Herbert, "ICMPv6 Errors for Discarding Packets Due to Processing Limits," RFC 8883, Sep. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8883

Acknowledgments

I would like to thank Mauro Conti, my supervisor, for giving me the opportunity to work on this topic. I am especially grateful to my co-supervisor, Enrico Bassetti, for his guidance and support throughout the thesis. Last but not least, I would like to thank my family and friends, whose encouragement has been a constant source of strength during my journey.